GUROBI OPTIMIZER REFERENCE MANUAL

Version 11.0, Copyright © 2023, Gurobi Optimization, LLC

Contents

1 Introduction **1**

2 Detailed Release Notes 11 **4**

3 C API Overview **10**

3.1 Environment Creation and Destruction . . . 16

GRBloadenv . . . 16
GRBemptyenv . . . 16
GRBstartenv . . . 16
GRBfreeenv . . . 17
GRBgetconcurrentenv . . . 17
GRBgetmultiobjenv . . . 18
GRBdiscardconcurrentenvs . . . 19
GRBdiscardmultiobjenvs . . . 19

3.2 Model Creation and Modification . . . 20

GRBloadmodel . . . 20
GRBnewmodel . . . 22
GRBcopymodel . . . 23
GRBcopymodeltoenv . . . 23
GRBaddconstr . . . 24
GRBaddconstrs . . . 25
GRBaddgenconstrXxx . . . 26
GRBaddgenconstrMax . . . 27
GRBaddgenconstrMin . . . 28
GRBaddgenconstrAbs . . . 28
GRBaddgenconstrAnd . . . 29
GRBaddgenconstrOr . . . 30
GRBaddgenconstrNorm . . . 31
GRBaddgenconstrIndicator . . . 31
GRBaddgenconstrPWL . . . 32
GRBaddgenconstrPoly . . . 33
GRBaddgenconstrExp . . . 34
GRBaddgenconstrExpA . . . 35
GRBaddgenconstrLog . . . 36
GRBaddgenconstrLogA . . . 37
GRBaddgenconstrLogistic . . . 38
GRBaddgenconstrPow . . . 38
GRBaddgenconstrSin . . . 39
GRBaddgenconstrCos . . . 40

GRBaddgenconstrTan . . . 41
GRBaddqconstr . . . 42
GRBaddqpterms . . . 43
GRBaddrangeconstr . . . 44
GRBaddrangeconstrs . . . 45
GRBaddsos . . . 47
GRBaddvar . . . 48
GRBaddvars . . . 49
GRBchgcoeffs . . . 50
GRBdelconstrs . . . 51
GRBdelgenconstrs . . . 51
GRBdelq . . . 51
GRBdelqconstrs . . . 52
GRBdelsos . . . 52
GRBdelvars . . . 53
GRBsetobjectiven . . . 53
GRBsetpwlobj . . . 54
GRBupdatemodel . . . 55
GRBfreemodel . . . 55
GRBXaddconstrs . . . 56
GRBXaddrangeconstrs . . . 57
GRBXaddvars . . . 58
GRBXchgcoeffs . . . 59
GRBXloadmodel . . . 60
3.3 Model Solution . . . 63
GRBoptimize . . . 63
GRBoptimizeasync . . . 63
GRBpresolvemodel . . . 64
GRBcomputeIIS . . . 64
GRBfeasrelax . . . 66
GRBfixmodel . . . 67
GRBreset . . . 67
GRBsync . . . 68
3.4 Model Queries . . . 69
GRBgetcoeff . . . 69
GRBgetconstrbyname . . . 69
GRBgetconstrs . . . 70
GRBgetenv . . . 70
GRBgetgenconstrMax . . . 71
GRBgetgenconstrMin . . . 72
GRBgetgenconstrAbs . . . 73
GRBgetgenconstrAnd . . . 73
GRBgetgenconstrOr . . . 74
GRBgetgenconstrNorm . . . 75
GRBgetgenconstrIndicator . . . 76

GRBgetgenconstrPWL . . . 78
GRBgetgenconstrPoly . . . 79
GRBgetgenconstrExp . . . 80
GRBgetgenconstrExpA . . . 80
GRBgetgenconstrLog . . . 81
GRBgetgenconstrLogA . . . 82
GRBgetgenconstrLogistic . . . 82
GRBgetgenconstrPow . . . 83
GRBgetgenconstrSin . . . 84
GRBgetgenconstrCos . . . 85
GRBgetgenconstrTan . . . 85
GRBgetjsonsolution . . . 86
GRBgetpwlobj . . . 86
GRBgetq . . . 87
GRBgetqconstr . . . 88
GRBgetqconstrbyname . . . 89
GRBgetsos . . . 89
GRBgetvarbyname . . . 90
GRBgetvars . . . 90
GRBsinglescenariomodel . . . 91
GRBXgetconstrs . . . 92
GRBXgetvars . . . 93
3.5 Input-Output . . . 94
GRBreadmodel . . . 94
GRBread . . . 94
GRBwrite . . . 95
3.6 Attribute Management . . . 96
GRBgetattrinfo . . . 96
GRBgetintattr . . . 96
GRBsetintattr . . . 97
GRBgetintattrelement . . . 97
GRBsetintattrelement . . . 98
GRBgetintattrarray . . . 98
GRBsetintattrarray . . . 99
GRBgetintattrlist . . . 100
GRBsetintattrlist . . . 100
GRBgetdblattr . . . 101
GRBsetdblattr . . . 101
GRBgetdblattrelement . . . 102
GRBsetdblattrelement . . . 102
GRBgetdblattrarray . . . 103
GRBsetdblattrarray . . . 103
GRBgetdblattrlist . . . 104
GRBsetdblattrlist . . . 105
GRBgetcharattrelement . . . 105

GRBsetcharattrelement . . . 106
GRBgetcharattrarray . . . 106
GRBsetcharattrarray . . . 107
GRBgetcharattrlist . . . 107
GRBsetcharattrlist . . . 108
GRBgetstrattr . . . 108
GRBsetstrattr . . . 109
GRBgetstrattrelement . . . 110
GRBsetstrattrelement . . . 110
GRBgetstrattrarray . . . 111
GRBsetstrattrarray . . . 111
GRBgetstrattrlist . . . 112
GRBsetstrattrlist . . . 113
GRBgetbatchattrinfo . . . 113
3.7 Parameter Management and Tuning . . . 115
GRBtunemodel . . . 115
GRBgettuneresult . . . 115
GRBgetdblparam . . . 116
GRBgetintparam . . . 116
GRBgetstrparam . . . 117
GRBsetdblparam . . . 117
GRBsetintparam . . . 118
GRBsetstrparam . . . 118
GRBgetdblparaminfo . . . 119
GRBgetintparaminfo . . . 120
GRBgetstrparaminfo . . . 120
GRBreadparams . . . 121
GRBwriteparams . . . 121
3.8 Monitoring Progress - Logging and Callbacks . . . 123
GRBmsg . . . 123
GRBsetcallbackfunc . . . 123
GRBgetcallbackfunc . . . 124
GRBcbget . . . 124
GRBversion . . . 125
3.9 Modifying Solver Behavior - Callbacks . . . 126
GRBcbcut . . . 126
GRBcblazy . . . 127
GRBcbsolution . . . 128
GRBcbproceed . . . 128
GRBcbstoponemultiobj . . . 129
GRBterminate . . . 130
3.10 Batch Requests . . . 131
GRBabortbatch . . . 131
GRBdiscardbatch . . . 131
GRBfreebatch . . . 131

GRBgetbatch . . . 132
GRBgetbatchenv . . . 132
GRBgetbatchintattr . . . 133
GRBgetbatchjsonsolution . . . 133
GRBgetbatchstrattr . . . 134
GRBoptimizebatch . . . 134
GRBretrybatch . . . 135
GRBupdatebatch . . . 135
GRBwritebatchjsonsolution . . . 136
3.11 Error Handling . . . 137
GRBgeterrormsg . . . 137
3.12 Advanced simplex routines . . . 138
GRBFSolve . . . 138
GRBBSolve . . . 138
GRBBinvColj . . . 139
GRBBinvRowi . . . 139
GRBgetBasisHead . . . 140

4 C++ API Overview **141**
4.1 GRBEnv . . . 147
GRBEnv() . . . 147
GRBEnv::get() . . . 148
GRBEnv::getErrorMsg() . . . 149
GRBEnv::getParamInfo() . . . 149
GRBEnv::message() . . . 150
GRBEnv::readParams() . . . 150
GRBEnv::resetParams() . . . 151
GRBEnv::set() . . . 151
GRBEnv::start() . . . 152
GRBEnv::writeParams() . . . 152
4.2 GRBModel . . . 154
GRBModel() . . . 154
GRBModel::addConstr() . . . 155
GRBModel::addConstrs() . . . 157
GRBModel::addGenConstrXxx() . . . 158
GRBModel::addQConstr() . . . 169
GRBModel::addRange() . . . 170
GRBModel::addRanges() . . . 171
GRBModel::addSOS() . . . 171
GRBModel::addVar() . . . 171
GRBModel::addVars() . . . 173
GRBModel::chgCoeff() . . . 174
GRBModel::chgCoeffs() . . . 175
GRBModel::computeIIS() . . . 175
GRBModel::discardConcurrentEnvs() . . . 176
GRBModel::discardMultiobjEnvs() . . . 176

GRBModel::feasRelax() . . . 176
GRBModel::fixedModel() . . . 178
GRBModel::get() . . . 179
GRBModel::getCoeff() . . . 183
GRBModel::getCol() . . . 183
GRBModel::getConcurrentEnv() . . . 183
GRBModel::getConstrByName() . . . 184
GRBModel::getConstrs() . . . 184
GRBModel::getGenConstrXxx() . . . 184
GRBModel::getGenConstrs() . . . 192
GRBModel::getJSONSolution() . . . 193
GRBModel::getMultiobjEnv() . . . 193
GRBModel::getObjective() . . . 193
GRBModel::getPWLObj() . . . 194
GRBModel::getQCRow() . . . 194
GRBModel::getQConstrs() . . . 195
GRBModel::getRow() . . . 195
GRBModel::getSOS() . . . 195
GRBModel::getSOSs() . . . 196
GRBModel::getTuneResult() . . . 196
GRBModel::getVarByName() . . . 196
GRBModel::getVars() . . . 196
GRBModel::optimize() . . . 197
GRBModel::optimizeasync() . . . 197
GRBModel::optimizeBatch() . . . 197
GRBModel::presolve() . . . 198
GRBModel::read() . . . 198
GRBModel::remove() . . . 198
GRBModel::reset() . . . 199
GRBModel::setCallback() . . . 200
GRBModel::set() . . . 200
GRBModel::setObjective() . . . 204
GRBModel::setObjectiveN() . . . 205
GRBModel::setPWLObj() . . . 205
GRBModel::singleScenarioModel() . . . 206
GRBModel::sync() . . . 206
GRBModel::terminate() . . . 206
GRBModel::tune() . . . 207
GRBModel::update() . . . 207
GRBModel::write() . . . 207
4.3 GRBVar . . . 208
GRBVar::get() . . . 208
GRBVar::index() . . . 209
GRBVar::sameAs() . . . 209
GRBVar::set() . . . 209

4.4 GRBConstr . . . 211
GRBConstr::get() . . . 211
GRBConstr::index() . . . 212
GRBConstr::sameAs() . . . 212
GRBConstr::set() . . . 212
4.5 GRBQConstr . . . 214
GRBQConstr::get() . . . 214
GRBQConstr::set() . . . 215
4.6 GRBSOS . . . 216
GRBSOS::get() . . . 216
GRBSOS::set() . . . 216
4.7 GRBGenConstr . . . 217
GRBGenConstr::get() . . . 217
GRBGenConstr::set() . . . 217
4.8 GRBExpr . . . 219
GRBExpr::getValue() . . . 219
4.9 GRBLinExpr . . . 220
GRBLinExpr() . . . 220
GRBLinExpr::addTerms() . . . 221
GRBLinExpr::clear() . . . 221
GRBLinExpr::getConstant() . . . 221
GRBLinExpr::getCoeff() . . . 222
GRBLinExpr::getValue() . . . 222
GRBLinExpr::getVar() . . . 222
GRBLinExpr::operator= . . . 222
GRBLinExpr::operator+ . . . 223
GRBLinExpr::operator- . . . 223
GRBLinExpr::operator+= . . . 223
GRBLinExpr::operator-= . . . 223
GRBLinExpr::operator*= . . . 224
GRBLinExpr::remove() . . . 224
GRBLinExpr::size() . . . 224
4.10 GRBQuadExpr . . . 225
GRBQuadExpr() . . . 226
GRBQuadExpr::addTerm() . . . 226
GRBQuadExpr::addTerms() . . . 227
GRBQuadExpr::clear() . . . 227
GRBQuadExpr::getCoeff() . . . 227
GRBQuadExpr::getLinExpr() . . . 228
GRBQuadExpr::getValue() . . . 228
GRBQuadExpr::getVar1() . . . 228
GRBQuadExpr::getVar2() . . . 228
GRBQuadExpr::operator= . . . 229
GRBQuadExpr::operator+ . . . 229
GRBQuadExpr::operator- . . . 229

GRBQuadExpr::operator+= . . . 229
GRBQuadExpr::operator-= . . . 230
GRBQuadExpr::operator*= . . . 230
GRBQuadExpr::remove() . . . 230
GRBQuadExpr::size() . . . 230
4.11 GRBTempConstr . . . 231
4.12 GRBColumn . . . 232
GRBColumn() . . . 232
GRBColumn::addTerm() . . . 232
GRBColumn::addTerms() . . . 232
GRBColumn::clear() . . . 232
GRBColumn::getCoeff() . . . 233
GRBColumn::getConstr() . . . 233
GRBColumn::remove() . . . 233
GRBColumn::size() . . . 233
4.13 GRBCallback . . . 234
GRBCallback() . . . 234
GRBCallback::abort() . . . 234
GRBCallback::addCut() . . . 234
GRBCallback::addLazy() . . . 235
GRBCallback::getDoubleInfo() . . . 236
GRBCallback::getIntInfo() . . . 236
GRBCallback::getNodeRel() . . . 237
GRBCallback::getSolution() . . . 237
GRBCallback::getStringInfo() . . . 238
GRBCallback::proceed() . . . 238
GRBCallback::setSolution() . . . 238
GRBCallback::stopOneMultiObj() . . . 239
GRBCallback::useSolution() . . . 240
4.14 GRBException . . . 241
GRBException() . . . 241
GRBException::getErrorCode() . . . 241
GRBException::getMessage() . . . 241
4.15 GRBBatch . . . 242
GRBBatch() . . . 242
GRBBatch::abort() . . . 243
GRBBatch::discard() . . . 243
GRBBatch::getJSONSolution() . . . 243
GRBBatch::get() . . . 243
GRBBatch::retry() . . . 244
GRBBatch::update() . . . 244
GRBBatch::writeJSONSolution() . . . 245
4.16 Non-Member Functions . . . 246
operator== . . . 246
operator<= . . . 246

operator>= . . . 246
operator+ . . . 247
operator- . . . 248
operator* . . . 249
operator/ . . . 251
4.17 Attribute Enums . . . 252
GRB_CharAttr . . . 252
GRB_DoubleAttr . . . 252
GRB_IntAttr . . . 252
GRB_StringAttr . . . 252
4.18 Parameter Enums . . . 253
GRB_DoubleParam . . . 253
GRB_IntParam . . . 253
GRB_StringParam . . . 253

5 Java API Overview **254**
5.1 GRBEnv . . . 261
GRBEnv() . . . 261
GRBEnv.dispose() . . . 262
GRBEnv.get() . . . 262
GRBEnv.getErrorMsg() . . . 263
GRBEnv.getParamInfo() . . . 263
GRBEnv.message() . . . 264
GRBEnv.readParams() . . . 264
GRBEnv.release() . . . 265
GRBEnv.resetParams() . . . 265
GRBEnv.set() . . . 265
GRBEnv.setLogCallback() . . . 266
GRBEnv.start() . . . 266
GRBEnv.writeParams() . . . 267
5.2 GRBModel . . . 268
GRBModel() . . . 268
GRBModel.addConstr() . . . 269
GRBModel.addConstrs() . . . 272
GRBModel.addGenConstrXxx() . . . 273
GRBModel.addQConstr() . . . 283
GRBModel.addRange() . . . 286
GRBModel.addRanges() . . . 286
GRBModel.addSOS() . . . 287
GRBModel.addVar() . . . 287
GRBModel.addVars() . . . 288
GRBModel.chgCoeff() . . . 290
GRBModel.chgCoeffs() . . . 291
GRBModel.computeIIS() . . . 291
GRBModel.discardConcurrentEnvs() . . . 292
GRBModel.discardMultiobjEnvs() . . . 292

GRBModel.dispose() . . . 293
GRBModel.feasRelax() . . . 293
GRBModel.fixedModel() . . . 295
GRBModel.get() . . . 295
GRBModel.getCoeff() . . . 310
GRBModel.getCol() . . . 310
GRBModel.getConcurrentEnv() . . . 310
GRBModel.getConstrByName() . . . 311
GRBModel.getConstrs() . . . 311
GRBModel.getGenConstrXxx() . . . 311
GRBModel.getGenConstrs() . . . 319
GRBModel.getJSONSolution() . . . 319
GRBModel.getMultiobjEnv() . . . 320
GRBModel.getObjective() . . . 320
GRBModel.getPWLObj() . . . 321
GRBModel.getQCRow() . . . 321
GRBModel.getQConstrs() . . . 321
GRBModel.getRow() . . . 321
GRBModel.getSOS() . . . 322
GRBModel.getSOSs() . . . 322
GRBModel.getTuneResult() . . . 322
GRBModel.getVarByName() . . . 323
GRBModel.getVars() . . . 323
GRBModel.optimize() . . . 323
GRBModel.optimizeasync() . . . 324
GRBModel.optimizeBatch() . . . 324
GRBModel.presolve() . . . 324
GRBModel.read() . . . 325
GRBModel.remove() . . . 325
GRBModel.reset() . . . 326
GRBModel.setCallback() . . . 326
GRBModel.set() . . . 327
GRBModel.setLogCallback() . . . 340
GRBModel.setObjective() . . . 340
GRBModel.setObjectiveN() . . . 341
GRBModel.setPWLObj() . . . 341
GRBModel.singleScenarioModel() . . . 342
GRBModel.sync() . . . 342
GRBModel.terminate() . . . 342
GRBModel.tune() . . . 343
GRBModel.update() . . . 343
GRBModel.write() . . . 343
5.3 GRBVar . . . 344
GRBVar.get() . . . 344
GRBVar.index() . . . 345

GRBVar.sameAs() . . . 345
GRBVar.set() . . . 345
5.4 GRBConstr . . . 347
GRBConstr.get() . . . 347
GRBConstr.index() . . . 348
GRBConstr.sameAs() . . . 348
GRBConstr.set() . . . 348
5.5 GRBQConstr . . . 350
GRBQConstr.get() . . . 350
GRBQConstr.set() . . . 351
5.6 GRBSOS . . . 352
GRBSOS.get() . . . 352
GRBSOS.set() . . . 352
5.7 GRBGenConstr . . . 353
GRBGenConstr.get() . . . 353
GRBGenConstr.set() . . . 353
5.8 GRBExpr . . . 355
GRBExpr.getValue() . . . 355
5.9 GRBLinExpr . . . 356
GRBLinExpr() . . . 356
GRBLinExpr.add() . . . 356
GRBLinExpr.addConstant() . . . 357
GRBLinExpr.addTerm() . . . 357
GRBLinExpr.addTerms() . . . 357
GRBLinExpr.clear() . . . 358
GRBLinExpr.getConstant() . . . 358
GRBLinExpr.getCoeff() . . . 358
GRBLinExpr.getValue() . . . 358
GRBLinExpr.getVar() . . . 359
GRBLinExpr.multAdd() . . . 359
GRBLinExpr.remove() . . . 359
GRBLinExpr.size() . . . 360
5.10 GRBQuadExpr . . . 361
GRBQuadExpr() . . . 361
GRBQuadExpr.add() . . . 362
GRBQuadExpr.addConstant() . . . 362
GRBQuadExpr.addTerm() . . . 362
GRBQuadExpr.addTerms() . . . 363
GRBQuadExpr.clear() . . . 364
GRBQuadExpr.getCoeff() . . . 364
GRBQuadExpr.getLinExpr() . . . 365
GRBQuadExpr.getValue() . . . 365
GRBQuadExpr.getVar1() . . . 365
GRBQuadExpr.getVar2() . . . 365
GRBQuadExpr.multAdd() . . . 366

GRBQuadExpr.remove() . . . 366
GRBQuadExpr.size() . . . 366
5.11 GRBColumn . . . 367
GRBColumn() . . . 367
GRBColumn.addTerm() . . . 367
GRBColumn.addTerms() . . . 367
GRBColumn.clear() . . . 368
GRBColumn.getCoeff() . . . 368
GRBColumn.getConstr() . . . 368
GRBColumn.remove() . . . 369
GRBColumn.size() . . . 369
5.12 GRBCallback . . . 370
GRBCallback() . . . 370
GRBCallback.abort() . . . 370
GRBCallback.addCut() . . . 370
GRBCallback.addLazy() . . . 371
GRBCallback.getDoubleInfo() . . . 372
GRBCallback.getIntInfo() . . . 372
GRBCallback.getNodeRel() . . . 372
GRBCallback.getSolution() . . . 373
GRBCallback.getStringInfo() . . . 374
GRBCallback.proceed() . . . 374
GRBCallback.setSolution() . . . 374
GRBCallback.stopOneMultiObj() . . . 375
GRBCallback.useSolution() . . . 376
5.13 GRBException . . . 377
GRBException() . . . 377
GRBException.getErrorCode() . . . 377
5.14 GRBBatch . . . 378
GRBBatch() . . . 378
GRBBatch.abort() . . . 379
GRBBatch.discard() . . . 379
GRBBatch.dispose() . . . 379
GRBBatch.getJSONSolution() . . . 380
GRBBatch.get() . . . 380
GRBBatch.retry() . . . 380
GRBBatch.update() . . . 381
GRBBatch.writeJSONSolution() . . . 381
5.15 GRB . . . 382
Constants . . . 382
GRB.CharAttr . . . 388
GRB.DoubleAttr . . . 388
GRB.DoubleParam . . . 388
GRB.IntAttr . . . 388
GRB.IntParam . . . 389

GRB.StringAttr . . . 389
GRB.StringParam . . . 389

6 .NET API Overview 390
6.1 GRBEnv . . . 397
GRBEnv() . . . 397
GRBEnv.Dispose() . . . 398
GRBEnv.ErrorMsg . . . 398
GRBEnv.Get() . . . 398
GRBEnv.GetParamInfo() . . . 399
GRBEnv.Message() . . . 400
GRBEnv.ReadParams() . . . 400
GRBEnv.Release() . . . 401
GRBEnv.ResetParams() . . . 401
GRBEnv.Set() . . . 401
GRBEnv.Start() . . . 402
GRBEnv.WriteParams() . . . 402
6.2 GRBModel . . . 403
GRBModel() . . . 403
GRBModel.AddConstr() . . . 404
GRBModel.AddConstrs() . . . 405
GRBModel.AddGenConstrXxx() . . . 406
GRBModel.AddQConstr() . . . 417
GRBModel.AddRange() . . . 417
GRBModel.AddRanges() . . . 418
GRBModel.AddSOS() . . . 419
GRBModel.AddVar() . . . 419
GRBModel.AddVars() . . . 420
GRBModel.ChgCoeff() . . . 422
GRBModel.ChgCoeffs() . . . 423
GRBModel.ComputeIIS() . . . 423
GRBModel.DiscardConcurrentEnvs() . . . 424
GRBModel.DiscardMultiobjEnvs() . . . 424
GRBModel.Dispose() . . . 424
GRBModel.FeasRelax() . . . 424
GRBModel.FixedModel() . . . 426
GRBModel.Get() . . . 426
GRBModel.GetCoeff() . . . 438
GRBModel.GetCol() . . . 438
GRBModel.GetConcurrentEnv() . . . 439
GRBModel.GetConstrByName() . . . 439
GRBModel.GetConstrs() . . . 439
GRBModel.GetGenConstrXxx() . . . 439
GRBModel.GetGenConstrs() . . . 447
GRBModel.GetJSONSolution() . . . 447
GRBModel.GetMultiobjEnv() . . . 447

GRBModel.GetObjective() . . . 448
GRBModel.GetPWLObj() . . . 448
GRBModel.GetQConstr() . . . 448
GRBModel.GetQConstrs() . . . 449
GRBModel.GetQCRow() . . . 449
GRBModel.GetRow() . . . 449
GRBModel.GetSOS() . . . 449
GRBModel.GetSOSs() . . . 450
GRBModel.GetTuneResult() . . . 450
GRBModel.GetVarByName() . . . 450
GRBModel.GetVars() . . . 451
GRBModel.Optimize() . . . 451
GRBModel.OptimizeAsync() . . . 451
GRBModel.OptimizeBatch() . . . 451
GRBModel.Presolve() . . . 452
GRBModel.Read() . . . 452
GRBModel.Remove() . . . 452
GRBModel.Reset() . . . 453
GRBModel.SetCallback() . . . 453
GRBModel.Set() . . . 454
GRBModel.SetObjective() . . . 466
GRBModel.SetObjectiveN() . . . 466
GRBModel.SetPWLObj() . . . 467
GRBModel.SingleScenarioModel() . . . 467
GRBModel.Sync() . . . 467
GRBModel.Terminate() . . . 468
GRBModel.Tune() . . . 468
GRBModel.Update() . . . 468
GRBModel.Write() . . . 468
6.3 GRBVar . . . 470
GRBVar.Get() . . . 470
GRBVar.Index . . . 470
GRBVar.SameAs() . . . 471
GRBVar.Set() . . . 471
6.4 GRBConstr . . . 472
GRBConstr.Get() . . . 472
GRBConstr.Index . . . 472
GRBConstr.SameAs() . . . 473
GRBConstr.Set() . . . 473
6.5 GRBQConstr . . . 475
GRBQConstr.Get() . . . 475
GRBQConstr.Set() . . . 475
6.6 GRBSOS . . . 477
GRBSOS.Get() . . . 477
GRBSOS.Set() . . . 477

6.7 GRBGenConstr . . . 478
GRBGenConstr.Get() . . . 478
GRBGenConstr.Set() . . . 478
6.8 GRBExpr . . . 480
GRBExpr.Value . . . 480
6.9 GRBLinExpr . . . 481
GRBLinExpr() . . . 481
GRBLinExpr.Add() . . . 482
GRBLinExpr.AddConstant() . . . 482
GRBLinExpr.AddTerm() . . . 482
GRBLinExpr.AddTerms() . . . 482
GRBLinExpr.Clear() . . . 483
GRBLinExpr.Constant . . . 483
GRBLinExpr.GetCoeff() . . . 483
GRBLinExpr.GetVar() . . . 483
GRBLinExpr.MultAdd() . . . 484
GRBLinExpr.Remove() . . . 484
GRBLinExpr.Size . . . 484
GRBLinExpr.Value . . . 484
6.10 GRBQuadExpr . . . 485
GRBQuadExpr() . . . 486
GRBQuadExpr.Add() . . . 486
GRBQuadExpr.AddConstant() . . . 486
GRBQuadExpr.AddTerm() . . . 487
GRBQuadExpr.AddTerms() . . . 487
GRBQuadExpr.Clear() . . . 488
GRBQuadExpr.GetCoeff() . . . 488
GRBQuadExpr.GetVar1() . . . 489
GRBQuadExpr.GetVar2() . . . 489
GRBQuadExpr.LinExpr() . . . 489
GRBQuadExpr.MultAdd() . . . 489
GRBQuadExpr.Remove() . . . 490
GRBQuadExpr.Size . . . 490
GRBQuadExpr.Value . . . 490
6.11 GRBTempConstr . . . 491
6.12 GRBColumn . . . 492
GRBColumn() . . . 492
GRBColumn.AddTerm() . . . 492
GRBColumn.AddTerms() . . . 492
GRBColumn.Clear() . . . 493
GRBColumn.GetCoeff() . . . 493
GRBColumn.GetConstr() . . . 493
GRBColumn.Remove() . . . 493
GRBColumn.Size . . . 494
6.13 Overloaded Operators . . . 495

operator <= . . . 495
operator >= . . . 495
operator == . . . 495
operator + . . . 496
operator - . . . 497
operator * . . . 498
operator / . . . 500
implicit cast . . . 500
6.14 GRBCallback . . . 502
GRBCallback() . . . 502
GRBCallback.Abort() . . . 502
GRBCallback.AddCut() . . . 502
GRBCallback.AddLazy() . . . 503
GRBCallback.GetDoubleInfo() . . . 504
GRBCallback.GetIntInfo() . . . 504
GRBCallback.GetNodeRel() . . . 504
GRBCallback.GetSolution() . . . 505
GRBCallback.GetStringInfo() . . . 505
GRBCallback.Proceed() . . . 506
GRBCallback.SetSolution() . . . 506
GRBCallback.StopOneMultiObj() . . . 506
GRBCallback.UseSolution() . . . 507
6.15 GRBException . . . 509
GRBException() . . . 509
GRBException.ErrorCode . . . 509
6.16 GRBBatch . . . 510
GRBBatch() . . . 510
GRBBatch.Abort() . . . 511
GRBBatch.Discard() . . . 511
GRBBatch.GetJSONSolution() . . . 511
GRBBatch.Get() . . . 511
GRBBatch.Retry() . . . 512
GRBBatch.Update() . . . 512
GRBBatch.WriteJSONSolution() . . . 512
6.17 GRB . . . 513
Constants . . . 513
GRB.CharAttr . . . 529
GRB.DoubleAttr . . . 529
GRB.DoubleParam . . . 529
GRB.IntAttr . . . 529
GRB.IntParam . . . 529
GRB.StringAttr . . . 529
GRB.StringParam . . . 529

7 Python API Overview **530**
7.1 Global Functions . . . 536
disposeDefaultEnv() . . . 536
multidict() . . . 536
paramHelp() . . . 537
quicksum() . . . 537
read() . . . 537
readParams() . . . 538
resetParams() . . . 538
setParam() . . . 538
writeParams() . . . 539
7.2 Model . . . 540
Model() . . . 540
Model.addConstr() . . . 540
Model.addConstrs() . . . 541
Model.addGenConstrXxx() . . . 542
Model.addGenConstrMax() . . . 543
Model.addGenConstrMin() . . . 544
Model.addGenConstrAbs() . . . 544
Model.addGenConstrAnd() . . . 545
Model.addGenConstrOr() . . . 546
Model.addGenConstrNorm() . . . 546
Model.addGenConstrIndicator() . . . 547
Model.addGenConstrPWL() . . . 548
Model.addGenConstrPoly() . . . 549
Model.addGenConstrExp() . . . 549
Model.addGenConstrExpA() . . . 550
Model.addGenConstrLog() . . . 550
Model.addGenConstrLogA() . . . 551
Model.addGenConstrLogistic() . . . 552
Model.addGenConstrPow() . . . 552
Model.addGenConstrSin() . . . 553
Model.addGenConstrCos() . . . 553
Model.addGenConstrTan() . . . 554
Model.addLConstr() . . . 555
Model.addMConstr() . . . 555
Model.addMQConstr() . . . 556
Model.addMVar() . . . 557
Model.addQConstr() . . . 557
Model.addRange() . . . 558
Model.addSOS() . . . 559
Model.addVar() . . . 559
Model.addVars() . . . 560
Model.cbCut() . . . 561
Model.cbGet() . . . 562

Model.cbGetNodeRel() . . . 562
Model.cbGetSolution() . . . 563
Model.cbLazy() . . . 563
Model.cbProceed() . . . 564
Model.cbSetSolution() . . . 565
Model.cbStopOneMultiObj() . . . 565
Model.cbUseSolution() . . . 566
Model.chgCoeff() . . . 566
Model.close() . . . 567
Model.computeIIS() . . . 567
Model.copy() . . . 568
Model.discardConcurrentEnvs() . . . 569
Model.discardMultiobjEnvs() . . . 569
Model.dispose() . . . 569
Model.feasRelaxS() . . . 570
Model.feasRelax() . . . 571
Model.fixed() . . . 572
Model.getA() . . . 572
Model.getAttr() . . . 573
Model.getCoeff() . . . 573
Model.getCol() . . . 573
Model.getConcurrentEnv() . . . 574
Model.getConstrByName() . . . 574
Model.getConstrs() . . . 575
Model.getGenConstrMax() . . . 575
Model.getGenConstrMin() . . . 575
Model.getGenConstrAbs() . . . 576
Model.getGenConstrAnd() . . . 576
Model.getGenConstrOr() . . . 577
Model.getGenConstrNorm() . . . 577
Model.getGenConstrIndicator() . . . 578
Model.getGenConstrPWL() . . . 578
Model.getGenConstrPoly() . . . 579
Model.getGenConstrExp() . . . 579
Model.getGenConstrExpA() . . . 580
Model.getGenConstrLog() . . . 580
Model.getGenConstrLogA() . . . 580
Model.getGenConstrLogistic() . . . 581
Model.getGenConstrPow() . . . 581
Model.getGenConstrSin() . . . 582
Model.getGenConstrCos() . . . 582
Model.getGenConstrTan() . . . 583
Model.getGenConstrs() . . . 583
Model.getJSONSolution() . . . 583
Model.getMultiobjEnv() . . . 584

Model.getObjective() . . . 584
Model.getParamInfo() . . . 585
Model.getPWLObj() . . . 585
Model.getQConstrs() . . . 585
Model.getQCRow() . . . 586
Model.getRow() . . . 586
Model.getSOS() . . . 586
Model.getSOSs() . . . 587
Model.getTuneResult() . . . 587
Model.getVarByName() . . . 587
Model.getVars() . . . 588
Model.message() . . . 588
Model.optimize() . . . 588
Model.optimizeBatch() . . . 588
Model.presolve() . . . 589
Model.printAttr() . . . 589
Model.printQuality() . . . 590
Model.printStats() . . . 590
Model.read() . . . 590
Model.relax() . . . 591
Model.remove() . . . 591
Model.reset() . . . 591
Model.resetParams() . . . 591
Model.setAttr() . . . 592
Model.setMObjective() . . . 592
Model.setObjective() . . . 593
Model.setObjectiveN() . . . 594
Model.setPWLObj() . . . 594
Model.setParam() . . . 595
Model.singleScenarioModel() . . . 595
Model.terminate() . . . 595
Model.tune() . . . 596
Model.update() . . . 596
Model.write() . . . 596
7.3 Var . . . 598
Var.getAttr() . . . 598
Var.sameAs() . . . 598
Var.index . . . 599
Var.setAttr() . . . 599
7.4 MVar . . . 600
MVar.copy() . . . 600
MVar.diagonal() . . . 601
MVar.fromlist() . . . 601
MVar.fromvar() . . . 602
MVar.getAttr() . . . 602

MVar.item() . . . 602
MVar.ndim . . . 602
MVar.reshape() . . . 603
MVar.setAttr() . . . 603
MVar.shape . . . 604
MVar.size . . . 604
MVar.sum() . . . 604
MVar.T . . . 605
MVar.tolist() . . . 605
MVar.transpose() . . . 605
7.5 Constr . . . 606
Constr.getAttr() . . . 606
Constr.index . . . 606
Constr.sameAs() . . . 607
Constr.setAttr() . . . 607
7.6 MConstr . . . 608
MConstr.fromlist() . . . 608
MConstr.getAttr() . . . 608
MConstr.setAttr() . . . 609
MConstr.tolist() . . . 609
7.7 MQConstr . . . 611
MQConstr.fromlist() . . . 611
MQConstr.getAttr() . . . 611
MQConstr.setAttr() . . . 612
MQConstr.tolist() . . . 612
7.8 QConstr . . . 614
QConstr.getAttr() . . . 614
QConstr.setAttr() . . . 614
7.9 SOS . . . 616
SOS.getAttr() . . . 616
SOS.index . . . 616
SOS.setAttr() . . . 616
7.10 GenConstr . . . 618
GenConstr.getAttr() . . . 618
GenConstr.setAttr() . . . 618
7.11 MGenConstr . . . 618
MGenConstr.fromlist() . . . 619
MGenConstr.getAttr() . . . 619
MGenConstr.setAttr() . . . 620
MGenConstr.tolist() . . . 620
7.12 LinExpr . . . 621
LinExpr() . . . 621
LinExpr.add() . . . 622
LinExpr.addConstant() . . . 622
LinExpr.addTerms() . . . 622

LinExpr.clear() . . . 623
LinExpr.copy() . . . 623
LinExpr.getConstant() . . . 623
LinExpr.getCoeff() . . . 623
LinExpr.getValue() . . . 624
LinExpr.getVar() . . . 624
LinExpr.remove() . . . 624
LinExpr.size() . . . 624
LinExpr.__eq__() . . . 625
LinExpr.__le__() . . . 625
LinExpr.__ge__() . . . 625
7.13 QuadExpr . . . 626
QuadExpr() . . . 626
QuadExpr.add() . . . 627
QuadExpr.addConstant() . . . 627
QuadExpr.addTerms() . . . 627
QuadExpr.clear() . . . 628
QuadExpr.copy() . . . 628
QuadExpr.getCoeff() . . . 628
QuadExpr.getLinExpr() . . . 628
QuadExpr.getValue() . . . 629
QuadExpr.getVar1() . . . 629
QuadExpr.getVar2() . . . 629
QuadExpr.remove() . . . 629
QuadExpr.size() . . . 630
QuadExpr.__eq__() . . . 630
QuadExpr.__le__() . . . 630
QuadExpr.__ge__() . . . 630
7.14 GenExpr . . . 631
7.15 MLinExpr . . . 632
MLinExpr.clear() . . . 632
MLinExpr.copy() . . . 632
MLinExpr.getValue() . . . 633
MLinExpr.item() . . . 633
MLinExpr.ndim . . . 633
MLinExpr.shape . . . 634
MLinExpr.size . . . 634
MLinExpr.sum() . . . 634
MLinExpr.zeros() . . . 634
MLinExpr.__eq__() . . . 635
MLinExpr.__ge__() . . . 635
MLinExpr.__getitem__() . . . 635
MLinExpr.__le__() . . . 636
MLinExpr.__setitem__() . . . 636
7.16 MQuadExpr . . . 637

MQuadExpr.clear() . . . 637
MQuadExpr.copy() . . . 637
MQuadExpr.getValue() . . . 638
MQuadExpr.item() . . . 638
MQuadExpr.ndim . . . 638
MQuadExpr.shape . . . 638
MQuadExpr.size . . . 639
MQuadExpr.sum() . . . 639
MQuadExpr.zeros() . . . 639
MQuadExpr.__eq__() . . . 640
MQuadExpr.__ge__() . . . 640
MQuadExpr.__getitem__() . . . 640
MQuadExpr.__le__() . . . 641
MQuadExpr.__setitem__() . . . 641
7.17 TempConstr . . . 642
7.18 Column . . . 644
Column() . . . 644
Column.addTerms() . . . 644
Column.clear() . . . 645
Column.copy() . . . 645
Column.getCoeff() . . . 645
Column.getConstr() . . . 645
Column.remove() . . . 645
Column.size() . . . 646
7.19 Callbacks . . . 647
7.20 GurobiError . . . 648
7.21 Env . . . 649
Env() . . . 649
Env.close() . . . 650
Env.dispose() . . . 651
Env.getParam() . . . 651
Env.resetParams() . . . 651
Env.setParam() . . . 651
Env.start() . . . 652
Env.writeParams() . . . 652
7.22 Batch . . . 653
Batch() . . . 653
Batch.abort() . . . 654
Batch.discard() . . . 654
Batch.dispose() . . . 654
Batch.getJSONSolution() . . . 654
Batch.retry() . . . 655
Batch.update() . . . 655
Batch.writeJSONSolution() . . . 655
7.23 GRB . . . 657

Constants . . . 657
GRB.Attr . . . 659
GRB.Param . . . 659
7.24 tuplelist . . . 660
tuplelist() . . . 660
tuplelist.select() . . . 660
tuplelist.clean() . . . 661
tuplelist.___contains___() . . . 661
7.25 tupledict . . . 662
tupledict() . . . 662
tupledict.select() . . . 662
tupledict.sum() . . . 663
tupledict.prod() . . . 663
tupledict.clean() . . . 664
7.26 General Constraint Helper Functions . . . 665
abs_() . . . 665
and_() . . . 665
max_() . . . 665
min_() . . . 666
or_() . . . 666
norm() . . . 667
7.27 Matrix-Friendly API Functions . . . 668
hstack() . . . 668
vstack() . . . 668
concatenate() . . . 668

8 MATLAB API Overview 670
8.1 Common Arguments . . . 673
The model argument . . . 673
The params argument . . . 689
8.2 Solving a Model . . . 692
gurobi() . . . 692
gurobi_iis() . . . 695
gurobi_feasrelax() . . . 697
gurobi_relax() . . . 698
8.3 Input-Output . . . 699
gurobi_read() . . . 699
gurobi_write() . . . 699
8.4 Using Gurobi within MATLAB's Problem-Based Optimization . . . 700
8.5 Setting up the Gurobi MATLAB interface . . . 702

9 R API Overview 703
9.1 Common Arguments . . . 706
The model argument . . . 706
The params argument . . . 722
9.2 Solving a Model . . . 724

gurobi() . . . 724
gurobi_iis() . . . 728
gurobi_feasrelax() . . . 730
gurobi_relax() . . . 731
9.3 Input-Output . . . 731
gurobi_read() . . . 731
gurobi_write() . . . 732
9.4 Installing the R package . . . 732

10 Variables and Constraints and Objectives **734**
10.1 Variables . . . 734
10.2 Constraints . . . 735
Linear Constraints . . . 735
SOS Constraints . . . 736
Quadratic Constraints . . . 737
General Constraints . . . 737
10.3 Objectives . . . 745
10.4 Tolerances and Ill Conditioning -- A Caveat . . . 750

11 Environments **752**
11.1 Session boundaries . . . 752
11.2 Configuration parameters . . . 753
11.3 Algorithmic parameters . . . 755
11.4 Concurrent environments . . . 756
11.5 Multi-objective environments . . . 756

12 Attributes **758**
12.1 Model Attributes . . . 764
NumConstrs . . . 764
NumVars . . . 764
NumSOS . . . 765
NumQConstrs . . . 765
NumGenConstrs . . . 765
NumNZs . . . 765
DNumNZs . . . 765
NumQNZs . . . 765
NumQCNZs . . . 766
NumIntVars . . . 766
NumBinVars . . . 766
NumPWLObjVars . . . 766
ModelName . . . 766
ModelSense . . . 766
ObjCon . . . 767
Fingerprint . . . 767
ObjVal . . . 767
ObjBound . . . 767

ObjBoundC . . . 767
PoolObjBound . . . 768
PoolObjVal . . . 768
MIPGap . . . 768
Runtime . . . 768
Work . . . 769
Status . . . 769
SolCount . . . 769
IterCount . . . 769
BarIterCount . . . 769
NodeCount . . . 769
ConcurrentWinMethod . . . 770
IsMIP . . . 770
IsQP . . . 770
IsQCP . . . 770
IsMultiObj . . . 770
IISMinimal . . . 771
MaxCoeff . . . 771
MinCoeff . . . 771
MaxBound . . . 771
MinBound . . . 771
MaxObjCoeff . . . 771
MinObjCoeff . . . 772
MaxRHS . . . 772
MinRHS . . . 772
MaxQCCoeff . . . 772
MinQCCoeff . . . 772
MaxQCLCoeff . . . 772
MinQCLCoeff . . . 773
MaxQCRHS . . . 773
MinQCRHS . . . 773
MaxQObjCoeff . . . 773
MinQObjCoeff . . . 773
OpenNodeCount . . . 773
Kappa . . . 774
KappaExact . . . 774
FarkasProof . . . 774
TuneResultCount . . . 775
NumStart . . . 775
LicenseExpiration . . . 775
12.2 Variable Attributes . . . 775
LB . . . 775
UB . . . 776
Obj . . . 776
VarName . . . 776

VTag . . . 776
VType . . . 777
X . . . 777
Xn . . . 777
RC . . . 777
BarX . . . 777
Start . . . 778
VarHintVal . . . 778
VarHintPri . . . 779
BranchPriority . . . 779
Partition . . . 780
VBasis . . . 780
PStart . . . 780
IISLB . . . 781
IISLBForce . . . 781
IISUB . . . 782
IISUBForce . . . 782
PoolIgnore . . . 782
PWLObjCvx . . . 783
SAObjLow . . . 783
SAObjUp . . . 783
SALBLow . . . 783
SALBUp . . . 783
SAUBLow . . . 783
SAUBUp . . . 784
UnbdRay . . . 784
12.3 Linear Constraint Attributes . . . 784
Sense . . . 784
RHS . . . 784
ConstrName . . . 785
CTag . . . 785
Pi . . . 785
Slack . . . 786
CBasis . . . 786
DStart . . . 786
Lazy . . . 787
IISConstr . . . 787
IISConstrForce . . . 788
SARHSLow . . . 788
SARHSUp . . . 788
FarkasDual . . . 788
12.4 SOS Attributes . . . 789
IISSOS . . . 789
IISSOSForce . . . 790
12.5 Quadratic Constraint Attributes . . . 790

QCSense . . . 790
QCRHS . . . 790
QCName . . . 791
QCPi . . . 791
QCSlack . . . 791
QCTag . . . 791
IISQConstr . . . 791
IISQConstrForce . . . 792
12.6 General Constraint Attributes . . . 792
FuncPieceError . . . 792
FuncPieceLength . . . 792
FuncPieceRatio . . . 793
FuncPieces . . . 793
FuncNonlinear . . . 793
GenConstrType . . . 794
GenConstrName . . . 794
IISGenConstr . . . 794
IISGenConstrForce . . . 794
12.7 Quality Attributes . . . 794
MaxVio . . . 795
BoundVio . . . 795
BoundSVio . . . 795
BoundVioIndex . . . 795
BoundSVioIndex . . . 795
BoundVioSum . . . 796
BoundSVioSum . . . 796
ConstrVio . . . 796
ConstrSVio . . . 796
ConstrVioIndex . . . 796
ConstrSVioIndex . . . 797
ConstrVioSum . . . 797
ConstrSVioSum . . . 797
ConstrResidual . . . 797
ConstrSResidual . . . 797
ConstrResidualIndex . . . 798
ConstrSResidualIndex . . . 798
ConstrResidualSum . . . 798
ConstrSResidualSum . . . 798
DualVio . . . 798
DualSVio . . . 799
DualVioIndex . . . 799
DualSVioIndex . . . 799
DualVioSum . . . 799
DualSVioSum . . . 799
DualResidual . . . 800

DualSResidual . . . 800
DualResidualIndex . . . 800
DualSResidualIndex . . . 800
DualResidualSum . . . 800
DualSResidualSum . . . 801
ComplVio . . . 801
ComplVioIndex . . . 801
ComplVioSum . . . 801
IntVio . . . 801
IntVioIndex . . . 802
IntVioSum . . . 802
12.8 Multi-objective Attributes . . . 802
ObjN . . . 802
ObjNCon . . . 802
ObjNPriority . . . 803
ObjNWeight . . . 803
ObjNRelTol . . . 803
ObjNAbsTol . . . 803
ObjNVal . . . 804
ObjNName . . . 804
NumObj . . . 804
12.9 Multi-Scenario Attributes . . . 805
ScenNLB . . . 805
ScenNUB . . . 805
ScenNObj . . . 806
ScenNRHS . . . 806
ScenNName . . . 806
ScenNObjBound . . . 806
ScenNObjVal . . . 807
ScenNX . . . 807
NumScenarios . . . 807
12.10Batch Attributes . . . 808
BatchErrorCode . . . 808
BatchErrorMessage . . . 808
BatchID . . . 808
BatchStatus . . . 808
12.11Attribute Examples . . . 809
C Attribute Examples . . . 810
C++ Attribute Examples . . . 811
C# Attribute Examples . . . 811
Java Attribute Examples . . . 812
Python Attribute Examples . . . 813
Visual Basic Attribute Examples . . . 813

13 Parameters **815**
13.1 Parameter Guidelines . . . 823
Continuous Models . . . 823
MIP Models . . . 824
13.2 Parameter Descriptions . . . 827
AggFill . . . 827
Aggregate . . . 828
BarConvTol . . . 828
BarCorrectors . . . 829
BarHomogeneous . . . 829
BarIterLimit . . . 829
BarOrder . . . 830
BarQCPConvTol . . . 830
BestBdStop . . . 831
BestObjStop . . . 831
BQPCuts . . . 831
BranchDir . . . 832
CliqueCuts . . . 832
CloudAccessID . . . 832
CloudHost . . . 833
CloudSecretKey . . . 833
CloudPool . . . 833
ComputeServer . . . 834
ConcurrentJobs . . . 834
ConcurrentMethod . . . 835
ConcurrentMIP . . . 835
ConcurrentSettings . . . 836
CoverCuts . . . 837
Crossover . . . 837
CrossoverBasis . . . 838
CSAPIAccessID . . . 838
CSAPISecret . . . 839
CSAppName . . . 839
CSAuthToken . . . 839
CSBatchMode . . . 839
CSClientLog . . . 840
CSGroup . . . 840
CSIdleTimeout . . . 841
CSManager . . . 841
CSPriority . . . 842
CSQueueTimeout . . . 842
CSRouter . . . 842
CSTLSInsecure . . . 843
CutAggPasses . . . 843
Cutoff . . . 844

CutPasses . . . 844
Cuts . . . 845
DegenMoves . . . 845
Disconnected . . . 845
DisplayInterval . . . 846
DistributedMIPJobs . . . 846
DualReductions . . . 847
FeasibilityTol . . . 847
FeasRelaxBigM . . . 847
FlowCoverCuts . . . 848
FlowPathCuts . . . 848
FuncPieceError . . . 848
FuncPieceLength . . . 849
FuncPieceRatio . . . 849
FuncPieces . . . 849
FuncMaxVal . . . 850
FuncNonlinear . . . 850
GomoryPasses . . . 851
GUBCoverCuts . . . 851
Heuristics . . . 851
IgnoreNames . . . 852
IISMethod . . . 852
ImpliedCuts . . . 853
ImproveStartGap . . . 853
ImproveStartNodes . . . 853
ImproveStartTime . . . 854
InfProofCuts . . . 854
InfUnbdInfo . . . 855
InputFile . . . 855
IntegralityFocus . . . 855
IntFeasTol . . . 856
IterationLimit . . . 857
JobID . . . 857
JSONSolDetail . . . 857
LazyConstraints . . . 857
LicenseID . . . 858
LiftProjectCuts . . . 858
LPWarmStart . . . 859
LogFile . . . 860
LogToConsole . . . 860
MarkowitzTol . . . 860
MemLimit . . . 861
Method . . . 861
MinRelNodes . . . 862
MIPFocus . . . 863

MIPGap . . . 863
MIPGapAbs . . . 864
MIPSepCuts . . . 864
MIQCPMethod . . . 865
MIRCuts . . . 865
MixingCuts . . . 865
ModKCuts . . . 866
MultiObjMethod . . . 866
MultiObjPre . . . 867
MultiObjSettings . . . 867
NetworkAlg . . . 867
NetworkCuts . . . 868
NLPHeur . . . 868
NodefileDir . . . 869
NodefileStart . . . 869
NodeLimit . . . 869
NodeMethod . . . 870
NonConvex . . . 870
NoRelHeurTime . . . 871
NoRelHeurWork . . . 871
NormAdjust . . . 871
NumericFocus . . . 872
OBBT . . . 872
ObjNumber . . . 872
ObjScale . . . 873
OptimalityTol . . . 873
OutputFlag . . . 874
PartitionPlace . . . 874
PerturbValue . . . 875
PoolGap . . . 875
PoolGapAbs . . . 875
PoolSearchMode . . . 876
PoolSolutions . . . 876
PreCrush . . . 877
PreDepRow . . . 877
PreDual . . . 878
PreMIQCPForm . . . 878
PrePasses . . . 879
PreQLinearize . . . 879
Presolve . . . 879
PreSOS1BigM . . . 880
PreSOS1Encoding . . . 880
PreSOS2BigM . . . 881
PreSOS2Encoding . . . 881
PreSparsify . . . 882

ProjImpliedCuts . . . 882
PSDCuts . . . 883
PSDTol . . . 883
PumpPasses . . . 884
QCPDual . . . 884
Quad . . . 884
Record . . . 885
ResultFile . . . 885
RINS . . . 886
RelaxLiftCuts . . . 886
RLTCuts . . . 886
ScaleFlag . . . 887
ScenarioNumber . . . 887
Seed . . . 888
ServerPassword . . . 888
ServerTimeout . . . 888
Sifting . . . 889
SiftMethod . . . 889
SimplexPricing . . . 890
SoftMemLimit . . . 890
SolutionLimit . . . 890
SolutionTarget . . . 891
SolFiles . . . 891
SolutionNumber . . . 892
StartNodeLimit . . . 892
StartNumber . . . 893
StrongCGCuts . . . 893
SubMIPCuts . . . 893
SubMIPNodes . . . 894
Symmetry . . . 894
Threads . . . 895
TimeLimit . . . 895
TokenServer . . . 896
TSPort . . . 896
TuneBaseSettings . . . 896
TuneCleanup . . . 897
TuneCriterion . . . 897
TuneDynamicJobs . . . 897
TuneJobs . . . 898
TuneMetric . . . 898
TuneOutput . . . 899
TuneResults . . . 899
TuneTargetMIPGap . . . 900
TuneTargetTime . . . 900
TuneTimeLimit . . . 900

TuneTrials . . . 901
TuneUseFilename . . . 901
UpdateMode . . . 901
UserName . . . 902
VarBranch . . . 903
WLSAccessID . . . 903
WLSSecret . . . 903
WLSToken . . . 904
WLSTokenDuration . . . 904
WLSTokenRefresh . . . 904
WorkerPassword . . . 905
WorkerPool . . . 905
WorkLimit . . . 905
ZeroHalfCuts . . . 906
ZeroObjNodes . . . 906
13.3 Parameter Examples . . . 907
C Parameter Examples . . . 907
C++ Parameter Examples . . . 908
C# Parameter Examples . . . 908
Java Parameter Examples . . . 909
MATLAB Parameter Examples . . . 909
Python Parameter Examples . . . 910
R Parameter Examples . . . 910
Visual Basic Parameter Examples . . . 911

14 Optimization Status Codes **912**

15 Batch Status Codes **914**

16 Callback Codes **915**

17 Error Codes **920**

18 Model File Formats **923**
18.1 MPS format . . . 923
18.2 REW format . . . 932
18.3 DUA format . . . 932
18.4 LP format . . . 932
18.5 RLP format . . . 939
18.6 DLP format . . . 939
18.7 ILP format . . . 940
18.8 OPB format . . . 940
18.9 MST format . . . 940
18.10HNT format . . . 941
18.11ORD format . . . 941
18.12BAS format . . . 942

18.13SOL format . . . 943
18.14JSON solution format . . . 943
18.15ATTR format . . . 950
18.16PRM format . . . 951

19 Logging **952**
19.1 Header . . . 952
19.2 Simplex Logging . . . 953
19.3 Barrier Logging . . . 954
19.4 Sifting Logging . . . 957
19.5 MIP Logging . . . 958
19.6 Solution Pool and Multi-Scenario Logging . . . 961
19.7 Multi-Objective Logging . . . 962
19.8 Distributed MIP Logging . . . 963
19.9 IIS Logging . . . 964

20 Gurobi Command-Line Tool **966**
20.1 Solving a Model . . . 967
20.2 Replaying Recording Files . . . 969

21 Solution Pool **970**
21.1 Finding Multiple Solutions . . . 970
21.2 Retrieving Solutions . . . 971
21.3 Examples . . . 971
21.4 Subtleties and Limitations . . . 973

22 Multiple Objectives **975**
22.1 Specifying Multiple Objectives . . . 975
22.2 Working With Multiple Objectives . . . 976
22.3 Additional Details . . . 979

23 Multiple Scenarios **981**
23.1 Definition of a Multi-Scenario Model . . . 981
23.2 Specifying Multiple Scenarios . . . 981
23.3 Logging . . . 982
23.4 Retrieving Solutions for Multiple Scenarios . . . 983
23.5 Tips and Tricks . . . 983
23.6 Limitations and Additional Considerations . . . 984

24 Batch Optimization **985**
24.1 Setting Up a Batch Environment . . . 985
24.2 Tagging Variables or Constraints . . . 985
24.3 Submitting a Batch Optimization Request . . . 986
24.4 Interacting with Batch Requests . . . 986
24.5 Interpreting the JSON Solution . . . 987
24.6 A Complete Example . . . 988

24.7 Limitations . . . 990

25 Recording API Calls **991**
25.1 Recording . . . 992
25.2 Replay . . . 992
25.3 Limitations . . . 993

26 Concurrent Optimizer **994**

27 Parameter Tuning Tool **997**
27.1 Command-Line Tuning . . . 998
27.2 Tuning API . . . 1001

28 Gurobi Instant Cloud **1002**
28.1 Client Setup . . . 1002
28.2 Instant Cloud Setup . . . 1003

29 Guidelines for Numerical Issues **1005**
29.1 Avoid rounding of input . . . 1005
29.2 Real numbers are not real . . . 1006
29.3 Tolerances and user-scaling . . . 1007
Models at the edge of infeasibility . . . 1008
Gurobi tolerances and the limitations of double-precision arithmetic . . . 1009
Why scaling and geometry is relevant . . . 1009
Recommended ranges for variables and constraints . . . 1012
Improving ranges for variables and constraints . . . 1012
Advanced user scaling . . . 1013
Avoid hiding large coefficients . . . 1013
Dealing with big-M constraints . . . 1014
29.4 Does my model have numerical issues? . . . 1015
29.5 Solver parameters to manage numerical issues . . . 1016
Presolve . . . 1016
Choosing the right algorithm . . . 1017
Making the algorithm less sensitive . . . 1018
29.6 Instability and the geometry of optimization problems . . . 1019
The case of linear systems: . . . 1019
The geometry of linear optimization problems . . . 1020
Multiple optimal solutions . . . 1021
Dealing with epsilon-optimal solutions . . . 1022
Thin feasible regions . . . 1024
Optimizing over the circle: . . . 1025
Optimizing over thin regions: . . . 1027
Stability and convergence . . . 1028
29.7 Further reading . . . 1028
Source code for the experiment of optimizing over a circle . . . 1029
Source code for the experiment on a thin feasible region . . . 1030

Source code for the experiment with column scalings 1030

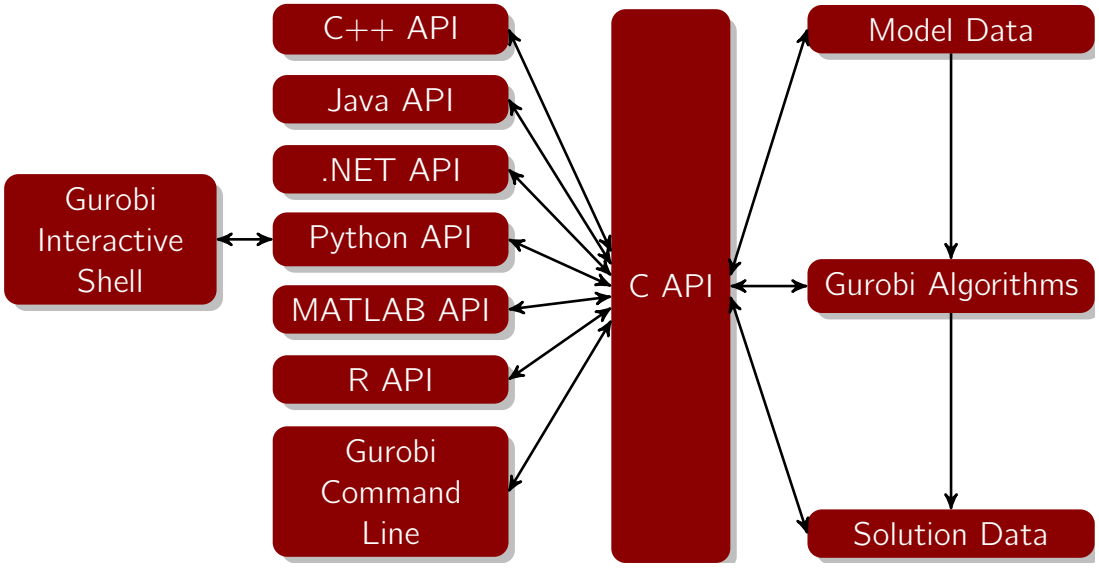
30 File Overview – Linux platform **1032**

31 File Overview – macOS platform **1035**

32 File Overview – Windows platform **1038**

33 Copyright Notices for 3rd Party Libraries **1041**

Introduction



This is the reference manual for the Gurobi™ Optimizer. It contains documentation for the following Gurobi language interfaces:

- C
- C++
- Java®
- Microsoft®.NET
- Python®
- MATLAB®
- R

The *Gurobi interactive shell* is also documented in the Python section.

The different Gurobi language interfaces share many common features. These are described at the end of this manual. Two particularly important common features are the Attribute interface and the Gurobi Parameter set. You may wish to bookmark these pages, since you are likely to refer to them frequently as you develop applications that use the Gurobi Optimizer.

Additional Topics

This document covers a number of additional topics, which are listed here:

- Detailed Release Notes
- Variables and Constraints and Objectives
- Environments
- Attributes
- Parameters
- Optimization Status Codes
- Callback Codes
- Error Codes
- File Formats
- Logging
- Command-Line Tool
- Solution Pool
- Multiple Objectives
- Multiple Scenarios
- Batch Optimization
- Recording API Calls
- Concurrent Optimizer
- Parameter Tuning Tool
- Instant Cloud
- Guidelines for Numerical Issues

Additional Resources

You can consult the Getting Started Knowledge Base article for a high-level overview of the Gurobi Optimizer, or the Gurobi Example Tour for a quick tour of the examples provided with the Gurobi distribution, or the Gurobi Remote Services Reference Manual for an overview of Gurobi Compute Server, Distributed Algorithms, and Gurobi Remote Services.

Getting Help

If you have a question that is not answered in this document, please visit the Gurobi support site at https://support.gurobi.com. There, you can read knowledge base articles and join the community discussion forum. Also, if you have a current maintenance contract, you can use the Gurobi support site to submit a request to the Gurobi support team.

Document version

Documentation revision: `6c1f3f6`
Code revision: `736e3da0`

Detailed Release Notes for Gurobi 11.0.0

Supported Platforms

Platform (port)	Operating System	Compiler	Notes
Windows 64-bit (win64)	Windows 10, 11, Windows Server 2016, 2019, 2022	Visual Studio 2017-2022	Use `gurobi_c++md2017.lib` (e.g.) for C++
Linux x86-64 64-bit (linux64)	Red Hat Enterprise Linux 8, 9 SUSE Enterprise Linux 15 Ubuntu 20.04, 22.04 Amazon Linux 2	GCC >= 8.5	
macOS 64-bit universal2 (macos_universal2)	12 (Monterey), 13 (Ventura), 14 (Sonoma)	Xcode 13/14	
Linux arm64 64-bit (armlinux64)	Red Hat Enterprise Linux 8, 9 SUSE Enterprise Linux 15 Ubuntu 20.04, 22.04 Amazon Linux 2	GCC >= 8.5	
AIX 64-bit (power64)	AIX 7.2, 7.3	XL C/C++ 9	Note that Gurobi 11 will be the last supported version for AIX. Due to limited Python support on AIX, this port does not include the Interactive Shell or the Python libraries.

Additional Supported Platform Information

Gurobi 11.0.0 supports the following language/platform versions:

Language	Version
Python	3.8, 3.9, 3.10, 3.11, 3.12
MATLAB	R2019a-R2023b
R	4.3
JDK	8, 11, 17
.NET	6.0

Bug Fixes

All bug fixes listed by version can be found on our website.

Changed in Gurobi 11.0

Linux GNU C library (glibc) dependency

- **Linux x86_64**: The Gurobi Optimizer now requires a minimum glibc version of 2.17. For version 10.0, the minimum required glibc version was 2.3.4.

  The list of supported platforms above takes these changes into account.

New features affecting all APIs

Mixed-Integer Non-Linear Programming (MINLP) Problems

The Gurobi Optimizer can now use *spatial branching* and *outer approximation* to solve models with non-linear functions, instead of using static piecewise-linear approximations. See the discussion of function constraints with dynamic piecewise-linear approximation for more information.

Default behavior of NonConvex parameter

The default behavior of the NonConvex parameter has changed. Previously, you had to explicitly set this parameter to value 2 if you wanted to solve a model with a non-convex quadratic objective or constraint. Now it will solve such models by default. To revert to the previous behavior, set the parameter to 1.

Copying models between environments

It is now possible to copy a model from one environment to another, using the following new routines or signatures:

- C: `GRBcopymodeltoenv`
- C++: `GRBModel(const GRBModel& model, const GRBenv& targetenv)`
- Java: `GRBModel(GRBModel model, GRBenv targetenv)`
- .NET: `GRBModel(GRBModel model, GRBenv targetenv)`
- Python: `model.copy(env)`

The primary use case for copying models is when performing optimization on multiple, related versions of the same problem on multiple threads. Multi-threading within an environment is not supported.

For Compute Server users, note that you can copy a model from a client to a Compute Server environment, but it is not possible to copy models between two Compute Server environments.

Better control over the concurrent LP optimizer

Option 5 of the Method parameter has been deprecated in favor of the new ConcurrentMethod parameter. The latter allows you to control which methods are run concurrently with the deterministic (`Method=4`) or non-deterministic (`Method=3`) concurrent optimizer. The deprecated setting `Method=5` can now be replicated by choosing `Method=4` along with `ConcurrentMethod=3`. To give another example, non-deterministic concurrent execution of the primal and dual simplex algorithms can be selected using `Method=3` along with `ConcurrentMethod=3`.

Change the Threads parameter when resuming optimization

If an optimization is interrupted (e.g., due to a time limit), the user can now modify the Threads parameter in order to change the number of threads to be used when the solving process is resumed with another call to optimize.

Tuning

The distributed tuning can use remote workers in a dynamic way. With the parameter TuneDynamicJobs you can control the size of a dynamic set of remote workers that are used in parallel. These workers are used for a limited amount of time and afterwards potentially returned so that they are available for other remote jobs.

Starting from Gurobi 11, we include in the results the first set of parameters used to solve the model (base settings). This result is stored as first setting in the tune results. Hence, to get the best tuned set of parameters, you need to request at least 2 sets (using TuneResults) and retrieve the set with index 1.

The tuner now supports the log callback. You can add a log callback to the environment and retrieve all the log messages coming from the tuner.

Piecewise-linear approximation

The default behavior of parameter FuncPieces, the main parameter for piecewise-linear approximation of function constraints, is changed. The new default for version 11.0 uses the relative error approach for the approximation, while for version 10.0 or earlier it mainly used the number of function constraints to set the total number of pieces.

New parameters

The following parameters are new in Gurobi 11.0:

- SolutionTarget to specify the solution target for LP.
- FuncNonlinear to control whether general function constraints shall be treated as nonlinear functions or via piecewise-linear approximation.
- MixingCut to control the generation of Mixing Cuts.
- TuneDynamicJobs to enable distributed tuning using a dynamic set of workers.
- TuneUseFilename to use the model file names as model names during tuning (Command-line only).

Changes to gurobipy

Installation and packaging changes

- `gurobipy` and its `setup.py` file are no longer distributed with the Gurobi installers. `gurobipy` must now be installed into Python environments using pip or conda.
- The `gurobi` conda packages have been updated so that pip can now recognize when `gurobipy` has already been installed into an environment by conda. As a result, using pip in a conda environment to install a package that depends on `gurobipy` (such as gurobi-machinelearning or gurobipy-pandas) no longer leads to a duplicate installation of `gurobipy`.

- Type hints for `gurobipy` are now included in both the pip wheels and conda packages. For Gurobi version 11, users should *not* install the gurobipy-stubs package as it will override the type annotations packaged with `gurobipy`. The `gurobipy-stubs` package will not be maintained for future versions of `gurobipy`.

New classes, methods, and properties

- SOS objects have a new `index` property returning the index of the SOS constraint in the model.
- MConstr and MQConstr objects now implement the class method `fromlist` which has the same behavior as the corresponding method on the MVar class.
- New methods hstack, vstack, and concatenate allow matrix-friendly objects to be joined together in the same way as the corresponding numpy functions.
- A new MGenConstr class has been added for holding an array of general constraints. For version 11, only the addGenConstrIndicator method is able to return objects of this type.
- Env objects now have an additional method getParam which returns the current value of the given parameter in the environment.

Enhancements to existing methods

- The Model.optimize and Model.computeIIS methods now accept any Python callable as a callback function. The TSP example code has been updated to demonstrate the use of a callable class.
- The matrix-friendly API is now fully integrated with callback methods: Model.cbLazy, Model.cbCut, and Model.cbSetSolution now accept matrix-friendly API objects.
- When constructing a set of linear constraints using Model.addMConstr or Model.addConstr with the matrix-friendly API, user-defined constraint names can be passed as a list or ndarray.
- Model.addGenConstrIndicator and the overloaded `>>` syntax for indicator constraints now handle matrix-friendly linear expression objects. This allows indicator constraints to be built in a vectorized manner (including broadcasting behavior).
- Users can now explicitly disable the default environment by setting the environment variable `GUROBIPY_ALLOW_DEFAULTENV` to `0` before starting a Python program. With this setting, environments must be managed explicitly using Env objects in user code. Calls to Model() without an Env object and global function calls such as setParam will raise a `GurobiError` if this setting is used.

Changes that may require updating your code

- Model.getAttr and Model.setAttr now raise an exception if the modeling objects passed to them do not belong to the model. Previously, the attribute values would be returned or set for a variable on the model with the same index as the modeling object which is typically not the desired behavior.

- A call to Model.addGenConstrIndicator using size-1 matrix-friendly modeling objects, which previously returned a GenConstr object, will now return a size-1 MGenConstr. The constraint name may also change depending on the shape of the result. To revert to the previous behaviour, call `.item()` on matrix-friendly modeling objects passed to this function.
- The tupledict now follows Python 3 dictionary semantics: its `keys`, `values`, and `items` methods now return iterable views instead of lists. Importantly, `keys` no longer returns a tuplelist. If you need this behavior, you can instead call `gp.tuplelist(td.keys())`.
- Env.setParam now raises a `GurobiError` if the user specifies a parameter which does not exist. In previous versions, such calls would print a message but would fail without an exception.
- Gurobi log messages now go to the 'gurobipy' logger instead of 'gurobipy.gurobipy'. If you use Python's logging library to collect Gurobi log messages, handlers should be attached either to the root logger or to `logging.getLogger("gurobipy")`.
- Calling the `sum()` method of an MVar, MLinExpr, or MQuadExpr which contains no elements (i.e. has at least one zero-length dimension) now returns an array of zeros of the appropriate shape instead of raising an exception. This matches the behavior of numpy arrays.
- Passing too many keys (more than the length of the contained tuples) to any of the pattern matching methods (tuplelist.select, tupledict.select, tupledict.sum, and tupledict.prod) now raises an `IndexError`. This change almost exclusively affects cases which returned an empty set in previous versions. The exception is where the additional indices are wildcards. i.e. for a tuplelist containing 2-tuples: `tl.select("*", 2, "*")` now raises an exception, while in previous versions it gives the same (non-empty) result as `tl.select("*", 2)`.
- The `gurobi` Python package alias has been removed from the conda packages on all platforms. Any usage of `import gurobi` must be replaced with `import gurobipy`.

Deprecated functionality

If you are upgrading from a previous version of Gurobi, we recommend first running your code with Gurobi 10 and warnings enabled to catch deprecations in `gurobipy`. Fixing these deprecated usages will help to keep compatibility for Gurobi 11 and later versions. Warnings can be enabled by running your code with the `-X dev` or `-W default` flags. See the Python Development Mode or warnings package documentation for further details.

In Gurobi 11, the following usage is deprecated and will be removed in a future version:

- Passing a tupledict to the quicksum function is deprecated as the result is inconsistent with passing a native Python dictionary to `quicksum`. The sum method of the tupledict should be used instead to compute the sum over the values of the tupledict.
- The `iterkeys()`, `itervalues()`, and `iteritems()` methods for tupledicts are deprecated. These methods are legacy syntax from Python 2. The corresponding Python 3 methods `keys()`, `values()`, and `items()` now return iterable views and are therefore preferred.
- The `gurobipy.system()` function is deprecated. `os.system()` from the Python standard library should be used instead.
- The `gurobipy.models()` function is deprecated and should no longer be used.

Changes to the Java package

Updates

We've made essential changes in the package structure to allow publishing to Maven Central. Please take note of the following updates:

- **Package Name Change**: The Java package name has been updated from `gurobi` to `com.gurobi.gurobi`. If you are integrating our Java package, ensure to update your import statements accordingly.

Changes to the MATLAB interface

On the macOS platform, the Gurobi mex interface is now also available for native Apple silicon MATLAB (mexmaca64).

Compute Server, Cluster Manager, and Instant Cloud

Detailed release notes for Compute Server, Cluster Manager, and Instant Cloud can be found here

This section documents the Gurobi C interface. This manual begins with a quick overview of the functions in the interface, and continues with detailed descriptions of all of the available interface routines.

If you are new to the Gurobi Optimizer, we suggest that you start with the Getting Started Knowledge Base article or the Example Tour. These documents provide concrete examples of how to use the routines described here.

Environments

The first step in using the Gurobi C optimizer is to create an environment, using the GRBloadenv call. The environment acts as a container for all data associated with a set of optimization runs. You will generally only need one environment in your program, even if you wish to work with multiple optimization models. Once you are done with an environment, you should call GRBfreeenv to release the associated resources.

For more advanced use cases, you can use the GRBemptyenv routine to create an uninitialized environment and then, programmatically, set all required options for your specific requirements. For further details see the Environment section.

Models

You can create one or more optimization models within an environment. A model consists of a set of variables, a linear, quadratic, or piecewise-linear objective function on those variables, and a set of constraints. Each variable has an associated lower bound, upper bound, type (continuous, binary, integer, semi-continuous, or semi-integer), and linear objective coefficient. Each linear constraint has an associated sense (less-than-or-equal, greater-than-or-equal, or equal), and right-hand side value. Refer to this section for more information on variables and constraints.

An optimization model may be specified all at once, through the GRBloadmodel routine, or built incrementally, by first calling GRBnewmodel and then calling GRBaddvars to add variables and GRBaddconstr, GRBaddqconstr, GRBaddsos, or any of the GRBaddgenconstrXxx methods to add constraints. Models are dynamic entities; you can always add or delete variables or constraints.

Specific variables and constraints are referred to throughout the Gurobi C interface using their indices. Variable indices are assigned as variables are added to the model, in a contiguous fashion. The same is true for constraints. In adherence to C language conventions, indices all start at 0.

We often refer to the *class* of an optimization model. At the highest level, a model can be continuous or discrete, depending on whether the modeling elements present in the model require discrete decisions to be made. Among continuous models...

- A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*.
- If the objective is quadratic, the model is a *Quadratic Program (QP)*.
- If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We sometimes refer to a few special cases of QCP: QCPs with convex constraints,

QCPs with non-convex constraints, *bilinear programs*, and *Second-Order Cone Programs (SOCP)*.

- If any of the constraints are non-linear (chosen from among the available general constraints), the model is a *Non-Linear Program (NLP)*.

A model that contains any integer variables, semi-continuous variables, semi-integer variables, Special Ordered Set (SOS) constraints, or general constraints, is discrete, and is referred to as a *Mixed Integer Program (MIP)*. The special cases of MIP, which are the discrete versions of the continuous models types we've already described, are...

- *Mixed Integer Linear Programs (MILP)*
- *Mixed Integer Quadratic Programs (MIQP)*
- *Mixed Integer Quadratically-Constrained Programs (MIQCP)*
- *Mixed Integer Second-Order Cone Programs (MISOCP)*
- *Mixed Integer Non-Linear Programs (MINLP)*

The Gurobi Optimizer handles all of these model classes. Note that the boundaries between them aren't as clear as one might like, because we are often able to transform a model from one class to a simpler class.

Solving a Model

Once you have built a model, you can call GRBoptimize to compute a solution. By default, `GRBoptimize()` will use the concurrent optimizer to solve LP models, the barrier algorithm to solve QP models with convex objectives and QCP models with convex constraints, and the branch-and-cut algorithm otherwise. The solution is stored as a set of *attributes* of the model. The C interface contains an extensive set of routines for querying these attributes.

The Gurobi algorithms keep careful track of the state of the model, so calls to `GRBoptimize()` will only perform further optimization if relevant data has changed since the model was last optimized. If you would like to discard previously computed solution information and restart the optimization from scratch without changing the model, you can call GRBreset.

After a MIP model has been solved, you can call GRBfixmodel to compute the associated *fixed* model. This model is identical to the original, except that the integer variables are fixed to their values in the MIP solution. If your model contains SOS constraints, some continuous variables that appear in these constraints may be fixed as well. In some applications, it can be useful to compute information on this fixed model (e.g., dual variables, sensitivity information, etc.), although you should be careful in how you interpret this information.

Multiple Solutions, Objectives, and Scenarios

By default, the Gurobi Optimizer assumes that your goal is to find one proven optimal solution to a single model with a single objective function. Gurobi provides the following features that allow you to relax these assumptions:

- Solution Pool: Allows you to find more solutions.
- Multiple Scenarios: Allows you to find solutions to multiple, related models.

- Multiple Objectives: Allows you to specify multiple objective functions and control the trade-off between them.

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call GRBcomputeIIS to compute an Irreducible Inconsistent Subsystem (IIS). This routine can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. This routine populates a set of IIS attributes.

To attempt to repair an infeasibility, call GRBfeasrelax to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Querying and Modifying Attributes

Most of the information associated with a Gurobi model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. To give a simple example, solving an optimization model causes the `X` variable attribute to be populated. Attributes such as `X` that are computed by the Gurobi Optimizer cannot be modified directly by the user, while others, such as the variable lower bound array (the `LB` attribute) can.

The Gurobi C interface contains an extensive set of routines for querying or modifying attribute values. The exact routine to use for a particular attribute depends on the type of the attribute. As mentioned earlier, attributes can be either variable attributes, constraint attributes, or model attributes. Variable and constraint attributes are arrays, and use a set of array attribute routines. Model attributes are scalars, and use a set of scalar routines. Attribute values can additionally be of type *char*, *int*, *double*, or *string* (really *char **).

Scalar model attributes are accessed through a set of `GRBget*attr()` routines (e.g., GRBgetintattr). In addition, those model attributes that can be set directly by the user (e.g., the objective sense) may be modified through the `GRBset*attr()` routines (e.g., GRBsetdblattr).

Array attributes are accessed through three sets of routines. The first set, the `GRBget*attrarray()` routines (e.g., GRBgetcharattrarray) return a contiguous sub-array of the attribute array, specified using the index of the first member and the length of the desired sub-array. The second set, the `GRBget*attrelement()` routines (e.g., GRBgetcharattrelement) return a single entry from the attribute array. Finally, the `GRBget*attrlist()` routines (e.g., GRBgetdblattrlist) retrieve attribute values for a list of indices.

Array attributes that can be set by the user are modified through the `GRBset*attrarray()`, `GRBset*attrelement()`, and `GRBset*attrlist()` routines.

The full list of Gurobi attributes can be found in the Attributes section.

Additional Model Modification Information

Most modifications to an existing model are done through the attribute interface (e.g., changes to variable bounds, constraint right-hand sides, etc.). The main exceptions are modifications to the constraints themselves, and to the quadratic and piecewise-linear portions of the objective function.

The constraint matrix can be modified in a few ways. The first is to call GRBchgcoeffs to change individual matrix coefficients. This routine can be used to modify the value of an existing

non-zero, to set an existing non-zero to zero, or to create a new non-zero. The constraint matrix is also modified when you remove constraints (through GRBdelconstrs) or variables (through GRBdelvars). The non-zero values associated with the deleted constraints or variables are removed along with the constraints or variables themselves.

Quadratic objective terms are added to the objective function using the GRBaddqpterms routine. You can add a list of quadratic terms in one call, or you can add terms incrementally through multiple calls. The GRBdelq routine allows you to delete all quadratic terms from the model. Note that quadratic models will typically have both quadratic and linear terms. Linear terms are entered and modified through the `Obj` attribute, in the same way that they are handled for models with purely linear objective functions.

If your variables have piecewise-linear objectives, you can specify them using the GRBsetpwlobj routine. Call this routine once for each relevant variable. The Gurobi simplex solver includes algorithmic support for convex piecewise-linear objective functions, so for continuous models you should see a substantial performance benefit from using this feature. To clear a previously specified piecewise-linear objective function, simply set the `Obj` attribute on the corresponding variable to 0.

Lazy Updates

One important item to note about model modification in the Gurobi optimizer is that it is performed in a *lazy* fashion, meaning that modifications don't affect the model immediately. Rather, they are queued and applied later. If your program simply creates a model and solves it, you will probably never notice this behavior. However, if you ask for information about the model before your modifications have been applied, the details of the lazy update approach may be relevant to you.

As we just noted, model modifications (bound changes, right-hand side changes, objective changes, etc.) are placed in a queue. These queued modifications can be applied to the model in three different ways. The first is by an explicit call to GRBupdatemodel. The second is by a call to GRBoptimize. The third is by a call to GRBwrite to write out the model. The first case gives you fine-grained control over when modifications are applied. The second and third make the assumption that you want all pending modifications to be applied before you optimize your model or write it to disk.

Why does the Gurobi interface behave in this manner? There are a few reasons. The first is that this approach makes it much easier to perform multiple modifications to a model, since the model remains unchanged between modifications. The second is that processing model modifications can be expensive, particularly in a Compute Server environment, where modifications require communication between machines. Thus, it is useful to have visibility into exactly when these modifications are applied. In general, if your program needs to make multiple modifications to the model, you should aim to make them in phases, where you make a set of modifications, then update, then make more modifications, then update again, etc. Updating after each individual modification can be extremely expensive.

If you forget to call update, your program won't crash. Your query will simply return the value of the requested data from the point of the last update. If the object you tried to query didn't exist then, you'll get an `INDEX_OUT_OF_RANGE` error instead.

The semantics of lazy updates have changed since earlier Gurobi versions. While the vast majority of programs are unaffected by this change, you can use the UpdateMode parameter to revert to the earlier behavior if you run into an issue.

Managing Parameters

The Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization. Parameters are set using the `GRBset*param()` routines (e.g., GRBsetintparam). Current values can be retrieved with the `GRBget*param()` routines (e.g., GRBgetdblparam). Parameters can be of type *int*, *double*, or *char ** (string). You can also read a set of parameter settings from a file using GRBreadparams, or write the set of changed parameters using GRBwriteparams.

We also include an automated parameter tuning tool that explores many different sets of parameter changes in order to find a set that improves performance. You can call GRBtunemodel to invoke the tuning tool on a model. Refer to the parameter tuning tool section for more information.

One thing we should note is that each model gets its own copy of the environment when it is created. Parameter changes to the original environment therefore have no effect on existing models. Use GRBgetenv to retrieve the environment associated with a particular model if you want to change a parameter for that model.

Monitoring Progress - Logging and Callbacks

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging behavior. If you would like to direct output to a file as well as to the screen, specify the log file name in GRBloadenv when you create your environment. You can modify the LogFile parameter if you wish to redirect the log to a different file after creating the environment. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter. A detailed description of the Gurobi log file can be found in the Logging section.

More detailed progress monitoring can be done through the Gurobi callback function. The GRBsetcallbackfunc routine allows you to install a function that the Gurobi Optimizer will call regularly during the optimization process. You can call GRBcbget from within the callback to obtain additional information about the state of the optimization.

Modifying Solver Behavior - Callbacks

Callbacks can also be used to modify the behavior of the Gurobi optimizer. If you call routine GRBterminate from within a callback, for example, the optimizer will terminate at the earliest convenient point. Routine GRBcbsolution allows you to inject a feasible solution (or partial solution) during the solution of a MIP model. Routines GRBcbcut and GRBcblazy allow you to add *cutting planes* and *lazy constraints* during a MIP optimization, respectively. Routine GRBcbstoponemultiobj allows you to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

Batch Optimization

Gurobi Compute Server enables programs to offload optimization computations onto dedicated servers. The Gurobi Cluster Manager adds a number of additional capabilities on top of this. One important one, *batch optimization*, allows you to build an optimization model with your client program, submit it to a Compute Server cluster (through the Cluster Manager), and later check

on the status of the model and retrieve its solution. You can use a Batch object to make it easier to work with batches. For details on batches, please refer to the Batch Optimization section.

Error Handling

Most of the Gurobi C library routines return an integer error code. A zero return value indicates that the routine completed successfully, while a non-zero value indicates that an error occurred. The list of possible error return codes can be found in the Error Codes section.

When an error occurs, additional information on the error can be obtained by calling GRBgeterrormsg.

3.1 Environment Creation and Destruction

GRBloadenv

```
int  GRBloadenv ( GRBenv      **envP,
                  const char  *logfilename )
```

Create an environment. Optimization models live within an environment, so this is typically the first Gurobi routine called in an application.

This routine will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This routine will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Return value:

A non-zero return value indicates that there was a problem creating the environment. Refer to the Error Code table for a list of possible return values.

Arguments:

`envP`: The location in which the pointer to the newly created environment should be placed.

`logfilename`: The name of the log file for this environment. May be `NULL` (or an empty string), in which case no log file is created.

GRBemptyenv

```
int  GRBemptyenv ( GRBenv  **envP )
```

Create an empty environment. Note that you will need to call GRBstartenv before you can use this environment.

You should use this routine if you want to set parameters before actually starting the environment. This can be useful if you want to connect to a Compute Server, a Token Server, the Gurobi Instant Cloud, a Cluster Manager or use a WLS license. See the Environment Section for more details.

Return value:

A non-zero return value indicates that there was a problem creating the environment. Refer to the Error Code table for a list of possible return values.

Arguments:

`envP`: The location in which the pointer to the newly created environment should be placed.

GRBstartenv

```
int  GRBstartenv ( GRBenv  *env )
```

Start an empty environment. This routine starts an empty environment created by GRBemptyenv. If the environment has already been started, this routine will do nothing. If the routine fails, the environment will have the same state as it had before the call to this function.

This routine will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This routine will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter). After that, it will apply all parameter changes specified by the user prior to this call. Note that this might overwrite parameters set in the license file, or in the `gurobi.env` file, if present.

After all these changes are performed, the code will actually activate the environment, and make it ready to work with models.

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Return value:

A non-zero return value indicates that there was a problem starting the environment. Refer to the Error Code table for a list of possible return values.

Arguments:

`env`: The empty environment to start.

GRBfreeenv

```
void  GRBfreeenv ( GRBenv *env )
```

Free an environment that was previously allocated by GRBloadenv, and release the associated memory. This routine should be called when an environment is no longer needed. In particular, it should only be called once all models built using the environment have been freed.

Arguments:

`env`: The environment to be freed.

GRBgetconcurrentenv

```
GRBenv *  GRBgetconcurrentenv ( GRBmodel *model,
                                int      num )
```

Create/retrieve a concurrent environment for a model.

This routine provides fine-grained control over the concurrent optimizer. By creating your own concurrent environments and setting appropriate parameters on these environments (e.g., the Method parameter), you can control exactly which strategies the concurrent optimizer employs. For example, if you create two concurrent environments, and set `Method` to primal simplex for one and dual simplex for the other, subsequent concurrent optimizer runs will use the two simplex algorithms rather than the default choices.

Note that you must create contiguously numbered concurrent environments, starting with `num=0`. For example, if you want three concurrent environments, they must be numbered 0, 1, and 2.

Once you create concurrent environments, they will be used for every subsequent concurrent optimization on that model. Use GRBdiscardconcurrentenvs to revert back to default concurrent optimizer behavior.

Return value:

The concurrent environment. A `NULL` return value indicates that there was a problem creating the environment.

Arguments:

`model`: The model for the concurrent environment.

`num`: The concurrent environment number.

Example usage:

```
GRBenv *env0 = GRBgetconcurrentenv(model, 0);
GRBenv *env1 = GRBgetconcurrentenv(model, 1);
```

GRBgetmultiobjenv

```
GRBenv*  GRBgetmultiobjenv (  GRBmodel  *model,
                              int       num )
```

Create/retrieve a multi-objective environment for the optimization pass with the given index. This environment enables fine-grained control over the multi-objective optimization process. Specifically, by changing parameters on this environment, you modify the behavior of the optimization that occurs during the corresponding pass of the multi-objective optimization.

Each multi-objective environment starts with a copy of the current model environment.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Please refer to the discussion on Combining Blended and Hierarchical Objectives for information on the optimization passes to solve multi-objective models.

Return value:

The environment associated with a given optimization pass when solving the multiobjective model. A `NULL` return value indicates that there was a problem retrieving the environment.

Arguments:

`model`: The model from where we want to retrieve the multiobjective environment.

`num`: The optimization pass number, starting from 0.

Example usage:

```
GRBenv *env0 = GRBgetmultiobjenv(model,0);
GRBenv *env1 = GRBgetmultiobjenv(model,1);

GRBsetintparam(env0, "Method", 2);
GRBsetintparam(env1, "Method", 1);

GRBoptimize(model);

GRBdiscardmultiobjenvs(model);
```

GRBdiscardconcurrentenvs

```
void  GRBdiscardconcurrentenvs (  GRBmodel *  model )
```

Discard concurrent environments for a model.

The concurrent environments created by GRBgetconcurrentenv will be used by every subsequent call to the concurrent optimizer until the concurrent environments are discarded.

Arguments:

`model`: The model for the concurrent environment.

Example usage:

```
GRBdiscardconcurrentenvs(model);
```

GRBdiscardmultiobjenvs

```
void  GRBdiscardmultiobjenvs (  GRBmodel  *model )
```

Discard all multi-objective environments associated with the model, thus restoring multi objective optimization to its default behavior.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Arguments:

`model`: The model in which all multi objective environments will be discarded.

Example usage:

```
GRBenv *env0 = GRBgetmultiobjenv(model,0);
GRBenv *env1 = GRBgetmultiobjenv(model,1);

GRBsetintparam(env0, "Method", 2);
GRBsetintparam(env1, "Method", 1);

GRBoptimize(model);

GRBdiscardmultiobjenvs(model);
```

3.2 Model Creation and Modification

GRBloadmodel

```
int  GRBloadmodel (  GRBenv      *env,
                     GRBmodel    **modelP,
                     const char  *Pname,
                     int         numvars,
                     int         numconstrs,
                     int         objsense,
                     double      objcon,
                     double      *obj,
                     char        *sense,
                     double      *rhs,
                     int         *vbeg,
                     int         *vlen,
                     int         *vind,
                     double      *vval,
                     double      *lb,
                     double      *ub,
                     char        *vtype,
                     const char  **varnames,
                     const char  **constrnames )
```

Create a new optimization model, using the provided arguments to initialize the model data (objective function, variable bounds, constraint matrix, etc.). The model is then ready for optimization, or for modification (e.g., addition of variables or constraints, changes to variable types or bounds, etc.).

If your constraint matrix may contain more than 2 billion non-zero values, you should consider using the GRBXloadmodel variant of this routine.

Return value:

A non-zero return value indicates that a problem occurred while creating the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment in which the new model should be created. Note that the new model gets a copy of this environment, so subsequent modifications to the original environment (e.g., parameter changes) won't affect the new model. Use GRBgetenv to modify the environment associated with a model.

modelP: The location in which the pointer to the newly created model should be placed.

Pname: The name of the model.

numvars: The number of variables in the model.

numconstrs: The number of constraints in the model.

objsense: The sense of the objective function. Allowed values are 1 (minimization) or -1 (maximization).

objcon: Constant objective offset.

obj: Objective coefficients for the new variables. This argument can be NULL, in which case the objective coefficients are set to 0.0.

sense: The senses of the new constraints. Options are '=' (equal), '<' (less-than-or-equal), or '>' (greater-than-or-equal). You can also use constants GRB_EQUAL, GRB_LESS_EQUAL, or GRB_GREATER_EQUAL.

rhs: Right-hand side values for the new constraints. This argument can be NULL if you are not adding any constraint.

vbeg: Constraint matrix non-zero values are passed into this routine in Compressed Sparse Column (CSC) format. Each column in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the constraint index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each variable in the model has a vbeg and vlen value, indicating the start position of the non-zeros for that variable in the vind and vval arrays, and the number of non-zero values for that variable, respectively. Thus, for example, if vbeg[2] = 10 and vlen[2] = 2, that would indicate that variable 2 has two non-zero values associated with it. Their constraint indices can be found in vind[10] and vind[11], and the numerical values for those non-zeros can be found in vval[10] and vval[11]. Note that the columns of the matrix must be ordered from first to last, implying that the values in vbeg must be non-decreasing.

vlen: Number of constraint matrix non-zero values associated with each variable. See the description of the vbeg argument for more information.

vind: Constraint indices associated with non-zero values. See the description of the vbeg argument for more information.

vval: Numerical values associated with constraint matrix non-zeros. See the description of the vbeg argument for more information.

lb: Lower bounds for the new variables. This argument can be NULL, in which case all variables get lower bounds of 0.0.

ub: Upper bounds for the new variables. This argument can be NULL, in which case all variables get infinite upper bounds.

vtype: Types for the variables. Options are GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT. This argument can be NULL, in which case all variables are assumed to be continuous.

varnames: Names for the new variables. This argument can be NULL, in which case all variables are given default names.

constrnames: Names for the new constraints. This argument can be NULL, in which case all constraints are given default names.

Important notes:

We recommend that you build a model one constraint or one variable at a time, using GRBaddconstr or GRBaddvar, rather than using this routine to load the entire constraint matrix at once. It is much simpler, less error prone, and it introduces no significant overhead.

Example usage:

```
/* maximize    x +   y + 2 z
   subject to  x + 2 y + 3 z <= 4
               x +   y       >= 1
   x, y, z binary */
```

```
int    vars    = 3;
int    constrs = 2;
int    vbeg[]  = {0, 2, 4};
int    vlen[]  = {2, 2, 1};
int    vind[]  = {0, 1, 0, 1, 0};
double vval[]  = {1.0, 1.0, 2.0, 1.0, 3.0};
double obj[]   = {1.0, 1.0, 2.0};
char   sense[] = {GRB_LESS_EQUAL, GRB_GREATER_EQUAL};
double rhs[]   = {4.0, 1.0};
char   vtype[] = {GRB_BINARY, GRB_BINARY, GRB_BINARY};

error = GRBloadmodel(env, &model, "example", vars, constrs, -1, 0.0,
                     obj, sense, rhs, vbeg, vlen, vind, vval,
                     NULL, NULL, vtype, NULL, NULL);
```

GRBnewmodel

```
int  GRBnewmodel (  GRBenv      *env,
                    GRBmodel    **modelP,
                    const char  *Pname,
                    int         numvars,
                    double      *obj,
                    double      *lb,
                    double      *ub,
                    char        *vtype,
                    const char  **varnames )
```

Create a new optimization model. This routine allows you to specify an initial set of variables (with objective coefficients, bounds, types, and names), but the initial model will have no constraints. Constraints can be added later with GRBaddconstr or GRBaddconstrs.

Return value:

A non-zero return value indicates that a problem occurred while creating the new model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment in which the new model should be created. Note that the new model will get a copy of this environment, so subsequent modifications to the original environment (e.g., parameter changes) won't affect the new model. Use GRBgetenv to modify the environment associated with a model.

modelP: The location in which the pointer to the new model should be placed.

Pname: The name of the model.

numvars: The number of variables in the model.

obj: Objective coefficients for the new variables. This argument can be `NULL`, in which case the objective coefficients are set to `0.0`.

lb: Lower bounds for the new variables. This argument can be `NULL`, in which case all variables get lower bounds of `0.0`.

ub: Upper bounds for the new variables. This argument can be NULL, in which case all variables get infinite upper bounds.

vtype: Types for the variables. Options are GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT. This argument can be NULL, in which case all variables are assumed to be continuous.

varnames: Names for the new variables. This argument can be NULL, in which case all variables are given default names.

Example usage:

```
double obj[] = {1.0, 1.0};
char *names[] = {"var1", "var2"};
error = GRBnewmodel(env, &model, "New", 2, obj, NULL, NULL, NULL, names);
```

GRBcopymodel

```
GRBmodel *  GRBcopymodel (  GRBmodel  *model )
```

Create a copy of an existing model.

Note that pending updates will not be applied to the model, so you should call GRBupdatemodel before copying if you would like those to be included in the copy.

Return value:

A copy of the input model. A NULL return value indicates that a problem was encountered.

Arguments:

model: The model to copy.

Example usage:

```
GRBupdatemodel(orig); /* if you have unstaged changes in orig */
GRBmodel *copy = GRBcopymodel(orig);
```

GRBcopymodeltoenv

```
int  GRBcopymodeltoenv (  GRBmodel  *model,
                          GRBenv    *targetenv,
                          GRBmodel  **resultP )
```

Copy an existing model to a different environment. Multiple threads can not work simultaneously within the same environment. Copies of models must therefore reside in different environments for multiple threads to operate on them simultaneously.

Note that this method itself is not thread safe, so you should either call it from the main thread or protect access to it with a lock.

Note that pending updates will not be applied to the model, so you should call GRBupdatemodel before copying if you would like those to be included in the copy.

For Compute Server users, note that you can copy a model from a client to a Compute Server environment, but it is not possible to copy models from a Compute Server environment to another (client or Compute Server) environment.

Return value:

A non-zero return value indicates that a problem occurred while copying the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to copy.

targetenv: The environment to copy the model into.

resultP: The resulting model copy.

Example usage:

```
error = GRBcopymodeltoenv(orig, env2, &copy);
```

GRBaddconstr

```
int  GRBaddconstr (  GRBmodel     *model,
                     int          numnz,
                     int          *cind,
                     double       *cval,
                     char         sense,
                     double       rhs,
                     const char   *constrname )
```

Add a new linear constraint to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while adding the constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new constraint should be added.

numnz: The number of non-zero coefficients in the new constraint.

cind: Variable indices for non-zero values in the new constraint.

cval: Numerical values for non-zero values in the new constraint.

sense: Sense for the new constraint. Options are `GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`.

rhs: Right-hand side value for the new constraint.

constrname: Name for the new constraint. This argument can be `NULL`, in which case the constraint is given a default name.

Example usage:

```
int    ind[] = {1, 3, 4};
double val[] = {1.0, 2.0, 1.0};
/* x1 + 2 x3 + x4 = 1 */
error = GRBaddconstr(model, 3, ind, val, GRB_EQUAL, 1.0, "New");
```

GRBaddconstrs

```
int  GRBaddconstrs ( GRBmodel    *model,
                     int         numconstrs,
                     int         numnz,
                     int         *cbeg,
                     int         *cind,
                     double      *cval,
                     char        *sense,
                     double      *rhs,
                     const char  **constrnames )
```

Add new linear constraints to a model. Note that, due to our lazy update approach, the new constraints won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

We recommend that you build your model one constraint at a time (using GRBaddconstr), since it introduces no significant overhead and we find that it produces simpler code. Feel free to use this routine if you disagree, though.

If your constraint matrix may contain more than 2 billion non-zero values, you should consider using the GRBXaddconstrs variant of this routine.

Return value:

A non-zero return value indicates that a problem occurred while adding the constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new constraints should be added.

`numconstrs`: The number of new constraints to add.

`numnz`: The total number of non-zero coefficients in the new constraints.

`cbeg`: Constraint matrix non-zero values are passed into this routine in Compressed Sparse Row (CSR) format by this routine. Each constraint in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the variable index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each new constraint has an associated `cbeg` value, indicating the start position of the non-zeros for that constraint in the `cind` and `cval` arrays. This routine requires that the non-zeros for constraint `i` immediately follow those for constraint `i-1` in `cind` and `cval`. Thus, `cbeg[i]` indicates both the index of the first non-zero in constraint `i` and the end of the non-zeros for constraint `i-1`. To give an example of how this representation is used, consider a case where `cbeg[2] = 10` and `cbeg[3] = 12`. This would indicate that constraint 2 has two non-zero values associated with it. Their variable indices can be found in `cind[10]` and `cind[11]`, and the numerical values for those non-zeros can be found in `cval[10]` and `cval[11]`.

`cind`: Variable indices associated with non-zero values. See the description of the `cbeg` argument for more information.

`cval`: Numerical values associated with constraint matrix non-zeros. See the description of the `cbeg` argument for more information.

`sense`: Sense for the new constraints. Options are `GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`.

rhs: Right-hand side values for the new constraints. This argument can be `NULL`, in which case the right-hand side values are set to `0.0`.

constrnames: Names for the new constraints. This argument can be `NULL`, in which case all constraints are given default names.

GRBaddgenconstrXxx

Each of the functions described below adds a new *general constraint* to a model.

Mathematical programming has traditionally defined a set of fundamental constraint types: variable bound constraints, linear constraints, quadratic constraints, integrality constraints, and SOS constraints. These are typically treated directly by the underlying solver (although not always), and are fundamental to the overall algorithm.

Gurobi accepts a number of additional constraint types, which we collectively refer to as *general (function) constraints*. These are typically *not* treated directly by the solver. Rather, they are transformed by presolve into constraints (and variables) chosen from among the fundamental types listed above. In some cases, the resulting constraint or constraints are mathematically equivalent to the original; in others, they are approximations. If such constraints appear in your model, but if you prefer to reformulate them yourself using fundamental constraint types instead, you can certainly do so. However, note that Gurobi can sometimes exploit information contained in the other constraints in the model to build a more efficient formulation than what you might create.

The additional constraint types that fall under this *general constraint* umbrella are:

- GRBaddgenconstrMax: $y = \max(x_1, x_2, ..., c)$
- GRBaddgenconstrMin: $y = \min(x_1, x_2, ..., c)$
- GRBaddgenconstrAbs: $y = |x|$
- GRBaddgenconstrAnd: $y = x_1 \wedge x_2 \wedge x_3 ...$
- GRBaddgenconstrOr: $y = x_1 \vee x_2 \vee x_3 ...$
- GRBaddgenconstrNorm: $y = norm(x_1, x_2, x_3 ...)$
- GRBaddgenconstrIndicator: $y = 1 \rightarrow a'x \leq b$ (an indicator constraint)
- GRBaddgenconstrPWL: $y = pwl(x)$ (a piecewise-linear function, specified using breakpoints)
- GRBaddgenconstrPoly: $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$
- GRBaddgenconstrExp: $y = e^x$
- GRBaddgenconstrExpA: $y = a^x$
- GRBaddgenconstrLog: $y = \log_e(x)$
- GRBaddgenconstrLogA: $y = \log_a(x)$
- GRBaddgenconstrLogistic: $y = \frac{1}{1+e^{-x}}$
- GRBaddgenconstrPow: $y = x^a$

- GRBaddgenconstrSin: $y = \sin(x)$
- GRBaddgenconstrCos: $y = \cos(x)$
- GRBaddgenconstrTan: $y = \tan(x)$

Please refer to this section for additional details on general constraints.

GRBaddgenconstrMax

```
int  GRBaddgenconstrMax (  GRBmodel     *model,
                           const char   *name,
                           int          resvar,
                           int          nvars,
                           const int    *vars,
                           double       constant )
```

Add a new general constraint of type `GRB_GENCONSTR_MAX` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A MAX constraint $r = \max\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the maximum of the operand variables $x_1, \ldots, x_n$ and the constant c.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new general constraint should be added.

`name`: Name for the new general constraint. This argument can be `NULL`, in which case the constraint is given a default name.

`resvar`: The index of the resultant variable r whose value will be equal to the max of the other variables.

`nvars`: The number n of operand variables over which the max will be taken.

`vars`: An array containing the indices of the operand variables x_j over which the max will be taken.

`constant`: An additional operand that allows you to include a constant c among the arguments of the max operation.

Example usage:

```
/* x5 = max(x1, x3, x4, 2.0) */
int ind[] = {1, 3, 4};
error = GRBaddgenconstrMax(model, "maxconstr", 5,
                           3, ind, 2.0);
```

GRBaddgenconstrMin

```
int  GRBaddgenconstrMin (  GRBmodel     *model,
                           const char   *name,
                           int          resvar,
                           int          nvars,
                           const int    *vars,
                           double       constant )
```

Add a new general constraint of type `GRB_GENCONSTR_MIN` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A MIN constraint $r = \min\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the minimum of the operand variables $x_1, \ldots, x_n$ and the constant c.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be `NULL`, in which case the constraint is given a default name.

resvar: The index of the resultant variable r whose value will be equal to the min of the other variables.

nvars: The number n of operand variables over which the min will be taken.

vars: An array containing the indices of the operand variables x_j over which the min will be taken.

constant: An additional operand that allows you to include a constant c among the arguments of the min operation.

Example usage:

```
/* x5 = min(x1, x3, x4, 2.0) */
int ind[] = {1, 3, 4};
error = GRBaddgenconstrMin(model, "minconstr", 5,
                           3, ind, 2.0);
```

GRBaddgenconstrAbs

```
int  GRBaddgenconstrAbs (  GRBmodel     *model,
                           const char   *name,
                           int          resvar,
                           int          argvar )
```

Add a new general constraint of type `GRB_GENCONSTR_ABS` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

An ABS constraint $r = \text{abs}\{x\}$ states that the resultant variable r should be equal to the absolute value of the argument variable x.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

resvar: The index of the resultant variable r whose value will be to equal the absolute value of the argument variable.

argvar: The index of the argument variable x for which the absolute value will be taken.

Example usage:

```
/* x5 = abs(x1) */
error = GRBaddgenconstrAbs(model, "absconstr", 5, 1);
```

GRBaddgenconstrAnd

```
int  GRBaddgenconstrAnd (  GRBmodel     *model,
                           const char   *name,
                           int          resvar,
                           int          nvars,
                           const int    *vars )
```

Add a new general constraint of type GRB_GENCONSTR_AND to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

An AND constraint $r = \text{and}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if all of the operand variables $x_1, \ldots, x_n$ are equal to 1. If any of the operand variables is 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

resvar: The index of the binary resultant variable r whose value will be equal to the AND concatenation of the other variables.

nvars: The number n of binary operand variables over which the AND will be taken.

vars: An array containing the indices of the binary operand variables x_j over which the AND concatenation will be taken.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Example usage:

```
/* x5 = and(x1, x3, x4) */
int ind[] = {1, 3, 4};
error = GRBaddgenconstrAnd(model, "andconstr", 5, 3, ind);
```

GRBaddgenconstrOr

```
int  GRBaddgenconstrOr (  GRBmodel    *model,
                          const char  *name,
                          int         resvar,
                          int         nvars,
                          const int   *vars )
```

Add a new general constraint of type `GRB_GENCONSTR_OR` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

An OR constraint $r = \text{or}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if any of the operand variables $x_1, \ldots, x_n$ is equal to 1. If all operand variables are 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be `NULL`, in which case the constraint is given a default name.

resvar: The index of the binary resultant variable r whose value will be equal to the OR concatenation of the other variables.

nvars: The number n of binary operand variables over which the OR will be taken.

vars: An array containing the indices of the binary operand variables x_j over which the OR concatenation will be taken.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Example usage:

```
/* x5 = or(x1, x3, x4) */
int ind[] = {1, 3, 4};
error = GRBaddgenconstrOr(model, "orconstr", 5, 3, ind);
```

GRBaddgenconstrNorm

```
int  GRBaddgenconstrNorm (  GRBmodel     *model,
                            const char   *name,
                            int          resvar,
                            int          nvars,
                            const int    *vars,
                            double       which )
```

Add a new general constraint of type `GRB_GENCONSTR_NORM` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A NORM constraint $r = \text{norm}\{x_1, \ldots, x_n\}$ states that the resultant variable r should be equal to the vector norm of the argument vector $x_1, \ldots, x_n$.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be `NULL`, in which case the constraint is given a default name.

resvar: The index of the resultant variable r whose value will be equal to the NORM of the other variables.

nvars: The number n of operand variables over which the NORM will be taken.

vars: An array containing the indices of the operand variables x_j over which the NORM will be taken. Note that this array may not contain duplicates.

which: Which norm to use. Options are 0, 1, 2, and GRB_INFINITY.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Example usage:

```
/* x5 = 2-norm(x1, x3, x4) */
int ind[] = {1, 3, 4};
error = GRBaddgenconstrNorm(model, "orconstr", 5, 3, ind, 2.0);
```

GRBaddgenconstrIndicator

```
int  GRBaddgenconstrIndicator (  GRBmodel      *model,
                                 const char    *name,
                                 int           binvar,
                                 int           binval,
                                 int           nvars,
                                 const int     *ind,
                                 const double  *val,
                                 char          sense,
                                 double        rhs )
```

Add a new general constraint of type `GRB_GENCONSTR_INDICATOR` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the

model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

An INDICATOR constraint $z = f \rightarrow a^T x \leq b$ states that if the binary indicator variable z is equal to f, where $f \in \{0, 1\}$, then the linear constraint $a^T x \leq b$ should hold. On the other hand, if $z = 1 - f$, the linear constraint may be violated. The sense of the linear constraint can also be specified to be "=" or "≥".

Note that the indicator variable z of a constraint will be forced to be binary, independent of how it was created.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

binvar: The index of the binary indicator variable z.

binval: The value f for the binary indicator variable that would force the linear constraint to be satisfied (0 or 1).

nvars: The number n of non-zero coefficients in the linear constraint triggered by the indicator.

ind: Indices for the variables x_j with non-zero values in the linear constraint.

val: Numerical values for non-zero values a_j in the linear constraint.

sense: Sense for the linear constraint. Options are GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL.

rhs: Right-hand side value for the linear constraint.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Example usage:

```
/* x7 = 1 -> x1 + 2 x3 + x4 = 1 */
int    ind[] = {1, 3, 4};
double val[] = {1.0, 2.0, 1.0};
error = GRBaddgenconstrIndicator(model, NULL, 7, 1,
                                 3, ind, val, GRB_EQUAL, 1.0);
```

GRBaddgenconstrPWL

```
int  GRBaddgenconstrPWL (  GRBmodel    *model,
                           const char  *name,
                           int         xvar,
                           int         yvar,
                           int         npts,
                           double      *xpts,
                           double      *ypts )
```

Add a new general constraint of type GRB_GENCONSTR_PWL to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model

(using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A piecewise-linear (PWL) constraint states that the relationship $y = f(x)$ must hold between variables x and y, where f is a piecewise-linear function. The breakpoints for f are provided as arguments. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

xvar: The index of variable x.

yvar: The index of variable y.

npts: The number of points that define the piecewise-linear function.

xpts: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

ypts: The y values for the points that define the piecewise-linear function.

Example usage:

```
double xpts[] = {1, 3, 5};
double ypts[] = {1, 2, 4};
error = GRBaddgenconstr(model, "pwl", xvar, yvar, 3, x, y);
```

GRBaddgenconstrPoly

```
int  GRBaddgenconstrPoly (  GRBmodel    *model,
                            const char  *name,
                            int         xvar,
                            int         yvar,
                            int         plen,
                            double      *p,
                            const char  *options )
```

Add a new general constraint of type GRB_GENCONSTR_POLY to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A polynomial function constraint states that the relationship $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$ should hold between variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new general constraint should be added.

`name`: Name for the new general constraint. This argument can be `NULL`, in which case the constraint is given a default name.

`xvar`: The index of variable x.

`yvar`: The index of variable y.

`plen`: The length of coefficient array `p`. If x^d is the highest power term, then `plen` should be $d + 1$.

`p`: The coefficients for the polynomial function (starting with the coefficient for the highest power).

`options`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = 3 x^4 + 7 x + 3 = 3 x^4 + 0 x^3 + 0 x^2 + 7 x + 3 */
int plen = 5;
double p[] = {3, 0, 0, 7, 3};
error = GRBaddgenconstrPoly(model, "poly", xvar, yvar, 5, p, "");
```

GRBaddgenconstrExp

```
int  GRBaddgenconstrExp (  GRBmodel    *model,
                           const char  *name,
                           int         xvar,
                           int         yvar,
                           const char  *options )
```

Add a new general constraint of type `GRB_GENCONSTR_EXP` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A natural exponential function constraint states that the relationship $y = exp(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

xvar: The index of variable x.

yvar: The index of variable y.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = exp(x) */
error = GRBaddgenconstrExp(model, "exp", xvar, yvar, "");
```

GRBaddgenconstrExpA

```
int  GRBaddgenconstrExpA (  GRBmodel     *model,
                            const char   *name,
                            int          xvar,
                            int          yvar,
                            double       a,
                            const char   *options )
```

Add a new general constraint of type GRB_GENCONSTR_EXPA to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

An exponential function constraint states that the relationship $y = a^x$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

xvar: The index of variable x.

yvar: The index of variable y.

a: The base of the function, $a > 0$.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = 3^x */
error = GRBaddgenconstrExpA(model, "expa", xvar, yvar, 3.0, "");
```

GRBaddgenconstrLog

```
int  GRBaddgenconstrLog (  GRBmodel    *model,
                           const char  *name,
                           int         xvar,
                           int         yvar,
                           const char  *options )
```

Add a new general constraint of type GRB_GENCONSTR_LOG to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A natural logarithmic function constraint states that the relationship $y = log(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

xvar: The index of variable x.

yvar: The index of variable y.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = log(x) */
error = GRBaddgenconstrLog(model, "log", xvar, yvar, "FuncPieces=-1 FuncPieceError=0.001
```

GRBaddgenconstrLogA

```
int  GRBaddgenconstrLogA (  GRBmodel     *model,
                            const char   *name,
                            int          xvar,
                            int          yvar,
                            double       a,
                            const char   *options )
```

Add a new general constraint of type `GRB_GENCONSTR_LOGA` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A logarithmic function constraint states that the relationship $y = log_a(x)$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new general constraint should be added.

`name`: Name for the new general constraint. This argument can be `NULL`, in which case the constraint is given a default name.

`xvar`: The index of variable x.

`yvar`: The index of variable y.

`a`: The base of the function, $a > 0$.

`options`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = log_10(x) */
error = GRBaddgenconstrLogA(model, "loga", xvar, yvar, 10.0, "");
```

GRBaddgenconstrLogistic

```
int  GRBaddgenconstrLogistic (  GRBmodel    *model,
                                const char  *name,
                                int         xvar,
                                int         yvar,
                                const char  *options )
```

Add a new general constraint of type GRB_GENCONSTR_LOGISTIC to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A logistic function constraint states that the relationship $y = \frac{1}{1+e^{-x}}$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

xvar: The index of variable x.

yvar: The index of variable y.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = 1 / (1 + exp(-x)) */
error = GRBaddgenconstrLogistic(model, "logistic", xvar, yvar, "");
```

GRBaddgenconstrPow

```
int  GRBaddgenconstrPow (  GRBmodel    *model,
                           const char  *name,
                           int         xvar,
                           int         yvar,
                           double      a,
                           const char  *options )
```

Add a new general constraint of type GRB_GENCONSTR_POW to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A power function constraint states that the relationship $y = x^a$ should hold for variables x and y, where a is the (constant) exponent.

If the exponent a is negative, the lower bound on x must be strictly positive. If the exponent isn't an integer, the lower bound on x must be non-negative.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

xvar: The index of variable x.

yvar: The index of variable y.

a: The exponent of the function.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = sqrt(x) */
error = GRBaddgenconstrPow(model, "pow", xvar, yvar, 0.5, "");
```

GRBaddgenconstrSin

```
int  GRBaddgenconstrSin (  GRBmodel    *model,
                           const char  *name,
                           int         xvar,
                           int         yvar,
                           const char  *options )
```

Add a new general constraint of type GRB_GENCONSTR_SIN to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A sine function constraint states that the relationship $y = sin(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new general constraint should be added.

`name`: Name for the new general constraint. This argument can be `NULL`, in which case the constraint is given a default name.

`xvar`: The index of variable x.

`yvar`: The index of variable y.

`options`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = sin(x) */
error = GRBaddgenconstrSin(model, "sin", xvar, yvar, "");
```

GRBaddgenconstrCos

```
int  GRBaddgenconstrCos (  GRBmodel    *model,
                           const char  *name,
                           int         xvar,
                           int         yvar,
                           const char  *options )
```

Add a new general constraint of type `GRB_GENCONSTR_COS` to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A cosine function constraint states that the relationship $y = cos(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

xvar: The index of variable x.

yvar: The index of variable y.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = cos(x) */
error = GRBaddgenconstrCos(model, "cos", xvar, yvar, "FuncPieces=-2");
```

GRBaddgenconstrTan

```
int  GRBaddgenconstrTan (  GRBmodel    *model,
                           const char  *name,
                           int         xvar,
                           int         yvar,
                           const char  *options )
```

Add a new general constraint of type GRB_GENCONSTR_TAN to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A tangent function constraint states that the relationship $y = tan(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Return value:

A non-zero return value indicates that a problem occurred while adding the general constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new general constraint should be added.

name: Name for the new general constraint. This argument can be NULL, in which case the constraint is given a default name.

xvar: The index of variable x.

yvar: The index of variable y.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Example usage:

```
/* y = tan(x) */
error = GRBaddgenconstrTan(model, "tan", xvar, yvar, "");
```

GRBaddqconstr

```
int  GRBaddqconstr (  GRBmodel    *model,
                      int         numlnz,
                      int         *lind,
                      double      *lval,
                      int         numqnz,
                      int         *qrow,
                      int         *qcol,
                      double      *qval,
                      char        sense,
                      double      rhs,
                      const char  *constrname )
```

Add a new quadratic constraint to a model. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

A quadratic constraint consists of a set of quadratic terms, a set of linear terms, a sense, and a right-hand side value: $x^T Q x + q^T x \leq b$. The quadratic terms are input through the **numqnz**, **qrow**, **qcol**, and **qval** arguments, and the linear terms are input through the **numlnz**, **lind**, and **lval** arguments.

Important note: Gurobi can handle both convex and non-convex quadratic constraints. The differences between them can be both important and subtle. Refer to this discussion for additional information.

Return value:

A non-zero return value indicates that a problem occurred while adding the quadratic constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new constraint should be added.

numlnz: The number of linear terms in the new quadratic constraint.

lind: Variable indices associated with linear terms.

lval: Numerical values associated with linear terms.

numqlnz: The number of quadratic terms in the new quadratic constraint.

qrow: Row indices associated with quadratic terms. A quadratic term is represented using three values: a pair of indices (stored in **qrow** and **qcol**), and a coefficient (stored in **qval**).

The associated arguments arrays provide the corresponding values for each quadratic term. To give an example, if you wish to input quadratic terms $2x_0^2 + x_0x_1 + x_1^2$, you would call this routine with `numqnz=3`, `qrow[] = {0, 0, 1}`, `qcol[] = {0, 1, 1}`, and `qval[] = {2.0, 1.0, 1.0}`.

qcol: Column indices associated with quadratic terms. See the description of the `qrow` argument for more information.

qval: Numerical values associated with quadratic terms. See the description of the `qrow` argument for more information.

sense: Sense for the new quadratic constraint. Options are `GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`.

rhs: Right-hand side value for the new quadratic constraint.

constrname: Name for the new quadratic constraint. This argument can be `NULL`, in which case the constraint is given a default name.

Example usage:

```
int    lind[] = {1, 2};
double lval[] = {2.0, 1.0};
int    qrow[] = {0, 0, 1};
int    qcol[] = {0, 1, 1};
double qval[] = {2.0, 1.0, 1.0};
/* 2 x0^2 + x0 x1 + x1^2 + 2 x1 + x2 <= 1 */
error = GRBaddqconstr(model, 2, lind, lval, 3, qrow, qcol, qval,
                      GRB_LESS_EQUAL, 1.0, "New");
```

GRBaddqpterms

```
int  GRBaddqpterms (  GRBmodel  *model,
                      int       numqnz,
                      int       *qrow,
                      int       *qcol,
                      double    *qval )
```

Add new quadratic objective terms into an existing model. Note that new terms are (numerically) added into existing terms, and that adding a term in row `i` and column `j` is equivalent to adding a term in row `j` and column `i`. You can add all quadratic objective terms in a single call, or you can add them incrementally in multiple calls.

Note that, due to our lazy update approach, the new quadratic terms won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

To build an objective that contains both linear and quadratic terms, use this routine to add the quadratic terms and use the Obj attribute to add the linear terms.

If you wish to change a quadratic term, you can either add the difference between the current term and the desired term using this routine, or you can call GRBdelq to delete all quadratic terms, and then rebuild your new quadratic objective from scratch.

Return value:

A non-zero return value indicates that a problem occurred while adding the quadratic terms. Refer to the Error Code table for a list of possible return values. Details on the error can

be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new quadratic objective terms should be added.

`numqnz`: The number of new quadratic objective terms to add.

`qrow`: Row indices associated with quadratic terms. A quadratic term is represented using three values: a pair of indices (stored in `qrow` and `qcol`), and a coefficient (stored in `qval`). The three argument arrays provide the corresponding values for each quadratic term. To give an example, to represent $2x_0^2 + x_0x_1 + x_1^2$, you would have `numqnz=3`, `qrow[] = {0, 0, 1}`, `qcol[] = {0, 1, 1}`, and `qval[] = {2.0, 1.0, 1.0}`.

`qcol`: Column indices associated with quadratic terms. See the description of the `qrow` argument for more information.

`qval`: Numerical values associated with quadratic terms. See the description of the `qrow` argument for more information.

Important notes:

Note that building quadratic objectives requires some care, particularly if you are migrating an application from another solver. Some solvers require you to specify the entire Q matrix, while others only accept the lower triangle. In addition, some solvers include an implicit 0.5 multiplier on Q, while others do not. The Gurobi interface is built around quadratic terms, rather than a Q matrix. If your quadratic objective contains a term `2 x y`, you can enter it as a single term, `2 x y`, or as a pair of terms, `x y` and `y x`.

Example usage:

```
int    qrow[] = {0, 0, 1};
int    qcol[] = {0, 1, 1};
double qval[] = {2.0, 1.0, 3.0};
/* minimize 2 x^2 + x*y + 3 y^2 */
error = GRBaddqpterms(model, 3, qrow, qcol, qval);
```

GRBaddrangeconstr

```
int  GRBaddrangeconstr (  GRBmodel    *model,
                          int         numnz,
                          int         *cind,
                          double      *cval,
                          double      lower,
                          double      upper,
                          const char  *constrname )
```

Add a new range constraint to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution. Note that, due to our lazy update approach, the new constraint won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while adding the constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new constraint should be added.

numnz: The number of non-zero coefficients in the linear expression.

cind: Variable indices for non-zero values in the linear expression.

cval: Numerical values for non-zero values in the linear expression.

lower: Lower bound on linear expression.

upper: Upper bound on linear expression.

constrname: Name for the new constraint. This argument can be NULL, in which case the constraint is given a default name.

Important notes:

Note that adding a range constraint to the model adds both a new constraint and a new variable. If you are keeping a count of the variables in the model, remember to add one whenever you add a range.

Note also that range constraints are stored internally as equality constraints. We use the extra variable that is added with a range constraint to capture the range information. Thus, the Sense attribute on a range constraint will always be GRB_EQUAL. In particular introducing a range constraint

$$L \leq a^T x \leq U$$

is equivalent to adding a slack variable s and the following constraints

$$\begin{aligned} a^T x - s &= L \\ 0 \leq s &\leq U - L. \end{aligned}$$

Example usage:

```
int    ind[] = {1, 3, 4};
double val[] = {1.0, 2.0, 3.0};
/* 1 <= x1 + 2 x3 + 3 x4 <= 2 */
error = GRBaddrangeconstr(model, 3, ind, val, 1.0, 2.0, "NewRange");
```

GRBaddrangeconstrs

```
int  GRBaddrangeconstrs (  GRBmodel     *model,
                           int          numconstrs,
                           int          numnz,
                           int          *cbeg,
                           int          *cind,
                           double       *cval,
                           double       *lower,
                           double       *upper,
                           const char   **constrnames )
```

Add new range constraints to a model. A range constraint states that the value of the input expression must be between the specified lower and upper bounds in any solution. Note that, due to our lazy update approach, the new constraints won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

If your constraint matrix may contain more than 2 billion non-zero values, you should consider using the GRBXaddrangeconstrs variant of this routine.

Return value:

A non-zero return value indicates that a problem occurred while adding the constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new constraints should be added.

`numconstrs`: The number of new constraints to add.

`numnz`: The total number of non-zero coefficients in the new constraints.

`cbeg`: Constraint matrix non-zero values are passed into this routine in Compressed Sparse Row (CSR) format by this routine. Each constraint in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the variable index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each new constraint has an associated `cbeg` value, indicating the start position of the non-zeros for that constraint in the `cind` and `cval` arrays. This routine requires that the non-zeros for constraint `i` immediately follow those for constraint `i-1` in `cind` and `cval`. Thus, `cbeg[i]` indicates both the index of the first non-zero in constraint `i` and the end of the non-zeros for constraint `i-1`. To give an example of how this representation is used, consider a case where `cbeg[2] = 10` and `cbeg[3] = 12`. This would indicate that constraint 2 has two non-zero values associated with it. Their variable indices can be found in `cind[10]` and `cind[11]`, and the numerical values for those non-zeros can be found in `cval[10]` and `cval[11]`.

`cind`: Variable indices associated with non-zero values. See the description of the `cbeg` argument for more information.

`cval`: Numerical values associated with constraint matrix non-zeros. See the description of the `cbeg` argument for more information.

`lower`: Lower bounds for the linear expressions.

`upper`: Upper bounds for the linear expressions.

`constrnames`: Names for the new constraints. This argument can be `NULL`, in which case all constraints are given default names.

Important notes:

Note that adding a range constraint to the model adds both a new constraint and a new variable. If you are keeping a count of the variables in the model, remember to add one for each range constraint.

Note also that range constraints are stored internally as equality constraints. We use the extra variable that is added with a range constraint to capture the range information. Thus, the Sense attribute on a range constraint will always be `GRB_EQUAL`.

GRBaddsos

```
int  GRBaddsos (  GRBmodel  *model,
                  int       numsos,
                  int       nummembers,
                  int       *types,
                  int       *beg,
                  int       *ind,
                  double    *weight )
```

Add new Special Ordered Set (SOS) constraints to a model. Note that, due to our lazy update approach, the new SOS constraints won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Please refer to this section for details on SOS constraints.

Return value:

A non-zero return value indicates that a problem occurred while adding the SOS constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new SOSs should be added.

`numsos`: The number of new SOSs to add.

`nummembers`: The total number of SOS members in the new SOSs.

`types`: The types of the SOS sets. SOS sets can be of type `GRB_SOS_TYPE1` or `GRB_SOS_TYPE2`.

`beg`: The members of the added SOS sets are passed into this routine in Compressed Sparse Row (CSR) format. Each SOS is represented as a list of index-value pairs, where each index entry provides the variable index for an SOS member, and each value entry provides the weight of that variable in the corresponding SOS set. Each new SOS has an associated `beg` value, indicating the start position of the SOS member list in the `ind` and `weight` arrays. This routine requires that the members for SOS `i` immediately follow those for SOS `i-1` in `ind` and `weight`. Thus, `beg[i]` indicates both the index of the first non-zero in constraint `i` and the end of the non-zeros for constraint `i-1`. To give an example of how this representation is used, consider a case where `beg[2] = 10` and `beg[3] = 12`. This would indicate that SOS number 2 has two members. Their variable indices can be found in `ind[10]` and `ind[11]`, and the associated weights can be found in `weight[10]` and `weight[11]`.

`ind`: Variable indices associated with SOS members. See the description of the `beg` argument for more information.

`weight`: Weights associated with SOS members. See the description of the `beg` argument for more information.

Example usage:

```
int    types[]  = {GRB_SOS_TYPE1, GRB_SOS_TYPE1};
int    beg[]    = {0, 2};
int    ind[]    = {1, 2, 1, 3};
double weight[] = {1, 2, 1, 2};
error = GRBaddsos(model, 2, 4, types, beg, ind, weight);
```

GRBaddvar

```
int  GRBaddvar ( GRBmodel    *model,
                 int         numnz,
                 int         *vind,
                 double      *vval,
                 double      obj,
                 double      lb,
                 double      ub,
                 char        vtype,
                 const char  *varname )
```

Add a new variable to a model. Note that, due to our lazy update approach, the new variable won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while adding the variable. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new variable should be added.

numnz: The number of non-zero coefficients in the new column.

vind: Constraint indices associated with non-zero values for the new variable.

vval: Numerical values associated with non-zero values for the new variable.

obj: Objective coefficient for the new variable.

lb: Lower bound for the new variable.

ub: Upper bound for the new variable.

vtype: Type for the new variable. Options are GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT.

varname: Name for the new variable. This argument can be NULL, in which case the variable is given a default name.

Example usage:

```
int    ind[] = {1, 3, 4};
double val[] = {1.0, 1.0, 1.0};
error = GRBaddvar(model, 3, ind, val, 1.0, 0.0, GRB_INFINITY,
                  GRB_CONTINUOUS, "New");
```

GRBaddvars

```
int  GRBaddvars (  GRBmodel    *model,
                   int         numvars,
                   int         numnz,
                   int         *vbeg,
                   int         *vind,
                   double      *vval,
                   double      *obj,
                   double      *lb,
                   double      *ub,
                   char        *vtype,
                   const char  **varnames )
```

Add new variables to a model. Note that, due to our lazy update approach, the new variables won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

If your constraint matrix may contain more than 2 billion non-zero values, you should consider using the GRBXaddvars variant of this routine.

Return value:

A non-zero return value indicates that a problem occurred while adding the variables. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new variables should be added.

numvars: The number of new variables to add.

numnz: The total number of non-zero coefficients in the new columns.

vbeg: Constraint matrix non-zero values are passed into this routine in Compressed Sparse Column (CSC) format. Each column in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the constraint index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each variable in the model has a `vbeg`, indicating the start position of the non-zeros for that variable in the `vind` and `vval` arrays. This routine requires columns to be stored contiguously, so the start position for a variable is the end position for the previous variable. To give an example, if `vbeg[2] = 10` and `vbeg[3] = 12`, that would indicate that variable 2 has two non-zero values associated with it. Their constraint indices can be found in `vind[10]` and `vind[11]`, and the numerical values for those non-zeros can be found in `vval[10]` and `vval[11]`.

vind: Constraint indices associated with non-zero values. See the description of the `vbeg` argument for more information.

vval: Numerical values associated with constraint matrix non-zeros. See the description of the `vbeg` argument for more information.

obj: Objective coefficients for the new variables. This argument can be `NULL`, in which case the objective coefficients are set to `0.0`.

lb: Lower bounds for the new variables. This argument can be `NULL`, in which case all variables get lower bounds of `0.0`.

ub: Upper bounds for the new variables. This argument can be `NULL`, in which case all variables get infinite upper bounds.

vtype: Types for the variables. Options are `GRB_CONTINUOUS`, `GRB_BINARY`, `GRB_INTEGER`, `GRB_SEMICONT`, or `GRB_SEMIINT`. This argument can be `NULL`, in which case all variables are assumed to be continuous.

varnames: Names for the new variables. This argument can be `NULL`, in which case all variables are given default names.

GRBchgcoeffs

```
int  GRBchgcoeffs (  GRBmodel  *model,
                     int       numchgs,
                     int       *cind,
                     int       *vind,
                     double    *val )
```

Change a set of constraint matrix coefficients. This routine can be used to set a non-zero coefficient to zero, to create a non-zero coefficient where the coefficient is currently zero, or to change an existing non-zero coefficient to a new non-zero value. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the changes won't actually be integrated into the model until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

If your constraint matrix may contain more than 2 billion non-zero values, you should consider using the GRBXchgcoeffs variant of this routine.

Return value:

A non-zero return value indicates that a problem occurred while performing the modification. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to modify.

numchgs: The number of coefficients to modify.

cind: Constraint indices for the coefficients to modify.

vind: Variable indices for the coefficients to modify.

val: The new values for the coefficients. For example, if `cind[0] = 1`, `vind[0] = 3`, and `val[0] = 2.0`, then the coefficient in constraint 1 associated with variable 3 would be changed to 2.0.

Example usage:

```
int cind[] = {0, 1};
int vind[] = {0, 0};
double val[] = {1.0, 1.0};
error = GRBchgcoeffs(model, 2, cind, vind, val);
```

GRBdelconstrs

```
int  GRBdelconstrs (  GRBmodel  *model,
                      int       numdel,
                      int       *ind )
```

Delete a list of constraints from an existing model. Note that, due to our lazy update approach, the constraints won't actually be removed until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while deleting the constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to modify.

numdel: The number of constraints to remove.

ind: The indices of the constraints to remove.

Example usage:

```
int first_four[] = {0, 1, 2, 3};
error = GRBdelconstrs(model, 4, first_four);
```

GRBdelgenconstrs

```
int  GRBdelgenconstrs (  GRBmodel  *model,
                         int       numdel,
                         int       *ind )
```

Delete a list of general constraints from an existing model. Note that, due to our lazy update approach, the general constraints won't actually be removed until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while deleting the constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to modify.

numdel: The number of general constraints to remove.

ind: The indices of the general constraints to remove.

Example usage:

```
int first_four[] = {0, 1, 2, 3};
error = GRBdelgenconstrs(model, 4, first_four);
```

GRBdelq

```
int  GRBdelq (  GRBmodel  *model )
```

Delete all quadratic objective terms from an existing model. Note that, due to our lazy update approach, the quadratic terms won't actually be removed until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while deleting the quadratic objective terms. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to modify.

Example usage:

```
error = GRBdelq(model);
```

GRBdelqconstrs

```
int  GRBdelqconstrs (  GRBmodel  *model,
                       int       numdel,
                       int       *ind )
```

Delete a list of quadratic constraints from an existing model. Note that, due to our lazy update approach, the quadratic constraints won't actually be removed until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while deleting the quadratic constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to modify.

`numdel`: The number of quadratic constraints to remove.

`ind`: The indices of the quadratic constraints remove.

Example usage:

```
int first_four[] = {0, 1, 2, 3};
error = GRBdelqconstrs(model, 4, first_four);
```

GRBdelsos

```
int  GRBdelsos (  GRBmodel  *model,
                  int       numdel,
                  int       *ind )
```

Delete a list of Special Ordered Set (SOS) constraints from an existing model. Note that, due to our lazy update approach, the SOS constraints won't actually be removed until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while deleting the constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to modify.

numdel: The number of SOSs to remove.

ind: The indices of the SOSs to remove.

Example usage:

```
int first_four[] = {0, 1, 2, 3};
error = GRBdelsos(model, 4, first_four);
```

GRBdelvars

```
int  GRBdelvars (  GRBmodel  *model,
                   int       numdel,
                   int       *ind )
```

Delete a list of variables from an existing model. Note that, due to our lazy update approach, the variables won't actually be removed until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while deleting the variables. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to modify.

numdel: The number of variables to remove.

ind: The indices of the variables to remove.

Example usage:

```
int first_two[] = {0, 1};
error = GRBdelvars(model, 2, first_two);
```

GRBsetobjectiven

```
int  GRBsetobjectiven (  GRBmodel    *model,
                         int         index,
                         int         priority,
                         double      weight,
                         double      abstol,
                         double      reltol,
                         const char  *name,
                         double      constant,
                         int         lnz,
                         int         *lind,
                         double      *lval )
```

Set an alternative optimization objective equal to a linear expression.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Note that you can also modify an alternative objective using the ObjN variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `ObjN` attribute can be used to modify individual terms.

Note that, due to our lazy update approach, the new alternative objective won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while setting the alternative objective. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model in which the new alternative objective should be set.

`index`: Index for new objective. If you use an index of 0, this routine will change the primary optimization objective.

`priority`: Priority for the alternative objective. This initializes the ObjNPriority attribute for this objective.

`weight`: Weight for the alternative objective. This initializes the ObjNWeight attribute for this objective.

`abstol`: Absolute tolerance for the alternative objective. This initializes the ObjNAbsTol attribute for this objective.

`reltol`: Relative tolerance for the alternative objective. This initializes the ObjNRelTol attribute for this objective.

`name`: Name of the alternative objective. This initializes the ObjNName attribute for this objective.

`constant`: Constant part of the linear expression for the new alternative objective.

`lnz`: Number of non-zero coefficients in new alternative objective.

`lind`: Variable indices for non-zero values in new alternative objective.

`lval`: Numerical values for non-zero values in new alternative objective.

Example usage:

```
int ind[] = {0, 1, 2};
double val[] = {1.0, 1.0, 1.0};
/* Objective expression: x0 + x1 + x2 */
error = GRBsetobjectiven(model, 0, 1, 0.0, 0.0, 0.0, "primary",
                         0.0, 3, ind, val);
```

GRBsetpwlobj

```
int  GRBsetpwlobj (  GRBmodel  *model,
                     int       var,
                     int       npoints,
                     double    *x,
                     double    *y )
```

Set a piecewise-linear objective function for a variable.

The arguments to this method specify a list of points that define a piecewise-linear objective function for a single variable. Specifically, the x and y arguments give coordinates for the vertices of the function.

For additional details on piecewise-linear objective functions, refer to this discussion.

Note that, due to our lazy update approach, the new piecewise-linear objective won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while setting the piecewise-linear objective. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to modify.

`var`: The variable whose objective function is being changed.

`npoints`: The number of points that define the piecewise-linear function.

`x`: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

`y`: The y values for the points that define the piecewise-linear function.

Example usage:

```
double x[] = {1, 3, 5};
double y[] = {1, 2, 4};
error = GRBsetpwlobj(model, var, 3, x, y);
```

GRBupdatemodel

```
int  GRBupdatemodel (  GRBmodel  *model )
```

Process any pending model modifications.

Return value:

A non-zero return value indicates that a problem occurred while updating the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to update.

Example usage:

```
error = GRBupdatemodel(model);
```

GRBfreemodel

```
int  GRBfreemodel (  GRBmodel  *model )
```

Free a model and release the associated memory.

Return value:

A non-zero return value indicates that a problem occurred while freeing the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to be freed.

Example usage:

```
error = GRBfreemodel(model);
```

GRBXaddconstrs

```
int   GRBXaddconstrs (  GRBmodel     *model,
                        int          numconstrs,
                        size_t       numnz,
                        size_t       *cbeg,
                        int          *cind,
                        double       *cval,
                        char         *sense,
                        double       *rhs,
                        const char   **constrnames )
```

The `size_t` version of GRBaddconstrs. The two arguments that count non-zero values are of type `size_t` in this version to support models with more than 2 billion non-zero values.

Add new linear constraints to a model. Note that, due to our lazy update approach, the new constraints won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

We recommend that you build your model one constraint at a time (using GRBaddconstr), since it introduces no significant overhead and we find that it produces simpler code. Feel free to use this routine if you disagree, though.

Return value:

A non-zero return value indicates that a problem occurred while adding the constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to which the new constraints should be added.

`numconstrs`: The number of new constraints to add.

`numnz`: The total number of non-zero coefficients in the new constraints.

`cbeg`: Constraint matrix non-zero values are passed into this routine in Compressed Sparse Row (CSR) format by this routine. Each constraint in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the variable index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each new constraint has an associated `cbeg` value, indicating the start position of the non-zeros for that constraint in the `cind` and `cval` arrays. This routine requires that the non-zeros for constraint `i` immediately follow those for constraint `i-1` in `cind` and `cval`. Thus, `cbeg[i]` indicates both the index of the first non-zero in constraint `i` and the end of the non-zeros for constraint `i-1`. To give an example of how this representation is used, consider a case where `cbeg[2] = 10` and `cbeg[3] = 12`. This would indicate that constraint 2 has two

non-zero values associated with it. Their variable indices can be found in `cind[10]` and `cind[11]`, and the numerical values for those non-zeros can be found in `cval[10]` and `cval[11]`.

cind: Variable indices associated with non-zero values. See the description of the `cbeg` argument for more information.

cval: Numerical values associated with constraint matrix non-zeros. See the description of the `cbeg` argument for more information.

sense: Sense for the new constraints. Options are `GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`.

rhs: Right-hand side values for the new constraints. This argument can be `NULL`, in which case the right-hand side values are set to `0.0`.

constrnames: Names for the new constraints. This argument can be `NULL`, in which case all constraints are given default names.

GRBXaddrangeconstrs

```
int  GRBXaddrangeconstrs (  GRBmodel     *model,
                            int          numconstrs,
                            size_t       numnz,
                            size_t       *cbeg,
                            int          *cind,
                            double       *cval,
                            double       *lower,
                            double       *upper,
                            const char   **constrnames )
```

The `size_t` version of GRBaddrangeconstrs. The argument that counts non-zero values is of type `size_t` in this version to support models with more than 2 billion non-zero values.

Add new range constraints to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution. Note that, due to our lazy update approach, the new constraints won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while adding the constraints. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new constraints should be added.

numconstrs: The number of new constraints to add.

numnz: The total number of non-zero coefficients in the new constraints.

cbeg: Constraint matrix non-zero values are passed into this routine in Compressed Sparse Row (CSR) format by this routine. Each constraint in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the variable index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each new constraint has an associated `cbeg` value, indicating the start position of the non-zeros for that constraint in the `cind` and `cval` arrays. This routine requires that the non-zeros for

constraint `i` immediately follow those for constraint `i-1` in `cind` and `cval`. Thus, `cbeg[i]` indicates both the index of the first non-zero in constraint `i` and the end of the non-zeros for constraint `i-1`. To give an example of how this representation is used, consider a case where `cbeg[2] = 10` and `cbeg[3] = 12`. This would indicate that constraint 2 has two non-zero values associated with it. Their variable indices can be found in `cind[10]` and `cind[11]`, and the numerical values for those non-zeros can be found in `cval[10]` and `cval[11]`.

cind: Variable indices associated with non-zero values. See the description of the `cbeg` argument for more information.

cval: Numerical values associated with constraint matrix non-zeros. See the description of the `cbeg` argument for more information.

lower: Lower bounds for the linear expressions.

upper: Upper bounds for the linear expressions.

constrnames: Names for the new constraints. This argument can be `NULL`, in which case all constraints are given default names.

Important notes:

Note that adding a range constraint to the model adds both a new constraint and a new variable. If you are keeping a count of the variables in the model, remember to add one for each range constraint.

Note also that range constraints are stored internally as equality constraints. We use the extra variable that is added with a range constraint to capture the range information. Thus, the Sense attribute on a range constraint will always be `GRB_EQUAL`.

GRBXaddvars

```
int  GRBXaddvars (  GRBmodel    *model,
                    int         numvars,
                    size_t      numnz,
                    size_t      *vbeg,
                    int         *vind,
                    double      *vval,
                    double      *obj,
                    double      *lb,
                    double      *ub,
                    char        *vtype,
                    const char  **varnames )
```

The `size_t` version of GRBaddvars. The two arguments that count non-zero values are of type `size_t` in this version to support models with more than 2 billion non-zero values.

Add new variables to a model. Note that, due to our lazy update approach, the new variables won't actually be added until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while adding the variables. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model to which the new variables should be added.

numvars: The number of new variables to add.

numnz: The total number of non-zero coefficients in the new columns.

vbeg: Constraint matrix non-zero values are passed into this routine in Compressed Sparse Column (CSC) format. Each column in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the constraint index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each variable in the model has a `vbeg`, indicating the start position of the non-zeros for that variable in the `vind` and `vval` arrays. This routine requires columns to be stored contiguously, so the start position for a variable is the end position for the previous variable. To give an example, if `vbeg[2] = 10` and `vbeg[3] = 12`, that would indicate that variable 2 has two non-zero values associated with it. Their constraint indices can be found in `vind[10]` and `vind[11]`, and the numerical values for those non-zeros can be found in `vval[10]` and `vval[11]`.

vind: Constraint indices associated with non-zero values. See the description of the `vbeg` argument for more information.

vval: Numerical values associated with constraint matrix non-zeros. See the description of the `vbeg` argument for more information.

obj: Objective coefficients for the new variables. This argument can be `NULL`, in which case the objective coefficients are set to `0.0`.

lb: Lower bounds for the new variables. This argument can be `NULL`, in which case all variables get lower bounds of `0.0`.

ub: Upper bounds for the new variables. This argument can be `NULL`, in which case all variables get infinite upper bounds.

vtype: Types for the variables. Options are `GRB_CONTINUOUS`, `GRB_BINARY`, `GRB_INTEGER`, `GRB_SEMICONT`, or `GRB_SEMIINT`. This argument can be `NULL`, in which case all variables are assumed to be continuous.

varnames: Names for the new variables. This argument can be `NULL`, in which case all variables are given default names.

GRBXchgcoeffs

```
int  GRBXchgcoeffs (  GRBmodel  *model,
                      size_t    numchgs,
                      int       *cind,
                      int       *vind,
                      double    *val )
```

The `size_t` version of GRBchgcoeffs. The argument that counts non-zero values is of type `size_t` in this version to support models with more than 2 billion non-zero values.

Change a set of constraint matrix coefficients. This routine can be used to set a non-zero coefficient to zero, to create a non-zero coefficient where the coefficient is currently zero, or to change an existing non-zero coefficient to a new non-zero value. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the changes won't actually be integrated into the model until you update the model (using GRBupdatemodel), optimize the model (using GRBoptimize), or write the model to disk (using GRBwrite).

Return value:

A non-zero return value indicates that a problem occurred while performing the modification. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to modify.

`numchgs`: The number of coefficients to modify.

`cind`: Constraint indices for the coefficients to modify.

`vind`: Variable indices for the coefficients to modify.

`val`: The new values for the coefficients. For example, if `cind[0] = 1`, `vind[0] = 3`, and `val[0] = 2.0`, then the coefficient in constraint 1 associated with variable 3 would be changed to 2.0.

Example usage:

```
int cind[] = {0, 1};
int vind[] = {0, 0};
double val[] = {1.0, 1.0};
error = GRBXchgcoeffs(model, 2, cind, vind, val);
```

GRBXloadmodel

```
int  GRBXloadmodel (  GRBenv      *env,
                      GRBmodel    **modelP,
                      const char  *Pname,
                      int         numvars,
                      int         numconstrs,
                      int         objsense,
                      double      objcon,
                      double      *obj,
                      char        *sense,
                      double      *rhs,
                      size_t      *vbeg,
                      int         *vlen,
                      int         *vind,
                      double      *vval,
                      double      *lb,
                      double      *ub,
                      char        *vtype,
                      const char  **varnames,
                      const char  **constrnames )
```

The `size_t` version of GRBloadmodel. The argument that counts non-zero values is of type `size_t` in this version to support models with more than 2 billion non-zero values.

Create a new optimization model, using the provided arguments to initialize the model data (objective function, variable bounds, constraint matrix, etc.). The model is then ready for optimization, or for modification (e.g., addition of variables or constraints, changes to variable types or bounds, etc.).

Return value:

A non-zero return value indicates that a problem occurred while creating the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment in which the new model should be created. Note that the new model gets a copy of this environment, so subsequent modifications to the original environment (e.g., parameter changes) won't affect the new model. Use GRBgetenv to modify the environment associated with a model.

modelP: The location in which the pointer to the newly created model should be placed.

Pname: The name of the model.

numvars: The number of variables in the model.

numconstrs: The number of constraints in the model.

objsense: The sense of the objective function. Allowed values are 1 (minimization) or -1 (maximization).

objcon: Constant objective offset.

obj: Objective coefficients for the new variables. This argument can be `NULL`, in which case the objective coefficients are set to `0.0`.

sense: The senses of the new constraints. Options are `'='` (equal), `'<'` (less-than-or-equal), or `'>'` (greater-than-or-equal). You can also use constants `GRB_EQUAL`, `GRB_LESS_EQUAL`, or `GRB_GREATER_EQUAL`.

rhs: Right-hand side values for the new constraints. This argument can be `NULL`, in which case the right-hand side values are set to `0.0`.

vbeg: Constraint matrix non-zero values are passed into this routine in Compressed Sparse Column (CSC) format. Each column in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the constraint index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each variable in the model has a `vbeg` and `vlen` value, indicating the start position of the non-zeros for that variable in the `vind` and `vval` arrays, and the number of non-zero values for that variable, respectively. Thus, for example, if `vbeg[2] = 10` and `vlen[2] = 2`, that would indicate that variable 2 has two non-zero values associated with it. Their constraint indices can be found in `vind[10]` and `vind[11]`, and the numerical values for those non-zeros can be found in `vval[10]` and `vval[11]`. Note that the columns of the matrix must be ordered from first to last, implying that the values in `vbeg` must be non-decreasing.

vlen: Number of constraint matrix non-zero values associated with each variable. See the description of the `vbeg` argument for more information.

vind: Constraint indices associated with non-zero values. See the description of the `vbeg` argument for more information.

vval: Numerical values associated with constraint matrix non-zeros. See the description of the `vbeg` argument for more information.

lb: Lower bounds for the new variables. This argument can be `NULL`, in which case all variables get lower bounds of `0.0`.

ub: Upper bounds for the new variables. This argument can be `NULL`, in which case all variables get infinite upper bounds.

vtype: Types for the variables. Options are `GRB_CONTINUOUS`, `GRB_BINARY`, `GRB_INTEGER`,

GRB_SEMICONT, or GRB_SEMIINT. This argument can be NULL, in which case all variables are assumed to be continuous.

varnames: Names for the new variables. This argument can be NULL, in which case all variables are given default names.

constrnames: Names for the new constraints. This argument can be NULL, in which case all constraints are given default names.

Important notes:

We recommend that you build a model one constraint or one variable at a time, using GRBaddconstr or GRBaddvar, rather than using this routine to load the entire constraint matrix at once. It is much simpler, less error prone, and it introduces no significant overhead.

Example usage:

```
/* maximize    x +   y + 2 z
   subject to  x + 2 y + 3 z <= 4
               x +   y       >= 1
   x, y, z binary */

int    vars    = 3;
int    constrs = 2;
size_t vbeg[]  = {0, 2, 4};
int    vlen[]  = {2, 2, 1};
int    vind[]  = {0, 1, 0, 1, 0};
double vval[]  = {1.0, 1.0, 2.0, 1.0, 3.0};
double obj[]   = {1.0, 1.0, 2.0};
char   sense[] = {GRB_LESS_EQUAL, GRB_GREATER_EQUAL};
double rhs[]   = {4.0, 1.0};
char   vtype[] = {GRB_BINARY, GRB_BINARY, GRB_BINARY};

error = GRBXloadmodel(env, &model, "example", vars, constrs, -1, 0.0,
                      obj, sense, rhs, vbeg, vlen, vind, vval,
                      NULL, NULL, vtype, NULL, NULL);
```

3.3 Model Solution

GRBoptimize

```
int  GRBoptimize (  GRBmodel  *model )
```

Optimize a model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion, this method will populate the solution related attributes of the model. See the Attributes section for more information on attributes.

Please consult this section for a discussion of some of the practical issues associated with solving a precisely defined mathematical model using finite-precision floating-point arithmetic.

Note that this routine will process all pending model modifications.

Return value:

A non-zero return value indicates that a problem occurred while optimizing the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to optimize. Note that this routine only reports whether the optimization ran into an error. Query the `Status` attribute to determine the result of the optimization (see the Attributes section for more information on querying attributes).

Example usage:

```
error = GRBoptimize(model);
```

GRBoptimizeasync

```
int  GRBoptimizeasync (  GRBmodel  *model )
```

Optimize a model asynchronously. This routine returns immediately. Your program can perform other computations while optimization proceeds in the background. To check the state of the asynchronous optimization, query the Status attribute for the model. A value of `IN_PROGRESS` indicates that the optimization has not yet completed. When you are done with your foreground tasks, you must call GRBsync to sync your foreground program with the asynchronous optimization task.

Note that the set of Gurobi calls that you are allowed to make while optimization is running in the background is severely limited. Specifically, you can only perform attribute queries, and only for a few attributes (listed below). Any other calls on the running model, *or on any other models that were built within the same Gurobi environment*, will fail with error code `OPTIMIZATION_IN_PROGRESS`.

Note that there are no such restrictions on models built in other environments. Thus, for example, you could create multiple environments, and then have a single foreground program launch multiple simultaneous asynchronous optimizations, each in its own environment.

As already noted, you are allowed to query the value of the `Status` attribute while an asynchronous optimization is in progress. The other attributes that can be queried are: ObjVal, ObjBound, IterCount, NodeCount, and BarIterCount. In each case, the returned value reflects progress in the optimization to that point. Any attempt to query the value of an attribute not on this list will return an `OPTIMIZATION_IN_PROGRESS` error.

Return value:

A non-zero return value indicates that a problem occurred while optimizing the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to optimize. Note that this routine only reports whether launching the asynchronous job ran into an error. Query the `Status` attribute to determine the result of the optimization (see the Attributes section for more information on querying attributes). The return value of GRBsync indicates whether the background optimization ran into an error.

Example usage:

```
error = GRBoptimizeasync(model);

/* ... perform other compute-intensive tasks... */

error = GRBsync(model);
```

GRBpresolvemodel

```
int  GRBpresolvemodel (  GRBmodel  *model,
                         GRBmodel  **presolvedP )
```

Perform presolve on a model.

Please note that the presolved model computed by this function may be different from the presolved model computed when optimizing the model.

Return value:

A non-zero return value indicates that a problem occurred while presolving the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to be presolved.

`presolveP`: The location in which the pointer to the presolved model should be placed.

Example usage:

```
error = GRBpresolvemodel(model, &presolvedP);
```

GRBcomputeIIS

```
int  GRBcomputeIIS (  GRBmodel  *model )
```

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the smallest one; there may exist others with fewer constraints or bounds.

IIS results are returned in a number of attributes: IISConstr, IISLB, IISUB, IISSOS, IISQConstr, and IISGenConstr. Each indicates whether the corresponding model element is a member of the computed IIS.

Note that for models with general function constraints, piecewise-linear approximation of the constraints may cause unreliable IIS results.

The IIS log provides information about the progress of the algorithm, including a guess at the eventual IIS size.

If an IIS computation is interrupted before completion, Gurobi will return the smallest infeasible subsystem found to that point.

The IISConstrForce, IISLBForce, IISUBForce, IISSOSForce, IISQConstrForce, and IISGenConstrForce attributes allow you mark model elements to either include or exclude from the computed IIS. Setting the attribute to 1 forces the corresponding element into the IIS, setting it to 0 forces it out of the IIS, and setting it to -1 allows the algorithm to decide.

To give an example of when these attributes might be useful, consider the case where an initial model is known to be feasible, but it becomes infeasible after adding constraints or tightening bounds. If you are only interested in knowing which of the changes caused the infeasibility, you can force the unmodified bounds and constraints into the IIS. That allows the IIS algorithm to focus exclusively on the new constraints, which will often be substantially faster.

Note that setting any of the `Force` attributes to 0 may make the resulting subsystem feasible, which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

This routine populates the IISConstr, IISGenConstr, IISQConstr, IISSOS, IISLB, and IISUB attributes. You can also obtain information about the results of the IIS computation by writing a `.ilp` format file (see GRBwrite). This file contains only the IIS from the original model.

Use the IISMethod parameter to adjust the behavior of the IIS algorithm.

Note that this routine can be used to compute IISs for both continuous and MIP models.

Return value:

A non-zero return value indicates that a problem occurred while computing the IIS. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The infeasible model. This routine will return an error if the input model is feasible.

Important note:

This routine only reports whether the computation ran into an error. Query the IISConstr, IISGenConstr, IISQConstr, IISSOS, IISLB, or IISUB attributes to determine the result of the computation (see the Attributes section for more information on querying attributes).

Example usage:

```
error = GRBcomputeIIS(model);
```

GRBfeasrelax

```
int  GRBfeasrelax (  GRBmodel  *model,
                     int       relaxobjtype,
                     int       minrelax,
                     double    *lbpen,
                     double    *ubpen,
                     double    *rhspen,
                     double    *feasobjP )
```

Modifies the input model to create a feasibility relaxation. Note that you need to call GRBoptimize on the result to compute the actual relaxed solution.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This routine provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost per unit violation in the lower bounds, upper bounds, and linear constraints, respectively.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the coefficients on the squares of the lower bound, upper bound, and linear constraint violations, respectively.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the weighted count of bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost of violating a lower bound, upper bound, and linear constraint, respectively.

To give an example, a violation of 2.0 on constraint `i` would contribute `2*rhspen[i]` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2*2*rhspen[i]` for `relaxobjtype=1`, and it would contribute `rhspen[i]` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=0`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=1`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `GRBfeasrelax` must solve an optimization problem to find the minimum possible relaxation for `minrelax=1`, which can be quite expensive.

In all cases, you can specify a penalty of `GRB_INFINITY` to indicate that a specific bound or linear constraint may not be violated.

Note that this is a destructive routine: it modifies the model passed to it. If you don't want to modify your original model, use GRBcopymodel to create a copy before calling this routine.

Return value:

A non-zero return value indicates that a problem occurred while computing the feasibility relaxation. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The original (infeasible) model. The model is modified by this routine.

`relaxobjtype`: The cost function used when finding the minimum cost relaxation.

`minrelax`: The type of feasibility relaxation to perform.

lbpen: The penalty associated with violating a lower bound. Can be NULL, in which case no lower bound violations are allowed.

ubpen: The penalty associated with violating an upper bound. Can be NULL, in which case no upper bound violations are allowed.

rhspen: The penalty associated with violating a linear constraint. Can be NULL, in which case no constraint violations are allowed.

feasobjP: When minrelax=1, this returns the objective value for the minimum cost relaxation.

Example usage:

```
double penalties[];
error = GRBfeasrelax(model, 0, 0, NULL, NULL, penalties, NULL);
error = GRBoptimize(model);
```

GRBfixmodel

```
int  GRBfixmodel (  GRBmodel  *model,
                    GRBmodel  **fixedP )
```

Create the fixed model associated with a MIP model. The MIP model must have a solution loaded (e.g., after a call to GRBoptimize). In the fixed model, each integer variable is fixed to the value that variable takes in the MIP solution. In addition, continuous variables may be fixed to satisfy SOS or general constraints. The result is a model without any integrality constraints, SOS constraints, or general constraints.

Note that, while the fixed problem is always a continuous model, it may contain a non-convex quadratic objective or non-convex quadratic constraints. As a result, it may still be solved using the MIP algorithm.

Return value:

A non-zero return value indicates that a problem occurred while creating the fixed model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The MIP model (with a solution loaded).

fixedP: The computed fixed model.

Example usage:

```
GRBmodel *fixed;
error = GRBfixmodel(model, &fixed);
```

GRBreset

```
int  GRBreset (  GRBmodel  *model,
                 int       clearall )
```

Reset the model to an unsolved state, discarding any previously computed solution information.

Return value:

A non-zero return value indicates that a problem occurred while resetting the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to reset.

`clearall`: A value of 1 discards additional information that affects the solution process but not the actual model (currently MIP starts, variable hints, branching priorities, lazy flags, and partition information). Pass 0 to just discard the solution.

Example usage:

```
error = GRBreset(model, 0);
```

GRBsync

```
int  GRBsync ( GRBmodel *model )
```

Wait for a previous asynchronous optimization call to complete.

Calling GRBoptimizeasync returns control to the calling routine immediately. The caller can perform other computations while optimization proceeds, and can check on the progress of the optimization by querying various model attributes. The `GRBsync` call forces the calling program to wait until the asynchronous optimization completes. You *must* call `GRBsync` before the corresponding model is freed.

The `GRBsync` call returns a non-zero error code if the optimization itself ran into any problems. In other words, error codes returned by this method are those that `GRBoptimize` itself would have returned, had the original method not been asynchronous.

Note that you need to call `GRBsync` even if you know that the asynchronous optimization has already completed.

Return value:

A non-zero return value indicates that a problem occurred while solving the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that is currently being solved.

Example usage:

```
error = GRBoptimizeasync(model);

/* ... perform other compute-intensive tasks... */

error = GRBsync(model);
```

3.4 Model Queries

While most model related queries are handled through the attribute interface, a few fall outside of that interface. These are described here.

GRBgetcoeff

```
int  GRBgetcoeff (  GRBmodel  *model,
                    int       constrind,
                    int       varind,
                    double    *valP )
```

Retrieve a single constraint matrix coefficient.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the coefficient. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the coefficient should be retrieved.

constrind: The constraint index for the desired coefficient.

varind: The variable index for the desired coefficient.

valP: The location in which the requested matrix coefficient should be placed.

Example usage:

```
double A12;
error = GRBgetcoeff(model, 1, 2, &A12);
```

GRBgetconstrbyname

```
int  GRBgetconstrbyname (  GRBmodel     *model,
                           const char   *name,
                           int          *constrnumP )
```

Retrieves a linear constraint from its name. If multiple linear constraints have the same name, this routine chooses one arbitrarily.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the linear constraint should be retrieved.

name: The name of the desired linear constraint.

constrnumP: Constraint number for a linear constraint with the indicated name. Returns -1 if no matching name is found.

GRBgetconstrs

```
int  GRBgetconstrs ( GRBmodel  *model,
                     int       *numnzP,
                     int       *cbeg,
                     int       *cind,
                     double    *cval,
                     int       start,
                     int       len )
```

Retrieve the non-zeros for a set of linear constraints from the constraint matrix. Typical usage is to call this routine twice. In the first call, you specify the requested set of constraints, with `NULL` values for `cbeg`, `cind`, and `cval`. The routine returns the number of non-zero values for the specified constraint range in `numnzP`. That allows you to make certain that `cind` and `cval` are of sufficient size to hold the result of the second call.

If your constraint matrix may contain more than 2 billion non-zero values, you should consider using the GRBXgetconstrs variant of this routine.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the constraint coefficients. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model from which the linear constraints should be retrieved.

`numnzP`: The number of non-zero values retrieved.

`cbeg`: Constraint matrix non-zero values are returned in Compressed Sparse Row (CSR) format. Each constraint in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the variable index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each constraint has an associated `cbeg` value, indicating the start position of the non-zeros for that constraint in the `cind` and `cval` arrays. The non-zeros for constraint `i` immediately follow those for constraint `i-1` in `cind` and `cval`. Thus, `cbeg[i]` indicates both the index of the first non-zero in constraint `i` and the end of the non-zeros for constraint `i-1`. For example, consider the case where `cbeg[2] = 10` and `cbeg[3] = 12`. This would indicate that constraint 2 has two non-zero values associated with it. Their variable indices can be found in `cind[10]` and `cind[11]`, and the numerical values for those non-zeros can be found in `cval[10]` and `cval[11]`.

`cind`: Variable indices associated with non-zero values. See the description of the `cbeg` argument for more information.

`cval`: Numerical values associated with constraint matrix non-zeros. See the description of the `cbeg` argument for more information.

`start`: The index of the first linear constraint to retrieve.

`len`: The number of linear constraints to retrieve.

GRBgetenv

```
GRBenv *  GRBgetenv ( GRBmodel  *model )
```

Retrieve the environment associated with a model.

Note that this environment is a model environment, not the original environment on which the model was created. See Algorithmic parameters for more information.

Return value:

The environment associated with the model. A `NULL` return value indicates that there was a problem retrieving the environment.

Arguments:

`model`: The model from which the environment should be retrieved.

Example usage:

```
GRBenv *env = GRBgetenv(model);
```

GRBgetgenconstrMax

```
int  GRBgetgenconstrMax (  GRBmodel  *model,
                           int       id,
                           int       *resvarP,
                           int       *nvarsP,
                           int       *vars,
                           double    *constantP )
```

Retrieve the data associated with a general constraint of type MAX. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `NULL` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `nvarsP`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also GRBaddgenconstrMax for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`resvarP`: The variable index associated with the resultant variable of the constraint.

`nvarsP`: The number of operand variables of the constraint.

`vars`: An array to store the variable indices associated with the variable operands of the constraint.

`constantP`: The additional constant operand of the constraint.

Example usage:

```
int type;
int resvar;
int nvars;
int *vars;
```

```
double constant;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_MAX) {
  error = GRBgetgenconstrMax(model, 3, &resvar, &nvars, NULL, &constant);
  /* ...allocate vars to hold 'nvars' values... */
  error = GRBgetgenconstrMax(model, 3, NULL, NULL, vars, NULL);
}
```

GRBgetgenconstrMin

```
int  GRBgetgenconstrMin (  GRBmodel  *model,
                           int       id,
                           int       *resvarP,
                           int       *nvarsP,
                           int       *vars,
                           double    *constantP )
```

Retrieve the data associated with a general constraint of type MIN. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `NULL` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `nvarsP`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also GRBaddgenconstrMin for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`resvarP`: The variable index associated with the resultant variable of the constraint.

`nvarsP`: The number of operand variables of the constraint.

`vars`: An array to store the variable indices associated with the variable operands of the constraint.

`constantP`: The additional constant operand of the constraint.

Example usage:

```
int type;
int resvar;
int nvars;
int *vars;
double constant;
```

```
error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_MIN) {
  error = GRBgetgenconstrMin(model, 3, &resvar, &nvars, NULL, &constant);
  /* ...allocate vars to hold 'nvars' values... */
  error = GRBgetgenconstrMin(model, 3, NULL, NULL, vars, NULL);
}
```

GRBgetgenconstrAbs

```
int  GRBgetgenconstrAbs (  GRBmodel  *model,
                           int       id,
                           int       *resvarP,
                           int       *argvarP )
```

Retrieve the data associated with a general constraint of type ABS. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrAbs for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`resvarP`: The variable index associated with the resultant variable of the constraint.

`argvarP`: The variable index associated with the argument variable of the constraint.

Example usage:

```
int type;
int resvar;
int argvar;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_ABS) {
  error = GRBgetgenconstrAbs(model, 3, &resvar, &argvar);
}
```

GRBgetgenconstrAnd

```
int  GRBgetgenconstrAnd (  GRBmodel  *model,
                           int       id,
                           int       *resvarP,
                           int       *nvarsP,
                           int       *vars )
```

Retrieve the data associated with a general constraint of type AND. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `NULL` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `nvarsP`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also GRBaddgenconstrAnd for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`resvarP`: The variable index associated with the binary resultant variable of the constraint.

`nvarsP`: The number of binary operand variables of the constraint.

`vars`: An array to store the variable indices associated with the binary variable operands of the constraint.

Example usage:

```
int type;
int resvar;
int nvars;
int *vars;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_AND) {
  error = GRBgetgenconstrAnd(model, 3, &resvar, &nvars, NULL);
  /* ...allocate vars to hold 'nvars' values... */
  error = GRBgetgenconstrAnd(model, 3, NULL, NULL, vars);
}
```

GRBgetgenconstrOr

```
int  GRBgetgenconstrOr (  GRBmodel  *model,
                          int       id,
                          int       *resvarP,
                          int       *nvarsP,
                          int       *vars )
```

Retrieve the data associated with a general constraint of type OR. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `NULL` value for the `vars` argument. The routine returns the total number of

operand variables in the specified general constraint in `nvarsP`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also GRBaddgenconstrOr for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`resvarP`: The variable index associated with the binary resultant variable of the constraint.

`nvarsP`: The number of binary operand variables of the constraint.

`vars`: An array to store the variable indices associated with the binary variable operands of the constraint.

Example usage:

```
int type;
int resvar;
int nvars;
int *vars;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_OR) {
  error = GRBgetgenconstrOr(model, 3, &resvar, &nvars, NULL);
  /* ...allocate vars to hold 'nvars' values... */
  error = GRBgetgenconstrOr(model, 3, NULL, NULL, vars);
}
```

GRBgetgenconstrNorm

```
int  GRBgetgenconstrNorm (  GRBmodel  *model,
                            int       id,
                            int       *resvarP,
                            int       *nvarsP,
                            int       *vars,
                            double    *whichP )
```

Retrieve the data associated with a general constraint of type NORM. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `NULL` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `nvarsP`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also GRBaddgenconstrNorm for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`resvarP`: The variable index associated with the resultant variable of the constraint.

`nvarsP`: The number of operand variables of the constraint.

`vars`: An array to store the variable indices associated with the variable operands of the constraint.

`whichP`: Which norm is used. Options are 0, 1, 2, and GRB_INFINITY.

Example usage:

```
int type;
int resvar;
int nvars;
int *vars;
double which;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_NORM) {
  error = GRBgetgenconstrNorm(model, 3, &resvar, &nvars, NULL, &which);
  /* ...allocate vars to hold 'nvars' values... */
  error = GRBgetgenconstrNorm(model, 3, NULL, NULL, vars);
}
```

GRBgetgenconstrIndicator

```
int  GRBgetgenconstrIndicator (  GRBmodel  *model,
                                 int       id,
                                 int       *binvarP,
                                 int       *binvalP,
                                 int       *nvarsP,
                                 int       *ind,
                                 double    *val,
                                 char      *senseP,
                                 double    *rhsP )
```

Retrieve the data associated with a general constraint of type INDICATOR. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with `NULL` values for the `ind` and `val` arguments. The routine returns the total number of non-zero coefficients in the linear constraint associated with the specified indicator constraint in `nvarsP`. That allows you to make certain that the `ind` and `val` arrays are of sufficient size to hold the result of the second call.

See also GRBaddgenconstrIndicator for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`binvarP`: The variable index associated with the binary indicator variable.

`binvalP`: The value that the indicator variable has to take in order to trigger the linear constraint.

`nvarsP`: The number of non-zero coefficients in the linear constraint triggered by the indicator.

`ind`: An array to store the variable indices for non-zero values in the linear constraint.

`val`: An array to store the numerical values for non-zero values in the linear constraint.

`senseP`: Sense for the linear constraint. Options are `GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`.

`rhsP`: Right-hand side value for the linear constraint.

Example usage:

```
int type;
int binvar;
int binval:
int nvars;
int *ind;
double *val;
char sense;
double rhs;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_INDICATOR) {
  error = GRBgetgenconstrIndicator(model, 3, &binvar, &binval, &nvars,
                                   NULL, NULL, &sense, &rhs);
  /* ...allocate ind and val to hold 'nvars' values... */
  error = GRBgetgenconstrIndicator(model, 3, NULL, NULL, NULL,
                                   ind, val, NULL, NULL);
}
```

GRBgetgenconstrPWL

```
int  GRBgetgenconstrPWL (  GRBmodel  *model,
                           int       id,
                           int       *xvarP,
                           int       *yvarP,
                           int       *nptsP,
                           double    *xpts,
                           double    *ypts )
```

Retrieve the data associated with a general constraint of type PWL. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `NULL` value for the `xpts` and `ypts` arguments. The routine returns the length for the `xpts` and `ypts` arrays in `nptsP`. That allows you to make certain that the `xpts` and `ypts` arrays are of sufficient size to hold the result of the second call.

See also GRBaddgenconstrPWL for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`xvarP`: The index of variable x.

`yvarP`: The index of variable y.

`nptsP`: The number of points that define the piecewise-linear function.

`xpts`: The x values for the points that define the piecewise-linear function.

`ypts`: The y values for the points that define the piecewise-linear function.

Example usage:

```
int type;
int xvar;
int yvar;
int npts;
double *xpts;
double *ypts;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_PWL) {
  error = GRBgetgenconstrPWL(model, 3, &xvar, &yvar, &npts, NULL, NULL);
  /* ...allocate xpts and ypts arrays with length npts */
  error = GRBgetgenconstrPWL(model, 3, NULL, NULL, NULL, xpts, ypts);
}
```

GRBgetgenconstrPoly

```
int  GRBgetgenconstrPoly (  GRBmodel  *model,
                            int       id,
                            int       *xvarP,
                            int       *yvarP,
                            int       *plenP,
                            double    *p )
```

Retrieve the data associated with a general constraint of type POLY. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `NULL` value for the `p` argument. The routine returns the length of the `p` array in `plenP`. That allows you to make certain that the `p` array is of sufficient size to hold the result of the second call.

See also GRBaddgenconstrPoly for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`xvarP`: The index of variable x.

`yvarP`: The index of variable y.

`plenP`: Pointer to store the array length for p. If x^d is the highest power term, then $d+1$ will be returned.

`p`: The coefficients for polynomial function.

Example usage:

```
int type;
int xvar;
int yvar;
int plen;
double *p;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_POLY) {
  error = GRBgetgenconstrPoly(model, 3, &xvar, &yvar, &plen, NULL);
  /* ...allocate p array with length plen */
  error = GRBgetgenconstrPoly(model, 3, NULL, NULL, NULL, p);
}
```

GRBgetgenconstrExp

```
int  GRBgetgenconstrExp (  GRBmodel  *model,
                           int       id,
                           int       *xvarP,
                           int       *yvarP )
```

Retrieve the data associated with a general constraint of type EXP. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrExp for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`xvarP`: The index of variable x.

`yvarP`: The index of variable y.

Example usage:

```
int type;
int xvar;
int yvar;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_EXP) {
  error = GRBgetgenconstrExp(model, 3, &xvar, &yvar);
}
```

GRBgetgenconstrExpA

```
int  GRBgetgenconstrExpA (  GRBmodel  *model,
                            int       id,
                            int       *xvarP,
                            int       *yvarP,
                            double    *aP )
```

Retrieve the data associated with a general constraint of type EXPA. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrExpA for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`xvarP`: The index of variable x.

`yvarP`: The index of variable y.

`aP`: The base of the function.

Example usage:

```
int type;
int xvar;
int yvar;
double a;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_EXPA) {
  error = GRBgetgenconstrExpA(model, 3, &xvar, &yvar, &a);
}
```

GRBgetgenconstrLog

```
int   GRBgetgenconstrLog (  GRBmodel  *model,
                            int       id,
                            int       *xvarP,
                            int       *yvarP )
```

Retrieve the data associated with a general constraint of type LOG. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrLog for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`xvarP`: The index of variable x.

`yvarP`: The index of variable y.

Example usage:

```
int type;
int xvar;
int yvar;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
```

```
if (type == GRB_GENCONSTR_LOG) {
  error = GRBgetgenconstrLog(model, 3, &xvar, &yvar);
}
```

GRBgetgenconstrLogA

```
int  GRBgetgenconstrLogA (  GRBmodel  *model,
                            int       id,
                            int       *xvarP,
                            int       *yvarP,
                            double    *aP )
```

Retrieve the data associated with a general constraint of type LOGA. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrLogA for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`xvarP`: The index of variable x.

`yvarP`: The index of variable y.

`aP`: The base of the function.

Example usage:

```
int type;
int xvar;
int yvar;
double a;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_LOGA) {
  error = GRBgetgenconstrLogA(model, 3, &xvar, &yvar, &a);
}
```

GRBgetgenconstrLogistic

```
int  GRBgetgenconstrLogistic (  GRBmodel  *model,
                                int       id,
                                int       *xvarP,
                                int       *yvarP )
```

Retrieve the data associated with a general constraint of type LOGISTIC. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrLogistic for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

`xvarP`: The index of variable x.

`yvarP`: The index of variable y.

Example usage:

```
int type;
int xvar;
int yvar;
double a;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_LOGISTIC) {
  error = GRBgetgenconstrLogistic(model, 3, &xvar, &yvar);
}
```

GRBgetgenconstrPow

```
int  GRBgetgenconstrPow (  GRBmodel  *model,
                           int       id,
                           int       *xvarP,
                           int       *yvarP,
                           double    *aP )
```

Retrieve the data associated with a general constraint of type POW. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrPow for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that contains the desired general constraint.

`id`: The index of the general constraint to retrieve.

Note that any of the following arguments can be `NULL`.

xvarP: The index of variable x.

yvarP: The index of variable y.

aP: The exponent of the function.

Example usage:

```
int type;
int xvar;
int yvar;
double a;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_POW) {
  error = GRBgetgenconstrPow(model, 3, &xvar, &yvar, &a);
}
```

GRBgetgenconstrSin

```
int  GRBgetgenconstrSin (  GRBmodel  *model,
                           int       id,
                           int       *xvarP,
                           int       *yvarP )
```

Retrieve the data associated with a general constraint of type SIN. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrSin for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model that contains the desired general constraint.

id: The index of the general constraint to retrieve.

Note that any of the following arguments can be NULL.

xvarP: The index of variable x.

yvarP: The index of variable y.

Example usage:

```
int type;
int xvar;
int yvar;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_SIN) {
  error = GRBgetgenconstrSin(model, 3, &xvar, &yvar);
}
```

GRBgetgenconstrCos

```
int  GRBgetgenconstrCos (  GRBmodel  *model,
                           int       id,
                           int       *xvarP,
                           int       *yvarP )
```

Retrieve the data associated with a general constraint of type COS. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrCos for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model that contains the desired general constraint.

id: The index of the general constraint to retrieve.

Note that any of the following arguments can be NULL.

xvarP: The index of variable x.

yvarP: The index of variable y.

Example usage:

```
int type;
int xvar;
int yvar;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_COS) {
  error = GRBgetgenconstrCos(model, 3, &xvar, &yvar);
}
```

GRBgetgenconstrTan

```
int  GRBgetgenconstrTan (  GRBmodel  *model,
                           int       id,
                           int       *xvarP,
                           int       *yvarP )
```

Retrieve the data associated with a general constraint of type TAN. Calling this method for a general constraint of a different type leads to an error return code. You can query the GenConstrType attribute to determine the type of the general constraint.

See also GRBaddgenconstrTan for a description of the semantics of this general constraint type.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the general constraint data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model that contains the desired general constraint.

id: The index of the general constraint to retrieve.

Note that any of the following arguments can be NULL.

xvarP: The index of variable x.

yvarP: The index of variable y.

Example usage:

```
int type;
int xvar;
int yvar;

error = GRBgetintattrelement(model, GRB_INT_ATTR_GENCONSTRTYPE, 3, &type);
if (type == GRB_GENCONSTR_TAN) {
  error = GRBgetgenconstrTan(model, 3, &xvar, &yvar);
}
```

GRBgetjsonsolution

```
int  GRBgetjsonsolution (  GRBmodel  model,
                           char**    buffP )
```

After a call to optimize, this method returns the resulting solution and related model attributes as a JSON string. Please refer to the JSON solution format section for details.

Return value:

A non-zero return value indicates that there was a problem generating the JSON solution string. Refer to the Error Code table for a list of possible return values.

Arguments:

model: Model from which to query its current JSON solution string.

buffP: The location in which the pointer to the newly created JSON string should be placed.

Important note:

On Windows, the string returned in **buffP** is allocated in a different heap from the calling program. You must call **GRBfree** to free it.

GRBgetpwlobj

```
int  GRBgetpwlobj (  GRBmodel  *model,
                     int       var,
                     int       *npointsP,
                     double    *x,
                     double    *y )
```

Retrieve the piecewise-linear objective function for a variable. The x and y arguments must be large enough to hold the result. If either are NULL, then **npointsP** will contain the number of points in the function on return.

Refer to this discussion for additional information on what the values in x and y mean.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the piecewise-linear objective function. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the piecewise-linear objective function is being retrieved.

var: The variable whose objective function is being retrieved.

npointsP: The number of points that define the piecewise-linear function.

x: The x values for the points that define the piecewise-linear function. These will always be in non-decreasing order.

y: The y values for the points that define the piecewise-linear function.

Example usage:

```
double *x;
double *y;

error = GRBgetpwlobj(model, var, &npoints, NULL, NULL);
/* ...allocate x and y to hold 'npoints' values... */
error = GRBgetpwlobj(model, var, &npoints, x, y);
```

GRBgetq

```
int  GRBgetq (  GRBmodel  *model,
                int       *numqnzP,
                int       *qrow,
                int       *qcol,
                double    *qval )
```

Retrieve all quadratic objective terms. The **qrow**, **qcol**, and **qval** arguments must be large enough to hold the result. You can query the NumQNZs attribute to determine how many terms will be returned.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the quadratic objective terms. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the quadratic objective terms should be retrieved.

numqnzP: The number of quadratic objective terms retrieved.

qrow: Row indices associated with quadratic terms. A quadratic term is represented using three values: a pair of indices (stored in **qrow** and **qcol**), and a coefficient (stored in **qval**). The three argument arrays provide the corresponding values for each quadratic term. To give an example, to represent $2x_0^2 + x_0x_1 + x_1^2$, you would have ***numqnzP=3**, **qrow[] = {0, 0, 1}**, **qcol[] = {0, 1, 1}**, and **qval[] = {2.0, 1.0, 1.0}**.

qcol: Column indices associated with quadratic terms. See the description of the **qrow** argument for more information.

qval: Numerical values associated with quadratic terms. See the description of the **qrow** argument for more information.

Example usage:

```
int     qnz;
int    *qrow, *qcol;
double *qval;

error = GRBgetdblattr(model, GRB_DBL_ATTR_NUMQNZS, &qnz);
/* ...allocate qrow, qcol, qval to hold 'qnz' values... */
error = GRBgetq(model, &qnz, qrow, qcol, qval);
```

GRBgetqconstr

```
int  GRBgetqconstr (  GRBmodel  *model,
                      int       qconstr,
                      int       *numlnzP,
                      int       *lind,
                      double    *lval,
                      int       *numqnzP,
                      int       *qrow,
                      int       *qcol,
                      double    *qval )
```

Retrieve the linear and quadratic terms associated with a single quadratic constraint. Typical usage is to call this routine twice. In the first call, you specify the requested quadratic constraint, with `NULL` values for the array arguments. The routine returns the total number of linear and quadratic terms in the specified quadratic constraint in `numlnzP` and `numqnzP`, respectively. That allows you to make certain that `lind`, `lval`, `qrow`, `qcol`, and `qval` are of sufficient size to hold the result of the second call.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the quadratic constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model from which the quadratic constraint should be retrieved.

`qconstr`: The index of the requested quadratic constraint.

`numlnzP`: The number of linear terms retrieved for the requested quadratic constraint.

`lind`: Variable indices associated with linear terms.

`lval`: Numerical coefficients associated with linear terms.

`numqnzP`: The number of quadratic terms retrieved for the requested quadratic constraint.

`qrow`: Row indices associated with quadratic terms. A quadratic term is represented using three values: a pair of indices (stored in `qrow` and `qcol`), and a coefficient (stored in `qval`). The associated arguments arrays provide the corresponding values for each quadratic term. To give an example, if the requested quadratic constraint has quadratic terms $2x_0^2 + x_0x_1 + x_1^2$, this routine would return `*numqnzP=3`, `qrow[] = {0, 0, 1}`, `qcol[] = {0, 1, 1}`, and `qval[] = {2.0, 1.0, 1.0}`.

`qcol`: Column indices associated with quadratic terms. See the description of the `qrow` argument for more information.

qval: Numerical values associated with quadratic terms. See the description of the **qrow** argument for more information.

GRBgetqconstrbyname

```
int  GRBgetqconstrbyname (  GRBmodel     *model,
                            const char   *name,
                            int          *constrnumP )
```

Retrieves a quadratic constraint from its name. If multiple quadratic constraints have the same name, this routine chooses one arbitrarily.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the quadratic constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the quadratic constraint should be retrieved.

name: The name of the desired quadratic constraint.

constrnumP: Constraint number for a quadratic constraint with the indicated name. Returns -1 if no matching name is found.

GRBgetsos

```
int  GRBgetsos (  GRBmodel  *model,
                  int       *nummembersP,
                  int       *sostype,
                  int       *beg,
                  int       *ind,
                  double    *weight,
                  int       start,
                  int       len )
```

Retrieve the members and weights of a set of SOS constraints. Typical usage is to call this routine twice. In the first call, you specify the requested SOS constraints, with **NULL** values for **ind** and **weight**. The routine returns the total number of members for the specified SOS constraints in **nummembersP**. That allows you to make certain that **ind** and **weight** are of sufficient size to hold the result of the second call.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the SOS members. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the SOS constraints should be retrieved.

nummembersP: The total number of SOS members retrieved.

sostype: The types of the SOS constraints. Possible values are **GRB_SOS_TYPE1** or **GRB_SOS_TYPE2**

beg: SOS constraints are returned in Compressed Sparse Row (CSR) format. Each SOS constraint in the model is represented as a list of index-value pairs, where each index

entry provides the variable index for an SOS member, and each value entry provides the corresponding SOS constraint weight. Each SOS constraint has an associated **beg** value, indicating the start position of the members of that constraint in the **ind** and **weight** arrays. The members for SOS constraint **i** immediately follow those for constraint **i-1** in **ind** and **weight**. Thus, **beg[i]** indicates both the index of the first member of SOS constraint **i** and the end of the member list for SOS constraint **i-1**. For example, consider the case where **beg[2] = 10** and **beg[3] = 12**. This would indicate that SOS constraint 2 has two members. Their variable indices can be found in **ind[10]** and **ind[11]**, and their SOS weights can be found in **weight[10]** and **weight[11]**.

ind: Variable indices associated with SOS members. See the description of the **beg** argument for more information.

weight: Weights associated with SOS members. See the description of the **beg** argument for more information.

start: The index of the first SOS constraint to retrieve.

len: The number of SOS constraints to retrieve.

GRBgetvarbyname

```
int  GRBgetvarbyname (  GRBmodel    *model,
                        const char  *name,
                        int         *varnumP )
```

Retrieves a variable from its name. If multiple variables have the same name, this routine chooses one arbitrarily.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the variable. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the variable should be retrieved.

name: The name of the desired variable.

varnumP: Variable number for a variable with the indicated name. Returns -1 if no matching name is found.

GRBgetvars

```
int  GRBgetvars (  GRBmodel  *model,
                   int       *numnzP,
                   int       *vbeg,
                   int       *vind,
                   double    *vval,
                   int       start,
                   int       len )
```

Retrieve the non-zeros for a set of variables from the constraint matrix. Typical usage is to call this routine twice. In the first call, you specify the requested set of variables, with **NULL** values for **vbeg**, **vind**, and **vval**. The routine returns the number of non-zero values for the specified variables

in numnzP. That allows you to make certain that vind and vval are of sufficient size to hold the result of the second call.

If your constraint matrix may contain more than 2 billion non-zero values, you should consider using the GRBXgetvars variant of this routine.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the variable coefficients. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the variables should be retrieved.

numnzP: The number of non-zero values retrieved.

vbeg: Constraint matrix non-zero values are returned in Compressed Sparse Column (CSC) format by this routine. Each column in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the constraint index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each variable has an associated vbeg value, indicating the start position of the non-zeros for that constraint in the vind and vval arrays. The non-zeros for variable i immediately follow those for variable i-1 in vind and vval. Thus, vbeg[i] indicates both the index of the first non-zero in variable i and the end of the non-zeros for variable i-1. For example, consider the case where vbeg[2] = 10 and vbeg[3] = 12. This would indicate that variable 2 has two non-zero values associated with it. Their constraint indices can be found in vind[10] and vind[11], and the numerical values for those non-zeros can be found in vval[10] and vval[11].

vind: Constraint indices associated with non-zero values. See the description of the vbeg argument for more information.

vval: Numerical values associated with constraint matrix non-zeros. See the description of the vbeg argument for more information.

start: The index of the first variable to retrieve.

len: The number of variables to retrieve.

GRBsinglescenariomodel

```
int  GRBsinglescenariomodel (  GRBmodel  *model,
                               GRBmodel  **singlescenarioP )
```

Capture a single scenario from a multi-scenario model. Use the ScenarioNumber parameter to indicate which scenario to capture.

Return value:

A non-zero return value indicates that a problem occurred while extracting the single-scenario model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model from which the scenario should be extracted.

singlescenarioP: The location in which the pointer to the requested single-scenario model should be placed.

GRBXgetconstrs

```
int  GRBXgetconstrs (  GRBmodel  *model,
                       size_t    *numnzP,
                       size_t    *cbeg,
                       int       *cind,
                       double    *cval,
                       int       start,
                       int       len )
```

The `size_t` version of GRBgetconstrs. The two arguments that count non-zero values are of type `size_t` in this version to support models with more than 2 billion non-zero values.

Retrieve the non-zeros for a set of linear constraints from the constraint matrix. Typical usage is to call this routine twice. In the first call, you specify the requested set of constraints, with `NULL` values for `cbeg`, `cind`, and `cval`. The routine returns the number of non-zero values for the specified constraint range in `numnzP`. That allows you to make certain that `cind` and `cval` are of sufficient size to hold the result of the second call.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the constraint coefficients. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model from which the constraints should be retrieved.

`numnzP`: The number of non-zero values retrieved.

`cbeg`: Constraint matrix non-zero values are returned in Compressed Sparse Row (CSR) format. Each constraint in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the variable index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each constraint has an associated `cbeg` value, indicating the start position of the non-zeros for that constraint in the `cind` and `cval` arrays. The non-zeros for constraint `i` immediately follow those for constraint `i-1` in `cind` and `cval`. Thus, `cbeg[i]` indicates both the index of the first non-zero in constraint `i` and the end of the non-zeros for constraint `i-1`. For example, consider the case where `cbeg[2] = 10` and `cbeg[3] = 12`. This would indicate that constraint 2 has two non-zero values associated with it. Their variable indices can be found in `cind[10]` and `cind[11]`, and the numerical values for those non-zeros can be found in `cval[10]` and `cval[11]`.

`cind`: Variable indices associated with non-zero values. See the description of the `cbeg` argument for more information.

`cval`: Numerical values associated with constraint matrix non-zeros. See the description of the `cbeg` argument for more information.

`start`: The index of the first constraint to retrieve.

`len`: The number of constraints to retrieve.

GRBXgetvars

```
int  GRBXgetvars (  GRBmodel  *model,
                    size_t    *numnzP,
                    size_t    *vbeg,
                    int       *vind,
                    double    *vval,
                    int       start,
                    int       len )
```

The `size_t` version of GRBgetvars. The two arguments that count non-zero values are of type `size_t` in this version to support models with more than 2 billion non-zero values.

Retrieve the non-zeros for a set of variables from the constraint matrix. Typical usage is to call this routine twice. In the first call, you specify the requested set of variables, with `NULL` values for `vbeg`, `vind`, and `vval`. The routine returns the number of non-zero values for the specified variables in `numnzP`. That allows you to make certain that `vind` and `vval` are of sufficient size to hold the result of the second call.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the variable coefficients. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model from which the variables should be retrieved.

`numnzP`: The number of non-zero values retrieved.

`vbeg`: Constraint matrix non-zero values are returned in Compressed Sparse Column (CSC) format by this routine. Each column in the constraint matrix is represented as a list of index-value pairs, where each index entry provides the constraint index for a non-zero coefficient, and each value entry provides the corresponding non-zero value. Each variable has an associated `vbeg` value, indicating the start position of the non-zeros for that constraint in the `vind` and `vval` arrays. The non-zeros for variable `i` immediately follow those for variable `i-1` in `vind` and `vval`. Thus, `vbeg[i]` indicates both the index of the first non-zero in variable `i` and the end of the non-zeros for variable `i-1`. For example, consider the case where `vbeg[2] = 10` and `vbeg[3] = 12`. This would indicate that variable 2 has two non-zero values associated with it. Their constraint indices can be found in `vind[10]` and `vind[11]`, and the numerical values for those non-zeros can be found in `vval[10]` and `vval[11]`.

`vind`: Constraint indices associated with non-zero values. See the description of the `vbeg` argument for more information.

`vval`: Numerical values associated with constraint matrix non-zeros. See the description of the `vbeg` argument for more information.

`start`: The index of the first variable to retrieve.

`len`: The number of variables to retrieve.

3.5 Input-Output

GRBreadmodel

```
int  GRBreadmodel (  GRBenv      *env,
                     const char  *filename,
                     GRBmodel    **modelP )
```

Read a model from a file.

Return value:

A non-zero return value indicates that a problem occurred while reading the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`env`: The environment in which to load the new model. This should come from a previous call to GRBloadenv.

`filename`: The path to the file to be read. Note that the type of the file is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, `.rlp`, `.dua`, `.dlp`, `.ilp`, or `.opb`. The files can be compressed, so additional suffixes of `.zip`, `.gz`, `.bz2`, or `.7z` are accepted.

`modelP`: The location in which the pointer to the model should be placed.

Example usage:

```
GRBmodel *model;
error = GRBreadmodel(env, "/tmp/model.mps.bz2", &model);
```

GRBread

```
int  GRBread (  GRBmodel    *model,
                const char  *filename )
```

This method is the general entry point for importing data from a file into a model. It can be used to read basis files for continuous models, start vectors for MIP models, or parameter settings. The type of data read is determined by the file suffix. File formats are described in the File Format section.

Note that reading a file does **not** process all pending model modifications. These modifications can be processed by calling GRBupdatemodel.

Note also that this is **not** the method to use if you want to read a new model from a file. For that, use the GRBreadmodel routine.

Return value:

A non-zero return value indicates that a problem occurred while reading the file. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model that will receive the information read from the file.

`filename`: The path to the file to be read. The suffix on the file must be either `.mst` or `.sol` for a MIP start file, `.hnt` for a MIP hint file, `.ord` for a priority order file, `.bas` for a basis file, or `.prm` for a parameter file, The suffix may optionally be followed by `.zip`, `.gz`, `.bz2`, or `.7z`.

Example usage:

```
error = GRBread(model, "/tmp/model.mst.bz2");
```

GRBwrite

```
int  GRBwrite (  GRBmodel    *model,
                 const char  *filename )
```

This routine is the general entry point for writing optimization data to a file. It can be used to write optimization models, solutions vectors, basis vectors, start vectors, or parameter settings. The type of data written is determined by the file suffix. File formats are described in the File Format section.

Note that writing a model to a file will process all pending model modifications. This is also true when writing other model information such as solutions, bases, etc.

Note also that when you write a Gurobi parameter file (PRM), both integer or double parameters not at their default value will be saved, but no string parameter will be saved into the file.

Return value:

A non-zero return value indicates that a problem occurred while writing the file. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model containing the data to be written.

`filename`: The name of the file to be written. The file type is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, or `.rlp` for writing the model itself, `.dua` or `.dlp` for writing the dualized model (only pure LP), `.ilp` for writing just the IIS associated with an infeasible model (see GRBcomputeIIS for further information), `.sol` for writing the solution selected by the SolutionNumber parameter, `.mst` for writing a start vector, `.hnt` for writing a hint file, `.bas` for writing an LP basis, `.prm` for writing modified parameter settings, `.attr` for writing model attributes, or `.json` for writing solution information in JSON format. If your system has compression utilities installed (e.g., `7z` or `zip` for Windows, and `gzip`, `bzip2`, or `unzip` for Linux or macOS), then the files can be compressed, so additional suffixes of `.gz`, `.bz2`, or `.7z` are accepted.

Example usage:

```
error = GRBwrite(model, "/tmp/model.rlp.gz");
```

3.6 Attribute Management

GRBgetattrinfo

```
int  GRBgetattrinfo (  GRBmodel     *model,
                       const char   *attrname,
                       int          *datatypeP,
                       int          *attrtypeP,
                       int          *settableP )
```

Obtain information about an attribute.

Return value:

A non-zero return value indicates that a problem occurred while obtaining information about the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an attribute. Available attributes are listed and described in the Attributes section of this document.

datatypeP: On completion, the integer pointed to by this argument will indicate the data type of the attribute. Possible types are char (0), int (1), double (2), or string(3). This argument can be NULL.

attrtypeP: On completion, the integer pointed to by this argument will indicate the type of the attribute. Possible types are model attribute (0), variable attribute (1), linear constraint attribute (2), (3) SOS constraint attribute, (4) quadratic constraint attribute, or (5) general constraint attribute. This argument can be NULL.

settableP: On completion, the integer pointed to by this argument will indicate whether the attribute can be set (1) or not (0). This argument can be NULL.

Example usage:

```
int datatype, attrtype, settable;
error = GRBgetattrinfo(model, "ModelName", &datatype, &attrtype, &settable);
```

GRBgetintattr

```
int  GRBgetintattr (  GRBmodel     *model,
                      const char   *attrname,
                      int          *valueP )
```

Query the value of an integer-valued model attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an integer-valued model attribute. Available attributes are listed and described in the Attributes section of this document.

valueP: The location in which the current value of the requested attribute should be placed.

Important note:

Note that this method should be used for scalar attributes only (i.e., model attributes). To query a single element of an array attribute, use GRBgetintattrelement instead.

Example usage:

```
error = GRBgetintattr(model, "NumBinVars", &numbin);
```

GRBsetintattr

```
int  GRBsetintattr (  GRBmodel    *model,
                      const char  *attrname,
                      int         newvalue )
```

Set the value of an integer-valued model attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an integer-valued model attribute. Available attributes are listed and described in the Attributes section of this document.

newvalue: The desired new value of this attribute.

Important note:

Note that this method should be used for scalar attributes only (i.e., model attributes). To modify a single element of an array attribute, use GRBsetintattrelement instead.

Example usage:

```
error = GRBsetintattr(model, "ModelSense", -1);
```

GRBgetintattrelement

```
int  GRBgetintattrelement (  GRBmodel    *model,
                             const char  *attrname,
                             int         element,
                             int         *valueP )
```

Query a single value from an integer-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an integer-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

element: The index of the requested array element.

valueP: A pointer to the location where the requested value should be returned.

Important note:

Note that this method should be used for array attributes only (i.e., variable or constraint attributes). To query a scalar attribute (i.e., a model attribute), use GRBgetintattr instead.

Example usage:

```
int first_one;
error = GRBgetintattrelement(model, "VBasis", 0, &first_one);
```

GRBsetintattrelement

```
int  GRBsetintattrelement (  GRBmodel    *model,
                             const char  *attrname,
                             int         element,
                             int         newvalue )
```

Set a single value in an integer-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an integer-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

element: The index of the array element to be changed.

newvalue: The value to which the attribute element should be set.

Important note:

Note that this method should be used for array attributes only (i.e., variable or constraint attributes). To modify a scalar attribute (i.e., a model attribute), use GRBsetintattr instead.

Example usage:

```
error = GRBsetintattrelement(model, "VBasis", 0, GRB_BASIC);
```

GRBgetintattrarray

```
int  GRBgetintattrarray (  GRBmodel    *model,
                           const char  *attrname,
                           int         start,
                           int         len,
                           int         *values )
```

Query the values of an integer-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an integer-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

start: The index of the first entry in the array to retrieve.

len: The number of array entries to retrieve.

values: A pointer to the location where the array attribute should be returned. Note that the result array must be as long as the requested sub-array.

Example usage:

```
int cbasis[NUMCONSTRS];
error = GRBgetintattrarray(model, "CBasis", 0, NUMCONSTRS, cbasis);
```

GRBsetintattrarray

```
int  GRBsetintattrarray (  GRBmodel    *model,
                           const char  *attrname,
                           int         start,
                           int         len,
                           int         *values )
```

Set the values of an integer-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an integer-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

start: The index of the first entry in the array to set.

len: The number of array entries to set.

values: A pointer to the desired new values for the specified sub-array of the attribute. Note that the values array must be as long as the sub-array to be changed.

Example usage:

```
int cbasis[] = {GRB_BASIC, GRB_BASIC, GRB_NONBASIC_LOWER, GRB_BASIC};
error = GRBsetintattrarray(model, "CBasis", 0, 4, cbasis);
```

GRBgetintattrlist

```
int  GRBgetintattrlist (  GRBmodel    *model,
                          const char  *attrname,
                          int         len,
                          int         *ind,
                          int         *values )
```

Query the values of an integer-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an integer-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

len: The number of attribute elements to retrieve.

ind: The indices of the desired attribute elements.

values: A pointer to the location where the requested attribute elements should be returned. Note that the result array must be as long as the requested index list.

Example usage:

```
int desired[] = {0, 2, 4, 6};
int cbasis[4];
error = GRBgetintattrlist(model, "CBasis", 4, desired, cbasis);
```

GRBsetintattrlist

```
int  GRBsetintattrlist (  GRBmodel    *model,
                          const char  *attrname,
                          int         len,
                          int         *ind,
                          int         *values )
```

Set the values of an integer-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of an integer-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

len: The number of array entries to set.

ind: The indices of the array attribute elements that will be set.

values: A pointer to the desired new values for the specified elements of the attribute. Note that the values array must be as long as the list of indices.

Example usage:

```
int change[] = {0, 1, 3};
int newbas[] = {GRB_BASIC, GRB_NONBASIC_LOWER, GRB_NONBASIC_LOWER};
error = GRBsetintattrlist(model, "VBasis", 3, change, newbas);
```

GRBgetdblattr

```
int  GRBgetdblattr (  GRBmodel    *model,
                      const char  *attrname,
                      double      *valueP )
```

Query the value of a double-valued model attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a double-valued model attribute. Available attributes are listed and described in the Attributes section of this document.

valueP: The location in which the current value of the requested attribute should be placed.

Important note:

Note that this method should be used for scalar attributes only (i.e., model attributes). To query a single element of an array attribute, use GRBgetdblattrelement instead.

Example usage:

```
error = GRBgetdblattr(model, "ObjCon", &objcon);
```

GRBsetdblattr

```
int  GRBsetdblattr (  GRBmodel    *model,
                      const char  *attrname,
                      double      newvalue )
```

Set the value of a double-valued model attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a double-valued model attribute. Available attributes are listed and described in the Attributes section of this document.

newvalue: The desired new value of this attribute.

Important note:

Note that this method should be used for scalar attributes only (i.e., model attributes). To modify a single element of an array attribute, use GRBsetdblattrelement instead.

Example usage:

```
error = GRBsetdblattr(model, "ObjCon", 0.0);
```

GRBgetdblattrelement

```
int  GRBgetdblattrelement (  GRBmodel    *model,
                             const char  *attrname,
                             int         element,
                             double      *valueP )
```

Query a single value from a double-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a double-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

element: The index of the requested array element.

values: A pointer to the location where the requested value should be returned.

Important note:

Note that this method should be used for array attributes only (i.e., variable or constraint attributes). To query a scalar attribute (i.e., a model attribute), use GRBgetdblattr instead.

Example usage:

```
double first_one;
error = GRBgetdblattrelement(model, "X", 0, &first_one);
```

GRBsetdblattrelement

```
int  GRBsetdblattrelement (  GRBmodel    *model,
                             const char  *attrname,
                             int         element,
                             double      newvalue )
```

Set a single value in a double-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a double-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

element: The index of the array element to be changed.

newvalue: The value to which the attribute element should be set.

Important note:

Note that this method should be used for array attributes only (i.e., variable or constraint attributes). To modify a scalar attribute (i.e., a model attribute), use GRBsetdblattr instead.

Example usage:

```
error = GRBsetdblattrelement(model, "Start", 0, 1.0);
```

GRBgetdblattrarray

```
int  GRBgetdblattrarray (  GRBmodel    *model,
                           const char  *attrname,
                           int         start,
                           int         len,
                           double      *values )
```

Query the values of a double-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a double-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

start: The index of the first entry in the array to retrieve.

len: The number of array entries to retrieve.

values: A pointer to the location where the array attribute should be returned. Note that the result array must be as long as the requested sub-array.

Example usage:

```
double lb[NUMVARS];
error = GRBgetdblattrarray(model, "LB", 0, cols, lb);
```

GRBsetdblattrarray

```
int  GRBsetdblattrarray (  GRBmodel    *model,
                           const char  *attrname,
                           int         start,
                           int         len,
                           double      *values )
```

Set the values of a double-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a double-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

start: The index of the first entry in the array to set.

len: The number of array entries to set.

values: A pointer to the desired new values for the specified sub-array of the attribute. Note that the values array must be as long as the sub-array to be changed.

Example usage:

```
double start[] = {1.0, 1.0, 0.0, 1.0};
error = GRBsetdblattrarray(model, "Start", 0, 4, start);
```

GRBgetdblattrlist

```
int  GRBgetdblattrlist (  GRBmodel     *model,
                          const char   *attrname,
                          int          len,
                          int          *ind,
                          double       *values )
```

Query the values of a double-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a double-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

len: The number of attribute elements to retrieve.

ind: The indices of the desired attribute elements.

values: A pointer to the location where the requested attribute elements should be returned. Note that the result array must be as long as the requested index list.

Example usage:

```
int desired[] = {0, 2, 4, 6};
double x[4];
error = GRBgetdblattrlist(model, "X", 4, desired, cbasis);
```

GRBsetdblattrlist

```
int  GRBsetdblattrlist (  GRBmodel     *model,
                          const char   *attrname,
                          int          len,
                          int          *ind,
                          double       *values )
```

Set the values of a double-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a double-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

len: The number of array entries to set.

ind: The indices of the array attribute elements that will be set.

values: A pointer to the desired new values for the specified elements of the attribute. Note that the values array must be as long as the list of indices.

Example usage:

```
int change[] = {0, 1, 3};
double start[] = {1.0, 3.0, 2.0};
error = GRBsetdblattrlist(model, "Start", 3, change, start);
```

GRBgetcharattrelement

```
int  GRBgetcharattrelement (  GRBmodel     *model,
                              const char   *attrname,
                              int          element,
                              char         *valueP )
```

Query a single value from a character-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a character-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

element: The index of the requested array element.

values: A pointer to the location where the requested value should be returned.

Example usage:

```
char first_one;
error = GRBgetcharattrelement(model, "VType", 0, &first_one);
```

GRBsetcharattrelement

```
int  GRBsetcharattrelement (  GRBmodel    *model,
                              const char  *attrname,
                              int         element,
                              char        newvalue )
```

Set a single value in a character-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a character-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

element: The index of the array element to be changed.

newvalue: The value to which the attribute element should be set.

Example usage:

```
error = GRBsetcharattrelement(model, "VType", 0, GRB_BINARY);
```

GRBgetcharattrarray

```
int  GRBgetcharattrarray (  GRBmodel    *model,
                            const char  *attrname,
                            int         start,
                            int         len,
                            char        *values )
```

Query the values of a character-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a character-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

start: The index of the first entry in the array to retrieve.

len: The number of array entries to retrieve.

values: A pointer to the location where the array attribute should be returned. Note that the result array must be as long as the requested sub-array.

Example usage:

```
char vtypes[NUMVARS];
error = GRBgetcharattrarray(model, "VType", 0, NUMVARS, vtypes);
```

GRBsetcharattrarray

```
int  GRBsetcharattrarray (  GRBmodel    *model,
                            const char  *attrname,
                            int         start,
                            int         len,
                            char        *values )
```

Set the values of a character-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a character-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

start: The index of the first entry in the array to set.

len: The number of array entries to set.

values: A pointer to the desired new values for the specified sub-array of the attribute. Note that the values array must be as long as the sub-array to be changed.

Example usage:

```
char vtypes[] = {GRB_BINARY, GRB_CONTINUOUS, GRB_INTEGER, GRB_BINARY};
error = GRBsetcharattrarray(model, "VType", 0, 4, vtypes);
```

GRBgetcharattrlist

```
int  GRBgetcharattrlist (  GRBmodel    *model,
                           const char  *attrname,
                           int         len,
                           int         *ind,
                           char        *values )
```

Query the values of a character-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a character-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

len: The number of attribute elements to retrieve.

ind: The indices of the desired attribute elements.

values: A pointer to the location where the requested attribute elements should be returned. Note that the result array must be as long as the requested index list.

Example usage:

```
int desired[] = {0, 2, 4, 6};
char vtypes[4];
error = GRBgetcharattrlist(model, "VType", 4, desired, vtypes);
```

GRBsetcharattrlist

```
int  GRBsetcharattrlist (  GRBmodel    *model,
                           const char  *attrname,
                           int         len,
                           int         *ind,
                           char        *values )
```

Set the values of a character-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a character-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

len: The number of array entries to set.

ind: The indices of the array attribute elements that will be set.

values: A pointer to the desired new values for the specified elements of the attribute. Note that the values array must be as long as the list of indices.

Example usage:

```
int change[] = {0, 1, 3};
char vtypes[] = {GRB_BINARY, GRB_BINARY, GRB_BINARY};
error = GRBsetcharattrlist(model, "Vtype", 3, change, vtypes);
```

GRBgetstrattr

```
int  GRBgetstrattr (  GRBmodel    *model,
                      const char  *attrname,
                      char        **valueP )
```

Query the value of a string-valued model attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

`attrname`: The name of a string-valued model attribute. Available attributes are listed and described in the Attributes section of this document.

`valueP`: The location in which the current value of the requested attribute should be placed.

Important notes:

Note that all interface routines that return string-valued attributes are returning pointers into internal Gurobi data structures. The user should copy the contents of the pointer to a different data structure before the next call to a Gurobi library routine. The user should also be careful to never modify the data pointed to by the returned character pointer.

Note that this method should be used for scalar attributes only (i.e., model attributes). To query a single element of an array attribute, use GRBgetstrattrelement instead.

Example usage:

```
char *modelname;
error = GRBgetstrattr(model, "ModelName", &modelname);
```

GRBsetstrattr

```
int  GRBsetstrattr (  GRBmodel    *model,
                      const char  *attrname,
                      const char  *newvalue )
```

Set the value of a string-valued model attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

`attrname`: The name of a string-valued model attribute. Available attributes are listed and described in the Attributes section of this document.

`newvalue`: The desired new value of this attribute.

Important note:

Note that this method should be used for scalar attributes only (i.e., model attributes). To modify a single element of an array attribute, use GRBsetstrattrelement instead.

Example usage:

```
error = GRBsetstrattr(model, "ModelName", "Modified name");
```

GRBgetstrattrelement

```
int  GRBgetstrattrelement (  GRBmodel     *model,
                             const char   *attrname,
                             int          element,
                             char         **valueP )
```

Query a single value from a string-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

`attrname`: The name of a string-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

`element`: The index of the requested array element.

`valueP`: A pointer to the location where the requested value should be returned.

Important notes:

Note that all interface routines that return string-valued attributes are returning pointers into internal Gurobi data structures. The user should copy the contents of the pointer to a different data structure before the next call to a Gurobi library routine. The user should also be careful to never modify the data pointed to by the returned character pointer.

Note that this method should be used for array attributes only (i.e., variable or constraint attributes). To query a scalar attribute (i.e., a model attribute), use GRBgetstrattr instead.

Example usage:

```
char **varname;
error = GRBgetstrattrelement(model, "VarName", 1, varname);
```

GRBsetstrattrelement

```
int  GRBsetstrattrelement (  GRBmodel     *model,
                             const char   *attrname,
                             int          element,
                             char         *newvalue )
```

Set a single value in a string-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

`attrname`: The name of a string-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

element: The index of the array element to be changed.

newvalue: The value to which the attribute element should be set.

Important note:

Note that this method should be used for array attributes only (i.e., variable or constraint attributes). To modify a scalar attribute (i.e., a model attribute), use GRBsetstrattr instead.

Example usage:

```
error = GRBsetstrattrelement(model, "ConstrName", 0, "NewConstr");
```

GRBgetstrattrarray

```
int  GRBgetstrattrarray (  GRBmodel    *model,
                           const char  *attrname,
                           int         start,
                           int         len,
                           char        **values )
```

Query the values of a string-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a string-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

start: The index of the first entry in the array to retrieve.

len: The number of array entries to retrieve.

values: A pointer to the location where the array attribute should be returned. Note that the result array must be as long as the requested sub-array.

Important notes:

Note that all interface routines that return string-valued attributes are returning pointers into internal Gurobi data structures. The user should copy the contents of the pointer to a different data structure before the next call to a Gurobi library routine. The user should also be careful to never modify the data pointed to by the returned character pointer.

Example usage:

```
char **varnames[NUMVARS];
error = GRBgetstrattrarray(model, "VarName", 0, NUMVARS, varnames);
```

GRBsetstrattrarray

```
int  GRBsetstrattrarray (  GRBmodel    *model,
                           const char  *attrname,
                           int         start,
                           int         len,
                           char        **values )
```

Set the values of a string-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a string-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

start: The index of the first entry in the array to set.

len: The number of array entries to set.

values: A pointer to the desired new values for the specified sub-array of the attribute. Note that the values array must be as long as the sub-array to be changed.

Example usage:

```
char **varnames[NUMVARS];
error = GRBsetstrattrarray(model, "VarName", 0, NUMVARS, varnames);
```

GRBgetstrattrlist

```
int  GRBgetstrattrlist (  GRBmodel     *model,
                          const char   *attrname,
                          int          len,
                          int          *ind,
                          char         **values )
```

Query the values of a string-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a string-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

len: The number of attribute elements to retrieve.

ind: The indices of the desired attribute elements.

values: A pointer to the location where the requested attribute elements should be returned. Note that the result array must be as long as the requested index list.

Important notes:

Note that all interface routines that return string-valued attributes are returning pointers into internal Gurobi data structures. The user should copy the contents of the pointer to a different data structure before the next call to a Gurobi library routine. The user should also be careful to never modify the data pointed to by the returned character pointer.

Example usage:

```
int desired[] = {0, 2, 4, 6};
char **varnames[4];
error = GRBgetstrattrlist(model, "VarName", 4, desired, varnames);
```

GRBsetstrattrlist

```
int  GRBsetstrattrlist (  GRBmodel    *model,
                          const char  *attrname,
                          int         len,
                          int         *ind,
                          char        **values )
```

Set the values of a string-valued array attribute.

Return value:

A non-zero return value indicates that a problem occurred while setting the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: A loaded optimization model, typically created by routine GRBnewmodel or GRBreadmodel.

attrname: The name of a string-valued array attribute. Available attributes are listed and described in the Attributes section of this document.

len: The number of array entries to set.

ind: The indices of the array attribute elements that will be set.

values: A pointer to the desired new values for the specified elements of the attribute. Note that the values array must be as long as the list of indices.

Example usage:

```
int chage[] = {0, 1, 3};
char **varnames[] = {"Var0", "Var1", "Var3"};
error = GRBsetstrattrlist(model, "VarName", 3, change, varnames);
```

GRBgetbatchattrinfo

```
int  GRBgetbatchattrinfo (  GRBbatch    *batch,
                            const char  *attrname,
                            int         *datatypeP,
                            int         *settableP )
```

Obtain information about a Batch attribute.

Return value:

A non-zero return value indicates that a problem occurred while obtaining information about a batch attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

batch: A batch request handle, typically created by routine GRBgetbatch.

attrname: The name of a batch attribute. Available attributes are listed and described in the Attributes section of this document.

datatypeP: On completion, the integer pointed to by this argument will indicate the data type of the attribute. Possible types are char (0), int (1), double (2), or string(3). This argument can be NULL.

settableP: On completion, the integer pointed to by this argument will indicate whether the attribute can be set (1) or not (0). This argument can be NULL.

Example usage:

```
int datatype, settable;
error = GRBgetbatchattrinfo(batch, "BatchID", &datatype, &settable);
```

3.7 Parameter Management and Tuning

GRBtunemodel

```
int   GRBtunemodel ( GRBmodel  *model )
```

Perform an automated search for parameter settings that improve performance on a model. Upon completion, this routine stores the best parameter sets it found. The number of stored parameter sets can be determined by querying the value of the TuneResultCount attribute. The actual settings can be retrieved using GRBgettuneresult.

Please refer to the parameter tuning section for details on the tuning tool.

Return value:

A non-zero return value indicates that a problem occurred while tuning the model. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to be tuned.

Example usage:

```
error = GRBtunemodel(model);
if (error) goto QUIT;

error = GRBgetintattr(model, "TuneResultCount", &nresults);
if (error) goto QUIT;
```

GRBgettuneresult

```
int   GRBgettuneresult ( GRBmodel  *model,
                         int       n )
```

Use this routine to retrieve the results of a previous GRBtunemodel call. Calling this routine with argument `n` causes tuned parameter set `n` to be copied into the model. Parameter sets are stored in order of decreasing quality, with parameter set 0 being the best. The number of available sets is stored in attribute TuneResultCount.

Once you have retrieved a tuning result, you can call GRBoptimize to use these parameter settings to optimize the model, or GRBwrite to write the changed parameters to a `.prm` file.

Please refer to the parameter tuning section for details on the tuning tool.

Return value:

A non-zero return value indicates that a problem occurred while retrieving a tuning result. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: A model that has previously been used as the argument of GRBtunemodel.

`n`: The index of the tuning result to retrieve. The best result is available as index 0. The number of stored results is available in attribute TuneResultCount.

Example usage:

```
error = GRBtunemodel(model);
if (error) goto QUIT;

error = GRBgetintattr(model, "TuneResultCount", &nresults);
if (error) goto QUIT;

if (nresults > 0) {
  error = GRBgettuneresult(model, 0);
  if (error) goto QUIT;
}
```

GRBgetdblparam

```
int  GRBgetdblparam (  GRBenv      *env,
                       const char  *paramname,
                       double      *valueP )
```

Retrieve the value of a double-valued parameter.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the parameter. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter value is being queried.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valueP: The location in which the current value of the requested parameter should be placed.

Example usage:

```
double cutoff;
error = GRBgetdblparam(GRBgetenv(model), "Cutoff", &cutoff);
```

GRBgetintparam

```
int  GRBgetintparam (  GRBenv      *env,
                       const char  *paramname,
                       int         *valueP )
```

Retrieve the value of an integer-valued parameter.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the parameter. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter value is being queried.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valueP: The location in which the current value of the requested parameter should be placed.

Example usage:

```
int limit;
error = GRBgetintparam(GRBgetenv(model), "SolutionLimit", &limit);
```

GRBgetstrparam

```
int  GRBgetstrparam (  GRBenv      *env,
                       const char  *paramname,
                       char        *value )
```

Retrieve the value of a string-valued parameter.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the parameter. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter value is being queried.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

value: The location in which the current value of the requested parameter should be placed.

Example usage:

```
char logfilename[GRB_MAX_STRLEN];
error = GRBgetstrparam(GRBgetenv(model), "LogFile", logfilename);
```

GRBsetdblparam

```
int  GRBsetdblparam (  GRBenv      *env,
                       const char  *paramname,
                       double      newvalue )
```

Modify the value of a double-valued parameter.

Return value:

A non-zero return value indicates that a problem occurred while modifying the parameter. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter value is being modified.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

Important note:

Note that a model gets its own copy of the environment when it is created. Changes to the original environment have no effect on the copy. Use GRBgetenv to retrieve the environment associated with a model if you would like a parameter change to affect that model.

Example usage:

```
error = GRBsetdblparam(GRBgetenv(model), "Cutoff", 100.0);
```

GRBsetintparam

```
int   GRBsetintparam (  GRBenv      *env,
                        const char  *paramname,
                        int         newvalue )
```

Modify the value of an integer-valued parameter.

Return value:

A non-zero return value indicates that a problem occurred while modifying the parameter. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter value is being modified.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

Important note:

Note that a model gets its own copy of the environment when it is created. Changes to the original environment have no effect on the copy. Use GRBgetenv to retrieve the environment associated with a model if you would like a parameter change to affect that model.

Example usage:

```
error = GRBsetintparam(GRBgetenv(model), "SolutionLimit", 5);
```

GRBsetstrparam

```
int   GRBsetstrparam (  GRBenv      *env,
                        const char  *paramname,
                        const char  *newvalue )
```

Modify the value of a string-valued parameter.

Return value:

A non-zero return value indicates that a problem occurred while modifying the parameter. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter value is being modified.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

Important note:

Note that a model gets its own copy of the environment when it is created. Changes to the original environment have no effect on the copy. Use GRBgetenv to retrieve the environment associated with a model if you would like a parameter change to affect that model.

Example usage:

```
error = GRBsetstrparam(GRBgetenv(model), "LogFile", "/tmp/new.log");
```

GRBgetdblparaminfo

```
int  GRBgetdblparaminfo (  GRBenv      *env,
                           const char  *paramname,
                           double      *valueP,
                           double      *minP,
                           double      *maxP,
                           double      *defaultP )
```

Retrieve information about a double-valued parameter. Specifically, retrieve the current value of the parameter, the minimum and maximum allowed values, and the default value.

Return value:

A non-zero return value indicates that a problem occurred while retrieving parameter information. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter information is being queried.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valueP (optional): The location in which the current value of the specified parameter should be placed.

minP (optional): The location in which the minimum allowed value of the specified parameter should be placed.

maxP (optional): The location in which the maximum allowed value of the specified parameter should be placed.

defaultP (optional): The location in which the default value of the specified parameter should be placed.

Example usage:

```
error = GRBgetdblparaminfo(GRBgetenv(model), "MIPGap", &currentGap,
                           &minAllowedGap, NULL, &defaultGap);
```

GRBgetintparaminfo

```
int  GRBgetintparaminfo (  GRBenv      *env,
                           const char  *paramname,
                           int         *valueP,
                           int         *minP,
                           int         *maxP,
                           int         *defaultP )
```

Retrieve information about an int-valued parameter. Specifically, retrieve the current value of the parameter, the minimum and maximum allowed values, and the default value.

Return value:

A non-zero return value indicates that a problem occurred while retrieving parameter information. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter information is being queried.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valueP (optional): The location in which the current value of the specified parameter should be placed.

minP (optional): The location in which the minimum allowed value of the specified parameter should be placed.

maxP (optional): The location in which the maximum allowed value of the specified parameter should be placed.

defaultP (optional): The location in which the default value of the specified parameter should be placed.

Example usage:

```
error = GRBgetintparaminfo(GRBgetenv(model), "SolutionLimit", &current,
                           &minAllowedLimit, NULL, &defaultLimit);
```

GRBgetstrparaminfo

```
int  GRBgetstrparaminfo (  GRBenv      *env,
                           const char  *paramname,
                           char        *value,
                           char        *default )
```

Retrieve information about a string-valued parameter. Specifically, retrieve the current and default values of the parameter.

Return value:

A non-zero return value indicates that a problem occurred while retrieving parameter information. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter information is being queried.

paramname: The name of the parameter. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

value (optional): The location in which the current value of the specified parameter should be placed.

default (optional): The location in which the default value of the specified parameter should be placed.

Example usage:

```
char defaultval[GRB_MAX_STRLEN];
char currentval[GRB_MAX_STRLEN];
error = GRBgetstrparaminfo(GRBgetenv(model), "LogFile", currentval,
                           defaultval);
```

GRBreadparams

```
int  GRBreadparams (  GRBenv      *env,
                      const char  *filename )
```

Import a set of parameter modifications from a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

A non-zero return value indicates that a problem occurred while reading the parameter file. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment into which the parameter changes should be imported.

filename: The path to the file to be read. The suffix on a parameter file should be `.prm`, optionally followed by `.zip`, `.gz`, `.bz2`, or `.7z`.

Example usage:

```
error = GRBreadparams(env, "/tmp/model.prm.bz2");
```

GRBwriteparams

```
int  GRBwriteparams (  GRBenv      *env,
                       const char  *filename )
```

Write the set of changed parameter values to a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

A non-zero return value indicates that a problem occurred while writing the parameter file. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

env: The environment whose parameter changes are being written.

`filename`: The path to the file to be written. The suffix on a parameter file should be `.prm`, optionally followed by `.gz`, `.bz2`, or `.7z`.

Example usage:

```
error = GRBwriteparams(env, "/tmp/model.prm");
```

3.8 Monitoring Progress - Logging and Callbacks

GRBmsg

```
void  GRBmsg (  GRBenv      *env,
                const char  *message )
```

Insert a message into the Gurobi log file.

Arguments:

env: The environment whose log file should receive the message.

message: The message to be appended to the log.

Example usage:

```
error = GRBmsg(env, "Add this message to the log");
```

GRBsetcallbackfunc

```
int  GRBsetcallbackfunc (  GRBmodel  *model,
                           int       (*cb)(GRBmodel *model, void *cbdata, int
                                     where, void *usrdata),
                           void      *usrdata )
```

Set up a user callback function. Note that a model can only have a single callback method, so this call will replace an existing callback. To disable a previously set callback, call this function with a `cb` argument of `NULL`.

When solving a model using multiple threads, the user callback is only ever called from a single thread, so you don't need to worry about the thread-safety of your callback.

Note that changing parameters from within a callback is not supported, doing so may lead to undefined behavior.

Return value:

A non-zero return value indicates that a problem occurred while setting the user callback. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

model: The model in which the callback should be installed.

cb: A function pointer to the user callback function. The callback will be called regularly from the Gurobi Optimizer. The **where** argument to the callback function will indicate where in the optimization process the callback was invoked. Possible values are described in the Callback Codes section. The user callback can then call a number of routines to retrieve additional details about the state of the optimization (e.g., GRBcbget), or to inject new information (e.g., GRBcbcut, GRBcbsolution). The user callback function should return 0 if no error was encountered, or it can return one of the Gurobi Error Codes if the user callback would like the optimization to stop and return an error result.

usrdata: An optional pointer to user data that will be passed back to the user callback function each time it is invoked (in the **usrdata** argument).

Example usage:

```
int mycallback(GRBmodel *model, void *cbdata, int where, void *usrdata);
error = GRBsetcallbackfunc(model, mycallback, NULL);
```

GRBgetcallbackfunc

```
int  GRBgetcallbackfunc (  GRBmodel  *model,
                           int       (**cb)(GRBmodel *model, void *cbdata,
                                     int where, void *usrdata) )
```

Retrieve the current user callback function.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the user callback. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model in which the callback should be installed.

`cb`: A function pointer to the user callback function.

Example usage:

```
int (*mycallback)(GRBmodel *model, void *cbdata, int where, void *usrdata);
error = GRBgetcallbackfunc(model, &mycallback);
```

GRBcbget

```
int  GRBcbget (  void  *cbdata,
                 int   where,
                 int   what,
                 void  *resultP )
```

Retrieve additional information about the progress of the optimization. Note that this routine can only be called from within a user callback function.

Return value:

A non-zero return value indicates that a problem occurred while retrieving the requested data. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`cbdata`: The `cbdata` argument that was passed into the user callback by the Gurobi Optimizer. This argument must be passed unmodified from the user callback to `GRBcbget()`.

`where`: The `where` argument that was passed into the user callback by the Gurobi Optimizer. This argument must be passed unmodified from the user callback to `GRBcbget()`.

`what`: The data requested by the user callback. Valid values are described in the Callback Codes section.

`resultP`: The location in which the requested data should be placed.

Example usage:

```
if (where == GRB_CB_MIP) {
  double nodecount;
  error = GRBcbget(cbdata, where, GRB_CB_MIP_NODECNT, (void *) &nodecount);
  if (error) return 0;
  printf("MIP node count is %d\n", nodecount);
}
```

GRBversion

```
void  GRBversion (  int  *majorP,
                    int  *minorP,
                    int  *technicalP )
```

Return the Gurobi library version number (major, minor, and technical).

Arguments:

majorP: The location in which the major version number should be placed. May be NULL.

minorP: The location in which the minor version number should be placed. May be NULL.

technicalP: The location in which the technical version number should be placed. May be NULL.

Example usage:

```
int major, minor, technical;
GRBversion(&major, &minor, &technical);
printf("Gurobi library version %d.%d.%d\n", major, minor, technical);
```

3.9 Modifying Solver Behavior - Callbacks

GRBcbcut

```
int  GRBcbcut (  void          *cbdata,
                 int           cutlen,
                 const int     *cutind,
                 const double  *cutval,
                 char          cutsense,
                 double        cutrhs )
```

Add a new cutting plane to the MIP model from within a user callback routine. Note that this routine can only be called when the `where` value on the callback routine is `GRB_CB_MIPNODE` (see the Callback Codes section for more information).

Cutting planes can be added at any node of the branch-and-cut tree. Note that cuts should be added sparingly, since they increase the size of the relaxation model that is solved at each node and can significantly degrade node processing speed.

Cutting planes are typically used to cut off the current relaxation solution. To retrieve the relaxation solution at the current node, call GRBcbget with `what = GRB_CB_MIPNODE_REL`.

You should consider setting parameter PreCrush to value 1 when adding your own cuts. This setting shuts off a few presolve reductions that can sometimes prevent your cut from being applied to the presolved model (which would result in your cut being silently ignored).

One very important note: you should only add cuts that are implied by the constraints in your model. If you cut off an integer solution that is feasible according to the original model constraints, *you are likely to obtain an incorrect solution to your MIP problem*.

Return value:

A non-zero return value indicates that a problem occurred while adding the cut. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`cbdata`: The `cbdata` argument that was passed into the user callback by the Gurobi Optimizer. This argument must be passed unmodified from the user callback to `GRBcbcut()`.

`cutlen`: The number of non-zero coefficients in the new cutting plane.

`cutind`: Variable indices for non-zero values in the new cutting plane.

`cutval`: Numerical values for non-zero values in the new cutting plane.

`cutsense`: Sense for the new cutting plane. Options are `GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`.

`cutrhs`: Right-hand side value for the new cutting plane.

Example usage:

```
if (where == GRB_CB_MIPNODE) {
  int cutind[] = {0, 1};
  double cutval[] = {1.0, 1.0};
  error = GRBcbcut(cbdata, 2, cutind, cutval, GRB_LESS_EQUAL, 1.0);
  if (error) return 0;
}
```

GRBcblazy

```
int  GRBcblazy (  void          *cbdata,
                  int           lazylen,
                  const int     *lazyind,
                  const double  *lazyval,
                  char          lazysense,
                  double        lazyrhs )
```

Add a new lazy constraint to the MIP model from within a user callback routine. Note that this routine can only be called when the `where` value on the callback routine is either `GRB_CB_MIPNODE` or `GRB_CB_MIPSOL` (see the Callback Codes section for more information).

Lazy constraints are typically used when the full set of constraints for a MIP model is too large to represent explicitly. By only including the constraints that are actually violated by solutions found during the branch-and-cut search, it is sometimes possible to find a proven optimal solution while only adding a fraction of the full set of constraints.

You would typically add a lazy constraint by querying the current node solution (by calling GRBcbget from a `GRB_CB_MIPSOL` or `GRB_CB_MIPNODE` callback, using `what=GRB_CB_MIPSOL_SOL` or `what=GRB_CB_MIPNODE_REL`), and then calling `GRBcblazy()` to add a constraint that cuts off the solution. Gurobi guarantees that you will have the opportunity to cut off any solutions that would otherwise be considered feasible.

MIP solutions may be generated outside of a MIP node. Thus, generating lazy constraints is optional when the `where` value in the callback function equals `GRB_CB_MIPNODE`. To avoid this, we recommend to always check when the `where` value equals `GRB_CB_MIPSOL`.

Your callback should be prepared to cut off solutions that violate any of your lazy constraints, including those that have already been added. Node solutions will usually respect previously added lazy constraints, but not always.

Note that you must set the LazyConstraints parameter if you want to use lazy constraints.

Return value:

A non-zero return value indicates that a problem occurred while adding the lazy constraint. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`cbdata`: The `cbdata` argument that was passed into the user callback by the Gurobi Optimizer. This argument must be passed unmodified from the user callback to `GRBcblazy()`.

`lazylen`: The number of non-zero coefficients in the new lazy constraint.

`lazyind`: Variable indices for non-zero values in the new lazy constraint.

`lazyval`: Numerical values for non-zero values in the new lazy constraint.

`lazysense`: Sense for the new lazy constraint. Options are `GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`.

`lazyrhs`: Right-hand side value for the new lazy constraint.

Example usage:

```
if (where == GRB_CB_MIPSOL) {
  int lazyind[] = {0, 1};
  double lazyval[] = {1.0, 1.0};
  error = GRBcblazy(cbdata, 2, lazyind, lazyval, GRB_LESS_EQUAL, 1.0);
```

```
  if (error) return 0;
}
```

GRBcbsolution

```
int  GRBcbsolution (  void          *cbdata,
                      const double  *solution,
                      double        *objP )
```

Provide a new feasible solution for a MIP model from within a user callback routine. Note that this routine can only be called when the `where` value on the callback routine is `GRB_CB_MIP`, `GRB_CB_MIPNODE`, or `GRB_CB_MIPSOL` (see the Callback Codes section for more information).

Heuristics solutions are typically built from the current relaxation solution. To retrieve the relaxation solution at the current node, call GRBcbget with `what = GRB_CB_MIPNODE_REL`.

When providing a solution, you can specify values for any subset of the variables in the model. To leave a variable value unspecified, set the variable to `GRB_UNDEFINED` in the `solution` vector. The Gurobi MIP solver will attempt to extend the specified partial solution to a complete solution.

Note that this method is not supported in a Compute Server environment.

Return value:

A non-zero return value indicates that a problem occurred while adding the new solution. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`cbdata`: The `cbdata` argument that was passed into the user callback by the Gurobi Optimizer. This argument must be passed unmodified from the user callback to `GRBcbsolution()`.

`solution`: The solution vector. You must provide one entry for each variable in the model. Note that you can leave an entry unspecified by setting it to `GRB_UNDEFINED`. The Gurobi optimizer will attempt to find appropriate values for the unspecified variables.

`objP`: Objective value for solution that results from this call. Returns `GRB_INFINITY` if no improved solution is found or the function has been called from a callback other than `GRB_CB_MIPNODE` as, in these contexts, the solution is stored instead of being processed immediately.

Example usage:

```
if (where == GRB_CB_MIPNODE) {
  error = GRBcbsolution(cbdata, solution, &obj);
  if (error) return 0;
}
```

GRBcbproceed

```
void  GRBcbproceed (  GRBModel  *model )
```

Generate a request to proceed to the next phase of the computation. This routine can be called from any callback. Note that the request is only accepted in a few phases of the algorithm, and it won't be acted upon immediately.

In the current Gurobi version, this callback allows you to proceed from the NoRel heuristic to the standard MIP search. You can determine the current algorithm phase using `MIP_PHASE`, `MIPNODE_PHASE`, or `MIPSOL_PHASE` queries from a callback.

Arguments:

`model`: The model.

Example usage:

```
if (solution_objective < target_value) {
  GRBcbproceed(model);
}
```

GRBcbstoponemultiobj

```
int  GRBcbstoponemultiobj (  GRBmodel  *model,
                             void*     cbdata,
                             int       objnum )
```

Interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process. Note that this routine can only be called for multi-objective MIP models and when the `where` value on the callback routine is not equal to `GRB_CB_MULTIOBJ` (see the Callback Codes section for more information).

You would typically stop a multi-objective optimization step by querying the last finished number of multi-objectives steps, and using that number to stop the current step and move on to the next hierarchical objective (if any) as shown in the following example:

Example usage:

```
#include <time.h>

typedef struct {
  int    objcnt;
  time_t starttime;
} usrdata_t;

int mycallback(GRBmodel *model,
               void     *cbdata,
               int       where,
               void     *usrdata)
{
  int error = 0;
  usrdata_t *ud = (usrdata_t*)usrdata;

  if (where == GRB_CB_MULTIOBJ) {
    /* get current objective number */
    error = GRBcbget(cbdata, where, MULTIOBJ_OBJCNT, (void*)&ud->objcnt);
    if (error) goto QUIT;

    /* reset start time to current time */
```

```
    ud->starttime = time();

  } else if (time() - ud->starttime > BIG ||
             /* takes too long or good enough */) {
    /* stop only this optimization step */
    error = GRBcbstoponemultiobj(model, cbdata, ud->objcnt);
    if (error) goto QUIT;
  }

QUIT:
  return error;
}
```

You should refer to the section on Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Return value:

A non-zero return value indicates that a problem occurred while stopping the multi-objective step specified by objcnt. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The `model` argument that was passed into the user callback by the Gurobi Optimizer. This argument must be passed unmodified from the user callback to `GRBcbstoponemultiobj()`.

`cbdata`: The `cbdata` argument that was passed into the user callback by the Gurobi Optimizer. This argument must be passed unmodified from the user callback to `GRBcbstoponemultiobj()`.

`objnum`: The number of the multi-objective optimization step to interrupt. For processes running locally, this argument can have the special value -1, meaning to stop the current step.

GRBterminate

```
void  GRBterminate (  GRBmodel  *model )
```

Generate a request to terminate the current optimization. This routine can be called at any time during an optimization (from a callback, from another thread, from an interrupt handler, etc.). Note that, in general, the request won't be acted upon immediately.

When the optimization stops, the Status attribute will be equal to `GRB_INTERRUPTED`.

Arguments:

`model`: The model to terminate.

Example usage:

```
if (time_to_quit)
  GRBterminate(model);
```

3.10 Batch Requests

GRBabortbatch

```
int  GRBabortbatch (  GRBbatch  *batch )
```

This function instructs the Cluster Manager to abort the processing of this batch request, changing its status to `ABORTED`. Please refer to the Batch Status Codes section for further details.

Return value:

A non-zero return value indicates that a problem occurred while aborting the batch request. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`batch`: The batch that will be aborted.

Example usage:

```
/* request to abort the batch */
error = GRBabortbatch(batch);
if (error) goto QUIT;
```

GRBdiscardbatch

```
int  GRBdiscardbatch (  GRBbatch  *batch )
```

This function instructs the Cluster Manager to remove all information related to the batch request in question, including the stored solution if available. Further queries for the associated batch request will fail with error code `GRB_ERROR_DATA_NOT_AVAILABLE`. Use this function with care, as the removed information can not be recovered later on.

Return value:

A non-zero return value indicates that a problem occurred while discarding the batch. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`batch`: The batch that will be discarded.

Example usage:

```
/* discard the batch object in the manager */
error = GRBdiscardbatch(batch);
if (error) goto QUIT;
```

GRBfreebatch

```
int  GRBfreebatch (  GRBbatch  *batch )
```

Free a batch structure and release the associated memory.

Return value:

A non-zero return value indicates that a problem occurred while freeing the batch. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`batch`: The batch structure to be freed.

Example usage:

```
GRBfreebatch(batch);
```

GRBgetbatch

```
int  GRBgetbatch (  GRBenv      *env,
                    const char  *BatchID,
                    GRBbatch    **batchP )
```

Given a `BatchID`, as returned by GRBoptimizebatch, and a Gurobi environment that can connect to the appropriate Cluster Manager (i.e., one where parameters CSManager, UserName, and ServerPassword have been set appropriately), this function returns a GRBbatch structure. With it, you can query the current status of the associated batch request and, once the batch request has been processed, you can query its solution. Please refer to the Batch Optimization section for details and examples.

Return value:

A non-zero return value indicates that a problem occurred while creating a `GRBbatch` structure. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`env`: The environment in which the new batch structure should be created.

`BatchID`: ID of the batch you want to access.

`batchP`: The location in which the pointer to the batch structure should be placed.

Example usage:

```
/* create batch-object */
error = GRBgetbatch(env, BatchID, &batch);
if (error) goto QUIT;
```

GRBgetbatchenv

```
GRBenv *  GRBgetbatchenv (  GRBbatch  *batch )
```

Retrieve the environment associated with a batch.

Return value:

The environment associated with the batch. A `NULL` return value indicates that there was a problem retrieving the environment.

Arguments:

`batch`: The batch from which the environment should be retrieved.

Example usage:

```
GRBenv *env = GRBgetbatchenv(batch);
```

GRBgetbatchintattr

```
int  GRBgetbatchintattr (  GRBbatch    *batch,
                           const char  *attrname,
                           int         *valueP )
```

Query the value of an integer-valued batch attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`batch`: A batch structure, typically created by routine GRBgetbatch.

`attrname`: The name of an integer-valued batch attribute. Available attributes are listed and described in the Attributes section of this document.

`valueP`: The location in which the current value of the requested attribute should be placed.

Example usage:

```
/* query the last error code */
error = GRBgetbatchintattr(batch, "BatchErrorCode", &errorCode);
if (error || !errorCode) goto QUIT;
```

Important notes:

Note that all Batch attributes are cached locally, and are only updated when you create a client-side batch object or when you explicitly update this cache (by calling the appropriate `update` function - GRBupdatebatch for **C**, update for **Python**, etc.).

GRBgetbatchjsonsolution

```
int  GRBgetbatchjsonsolution (  GRBbatch  *batch,
                                char**    jsonsolP )
```

This function retrieves the solution of a completed batch request from a Cluster Manager. The solution is returned as a JSON solution string. For this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details.

Return value:

A non-zero return value indicates that a problem occurred while querying the batch solution. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`batch`: The batch to query.

`jsonsolP`: The location in which the pointer to the newly created JSON string should be placed.

Important note:

On Windows, the string returned in `buffP` is allocated in a different heap from the calling program. You must call `GRBfree` to free it.

Example usage:

```
/* print JSON solution into string */
error = GRBgetbatchjsonsolution(batch, &jsonsol);
if (error) goto QUIT;
printf("JSON solution: %s\n", jsonsol);
```

GRBgetbatchstrattr

```
int  GRBgetbatchstrattr (  GRBbatch    *batch,
                           const char  *attrname,
                           char        **valueP )
```

Query the value of a string-valued batch attribute.

Return value:

A non-zero return value indicates that a problem occurred while querying the attribute. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

batch: A batch structure, typically created by routine GRBgetbatch.

attrname: The name of a string-valued batch attribute. Available attributes are listed and described in the Attributes section of this document.

valueP: The location in which the current value of the requested attribute should be placed.

Example usage:

```
/* query the last error message */
error = GRBgetbatchstrattr(batch, "BatchErrorMessage", &errorMsg);
if (error) goto QUIT;
```

Important notes:

Note that all interface routines that return string-valued attributes are returning pointers into internal Gurobi data structures. The user should copy the contents of the pointer to a different data structure before the next call to a Gurobi library routine. The user should also be careful to never modify the data pointed to by the returned character pointer.

Note that all Batch attributes are cached locally, and are only updated when you create a client-side batch object or when you explicitly update this cache (by calling the appropriate `update` function - GRBupdatebatch for **C**, update for **Python**, etc.).

GRBoptimizebatch

```
int  GRBoptimizebatch (  GRBmodel  *model,
                         char      *BatchID )
```

Submit a new batch request to the Cluster Manager. Returns the `BatchID` (a string), which uniquely identifies the job in the Cluster Manager and can be used to query the status of this request (from this program or from any other). Once the request has completed, the `BatchID` can also be used to retrieve the associated solution. To submit a batch request, you must tag at least one element of the model by setting one of the VTag, CTag or QCTag attributes. For more details on batch optimization, please refer to the Batch Optimization section.

Note that this routine will process all pending model modifications.

Return value:

A non-zero return value indicates that a problem occurred while submit a batch request. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model to optimize in batch mode. Note that this routine only reports whether the batch request ran into an error.

`BatchID`: On success, the location in which the BatchID of the newly created batch request should be stored. The pointer must point to a string of length `GRB_MAX_STRLEN+1` or more.

Example usage:

```
/* submit batch request to the Manager */
error = GRBoptimizebatch(model, BatchID);
if (error) goto QUIT;
```

GRBretrybatch

```
int  GRBretrybatch (  GRBbatch  *batch )
```

This function instructs the Cluster Manager to retry optimization of a failed or aborted batch request, changing its status to `SUBMITTED`. Please refer to the Batch Status Codes section for further details.

Return value:

A non-zero return value indicates that a problem occurred while retrying the batch. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`batch`: The batch to retry.

Example usage:

```
/* retry the batch request */
error = GRBretrybatch(batch);
if (error) goto QUIT;
```

GRBupdatebatch

```
int  GRBupdatebatch (  GRBbatch  *batch )
```

All Batch attribute values are cached locally, so queries return the value received during the last communication with the Cluster Manager. This function refreshes the values of all attributes with the values currently available in the Cluster Manager (which involves network communication).

Return value:

A non-zero return value indicates that a problem occurred while updating the batch request. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`batch`: The batch that will be updated.

Example usage:

```
/* update local attributes */
error = GRBupdatebatch(batch);
if (error) goto QUIT;
```

GRBwritebatchjsonsolution

```
int  GRBwritebatchjsonsolution (  GRBbatch     *batch,
                                  const char*  filename )
```

This function returns the stored solution of a completed batch request from a Cluster Manager. The solution is returned in a gzip-compressed JSON file. The file name you provide must end with a `.json.gz` extension. The JSON format is described in the JSON solution format section. Note that for this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details.

Return value:

A non-zero return value indicates that a problem occurred while writing the JSON solution string into the given filename. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`batch`: The batch request from qhere to query its solution.

`filename`: The name of the file in which to store the JSON solution. It must be a file name ending with the `.json.gz` extension.

Example usage:

```
/* save solution into a file */
error = GRBwritebatchjsonsolution(batch, "batch-sol.json.gz");
if (error) goto QUIT;
```

3.11 Error Handling

GRBgeterrormsg

```
char *  GRBgeterrormsg (  GRBenv  *env )
```

Retrieve the error message associated with the most recent error that occurred in an environment.

Return value:

A string containing the error message.

Arguments:

env: The environment in which the error occurred.

Example usage:

```
error = GRBgetintattr(model, "DOES_NOT_EXIST", &attr);
if (error)
  printf("%s\n", GRBgeterrormsg(env));
```

3.12 Advanced simplex routines

This section describes a set of advanced basis routines. These routines allow you to compute solutions to various linear systems involving the simplex basis matrix. Note that these should only be used by advanced users. We provide no technical support for these routines.

Before describing the routines, we should first describe the `GRBsvec` data structure that is used to input or return sparse vectors:

```
typedef   struct SVector {
          int              len;
          int              *ind;
          double           *val;
          } GRBsvec;
```

The `len` field gives the number of non-zero values in the vector. The `ind` and `val` fields give the index and value for each non-zero, respectively. Indices are zero-based. To give an example, the sparse vector `[0, 2.0, 0, 1.0]` would be represented as `len=2`, `ind = [1, 3]`, and `val = [2.0, 1.0]`.

The user is responsible for allocating and freeing the `ind` and `val` fields. The length of the result vector for these routines is not known in advance, so the user must allocate these arrays to hold the longest possible result (whose length is noted in the documentation for each routine).

GRBFSolve

```
int   GRBFSolve (  GRBmodel   *model,
                   GRBsvec    *b,
                   GRBsvec    *x )
```

Computes the solution to the linear system $Bx = b$, where B is the current simplex basis matrix, b is an input vector, and x is the result vector.

Return value:

A non-zero return value indicates that a problem occurred while computing the desired vector. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model. Note that the model must have a current optimal basis, as computed by GRBoptimize.

`b`: The sparse right-hand side vector. It should contain one entry for each non-zero value in the input.

`x`: The sparse result vector. The user is responsible for allocating the `ind` and `val` fields to be large enough to hold as many as one non-zero entry per constraint in the model.

GRBBSolve

```
int   GRBBSolve (  GRBmodel   *model,
                   GRBsvec    *b,
                   GRBsvec    *x )
```

Computes the solution to the linear system $B^T x = b$, where B^T is the transpose of the current simplex basis matrix, b is an input vector, and x is the result vector.

Return value:

A non-zero return value indicates that a problem occurred while computing the desired vector. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model. Note that the model must have a current optimal basis, as computed by GRBoptimize.

`b`: The sparse right-hand side vector. It should contain one entry for each non-zero value in the input.

`x`: The sparse result vector. The user is responsible for allocating the `ind` and `val` fields to be large enough to hold as many as one non-zero entry per constraint in the model.

GRBBinvColj

```
int   GRBBinvColj (  GRBmodel  *model,
                     int       j,
                     GRBsvec   *x )
```

Computes the solution to the linear system $Bx = A_j$, where B is the current simplex basis matrix and A_j is the column of the constraint matrix A associated with variable j.

Return value:

A non-zero return value indicates that a problem occurred while computing the desired vector. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model. Note that the model must have a current optimal basis, as computed by GRBoptimize.

`j`: Indicates the index of the column of A to use as the right-hand side for the linear solve. The index `j` must be between 0 and `cols-1`, where `cols` is the number of columns in the model.

`x`: The sparse result vector. The user is responsible for allocating the `ind` and `val` fields to be large enough to hold as many as one non-zero entry per constraint in the model.

GRBBinvRowi

```
int   GRBBinvRowi (  GRBmodel  *model,
                     int       i,
                     GRBsvec   *x )
```

Computes a single *tableau row*. More precisely, this routine returns row i from the matrix $B^{-1}A$, where B^{-1} is the inverse of the basis matrix and A is the constraint matrix. Note that the tableau will contain columns corresponding to the variables in the model, and also columns corresponding to artificial and slack variables associated with constraints.

Return value:

A non-zero return value indicates that a problem occurred while computing the desired vector. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model. Note that the model must have a current optimal basis, as computed by GRBoptimize.

`i`: The index of the desired tableau row.

`x`: The result vector. The result will contain one entry for each non-zero value. Note that the result may contain values for slack variables; the slack on row `i` will have index `cols+i`, where `cols` is the number of columns in the model. The user is responsible for allocating the `ind` and `val` fields to be large enough to hold the largest possible result. For this routine, the result could have one entry for each variable in the model, plus one entry for each constraint.

GRBgetBasisHead

```
int  GRBgetBasisHead ( GRBmodel  *model,
                       int       *bhead )
```

Returns the indices of the variables that make up the current basis matrix.

Return value:

A non-zero return value indicates that a problem occurred while extracting the basis. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by calling GRBgeterrormsg.

Arguments:

`model`: The model. Note that the model must have a current optimal basis, as computed by GRBoptimize.

`bhead`: The constraint matrix columns that make up the current basis. The result contains one entry per constraint in A. If `bhead[i]=j`, then column `i` in the basis matrix B is column `j` from the constraint matrix A. Note that the basis may contain slack or artificial variables. If `bhead[i]` is greater than or equal to `cols` (the number of columns in A), then the corresponding basis column is the artificial or slack variable from row `bhead[i]-cols`.

C++ API Overview

This section documents the Gurobi C++ interface. This manual begins with a quick overview of the classes exposed in the interface and the most important methods on those classes. It then continues with a comprehensive presentation of all of the available classes and methods.

If you are new to the Gurobi Optimizer, we suggest that you start with the Getting Started Knowledge Base article or the Example Tour. These documents provide concrete examples of how to use the classes and methods described here.

Environments

The first step in using the Gurobi C++ interface is to create an environment object. Environments are represented using the GRBEnv class. An environment acts as the container for all data associated with a set of optimization runs. You will generally only need one environment object in your program.

For more advanced use cases, you can use an empty environment to create an uninitialized environment and then, programmatically, set all required options for your specific requirements. For further details see the Environment section.

Models

You can create one or more optimization models within an environment. Each model is represented as an object of class GRBModel. A model consists of a set of decision variables (objects of class GRBVar), a linear or quadratic objective function on those variables (specified using GRBModel::setObjective), and a set of constraints on these variables (objects of class GRBConstr, GRBQConstr, GRBSOS, or GRBGenConstr). Each variable has an associated lower bound, upper bound, and type (continuous, binary, etc.). Each linear or quadratic constraint has an associated sense (less-than-or-equal, greater-than-or-equal, or equal), and right-hand side value. Refer to this section for more information on variables, constraints, and objectives.

Linear constraints are specified by building linear expressions (objects of class GRBLinExpr), and then specifying relationships between these expressions (for example, requiring that one expression be equal to another). Quadratic constraints are built in a similar fashion, but using quadratic expressions (objects of class GRBQuadExpr) instead.

An optimization model may be specified all at once, by loading the model from a file (using the appropriate GRBModel constructor), or built incrementally, by first constructing an empty object of class GRBModel and then subsequently calling GRBModel::addVar or GRBModel::addVars to add additional variables, and GRBModel::addConstr, GRBModel::addQConstr, GRBModel::addSOS, or any of the GRBModel::addGenConstrXxx methods to add constraints. Models are dynamic entities; you can always add or remove variables or constraints.

We often refer to the *class* of an optimization model. At the highest level, a model can be continuous or discrete, depending on whether the modeling elements present in the model require discrete decisions to be made. Among continuous models...

- A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*.

- If the objective is quadratic, the model is a *Quadratic Program (QP)*.

- If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We sometimes refer to a few special cases of QCP: QCPs with convex constraints, QCPs with non-convex constraints, *bilinear programs*, and *Second-Order Cone Programs (SOCP)*.

- If any of the constraints are non-linear (chosen from among the available general constraints), the model is a *Non-Linear Program (NLP)*.

A model that contains any integer variables, semi-continuous variables, semi-integer variables, Special Ordered Set (SOS) constraints, or general constraints, is discrete, and is referred to as a *Mixed Integer Program (MIP)*. The special cases of MIP, which are the discrete versions of the continuous models types we've already described, are...

- *Mixed Integer Linear Programs (MILP)*

- *Mixed Integer Quadratic Programs (MIQP)*

- *Mixed Integer Quadratically-Constrained Programs (MIQCP)*

- *Mixed Integer Second-Order Cone Programs (MISOCP)*

- *Mixed Integer Non-Linear Programs (MINLP)*

The Gurobi Optimizer handles all of these model classes. Note that the boundaries between them aren't as clear as one might like, because we are often able to transform a model from one class to a simpler class.

Solving a Model

Once you have built a model, you can call GRBModel::optimize to compute a solution. By default, optimize will use the concurrent optimizer to solve LP models, the barrier algorithm to solve QP models with convex objectives and QCP models with convex constraints, and the branch-and-cut algorithm otherwise. The solution is stored in a set of *attributes* of the model. These attributes can be queried using a set of attribute query methods on the GRBModel, GRBVar, GRBConstr, GRBQConstr, GRBSOS, and GRBGenConstr classes.

The Gurobi algorithms keep careful track of the state of the model, so calls to GRBModel::optimize will only perform further optimization if relevant data has changed since the model was last optimized. If you would like to discard previously computed solution information and restart the optimization from scratch without changing the model, you can call GRBModel::reset.

After a MIP model has been solved, you can call GRBModel::fixedModel to compute the associated *fixed* model. This model is identical to the original, except that the integer variables are fixed to their values in the MIP solution. If your model contains SOS constraints, some continuous variables that appear in these constraints may be fixed as well. In some applications, it can be useful to compute information on this fixed model (e.g., dual variables, sensitivity information, etc.), although you should be careful in how you interpret this information.

Multiple Solutions, Objectives, and Scenarios

By default, the Gurobi Optimizer assumes that your goal is to find one proven optimal solution to a single model with a single objective function. Gurobi provides the following features that allow you to relax these assumptions:

- Solution Pool: Allows you to find more solutions.
- Multiple Scenarios: Allows you to find solutions to multiple, related models.
- Multiple Objectives: Allows you to specify multiple objective functions and control the trade-off between them.

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call GRBModel::computeIIS to compute an Irreducible Inconsistent Subsystem (IIS). This method can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. This method populates a set of IIS attributes.

To attempt to repair an infeasibility, call GRBModel::feasRelax to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Querying and Modifying Attributes

Most of the information associated with a Gurobi model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. To give a simple example, solving an optimization model causes the `X` variable attribute to be populated. Attributes such as `X` that are computed by the Gurobi Optimizer cannot be modified directly by the user, while others, such as the variable lower bound (the `LB` attribute) can.

Attributes are queried using GRBVar::get, GRBConstr::get, GRBQConstr::get, GRBSOS::get, GRBGenConstr::get, or GRBModel::get, and modified using GRBVar::set, GRBConstr::set, GRBQConstr::set, GRBGenConstr::set, or GRBModel::set. Attributes are grouped into a set of enums by type (GRB_CharAttr, GRB_DoubleAttr, GRB_IntAttr, GRB_StringAttr). The `get()` and `set()` methods are overloaded, so the type of the attribute determines the type of the returned value. Thus, `constr.get(GRB.DoubleAttr.RHS)` returns a double, while `constr.get(GRB.CharAttr.Sense)` returns a char.

If you wish to retrieve attribute values for a set of variables or constraints, it is usually more efficient to use the array methods on the associated GRBModel object. Method GRBModel::get includes signatures that allow you to query or modify attribute values for arrays of variables or constraints.

The full list of attributes can be found in the Attributes section.

Additional Model Modification Information

Most modifications to an existing model are done through the attribute interface (e.g., changes to variable bounds, constraint right-hand sides, etc.). The main exceptions are modifications to the constraint matrix and the objective function.

The constraint matrix can be modified in a few ways. The first is to call the chgCoeffs method on a GRBModel object to change individual matrix coefficients. This method can be used to modify the value of an existing non-zero, to set an existing non-zero to zero, or to create a new non-zero. The constraint matrix is also modified when you remove a variable or constraint from the model (through the GRBModel::remove method). The non-zero values associated with the deleted constraint or variable are removed along with the constraint or variable itself.

The model objective function can also be modified in a few ways. The easiest is to build an expression that captures the objective function (a GRBLinExpr or GRBQuadExpr object), and then pass that expression to method GRBModel::setObjective. If you wish to modify the objective, you can simply call `setObjective` again with a new `GRBLinExpr` or `GRBQuadExpr` object.

For linear objective functions, an alternative to `setObjective` is to use the `Obj` variable attribute to modify individual linear objective coefficients.

If your variables have piecewise-linear objectives, you can specify them using the GRBModel::setPWLObj method. Call this method once for each relevant variable. The Gurobi simplex solver includes algorithmic support for convex piecewise-linear objective functions, so for continuous models you should see a substantial performance benefit from using this feature. To clear a previously specified piecewise-linear objective function, simply set the `Obj` attribute on the corresponding variable to 0.

Lazy Updates

One important item to note about model modification in the Gurobi optimizer is that it is performed in a *lazy* fashion, meaning that modifications don't affect the model immediately. Rather, they are queued and applied later. If your program simply creates a model and solves it, you will probably never notice this behavior. However, if you ask for information about the model before your modifications have been applied, the details of the lazy update approach may be relevant to you.

As we just noted, model modifications (bound changes, right-hand side changes, objective changes, etc.) are placed in a queue. These queued modifications can be applied to the model in three different ways. The first is by an explicit call to GRBModel::update. The second is by a call to GRBModel::optimize. The third is by a call to GRBModel::write to write out the model. The first case gives you fine-grained control over when modifications are applied. The second and third make the assumption that you want all pending modifications to be applied before you optimize your model or write it to disk.

Why does the Gurobi interface behave in this manner? There are a few reasons. The first is that this approach makes it much easier to perform multiple modifications to a model, since the model remains unchanged between modifications. The second is that processing model modifications can be expensive, particularly in a Compute Server environment, where modifications require communication between machines. Thus, it is useful to have visibility into exactly when these modifications are applied. In general, if your program needs to make multiple modifications to the model, you should aim to make them in phases, where you make a set of modifications, then update, then make more modifications, then update again, etc. Updating after each individual modification can be extremely expensive.

If you forget to call update, your program won't crash. Your query will simply return the value of the requested data from the point of the last update. If the object you tried to query didn't exist then, you'll get a `NOT_IN_MODEL` exception instead.

The semantics of lazy updates have changed since earlier Gurobi versions. While the vast

majority of programs are unaffected by this change, you can use the UpdateMode parameter to revert to the earlier behavior if you run into an issue.

Managing Parameters

The Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization. Parameters can be of type *int*, *double*, or *string*.

The simplest way to set parameters is through the GRBModel::set method on the model object. Similarly, parameter values can be queried with GRBModel::get.

Parameters can also be set on the Gurobi environment object, using GRBEnv::set. Note that each model gets its own copy of the environment when it is created, so parameter changes to the original environment have no effect on existing models.

You can read a set of parameter settings from a file using GRBEnv::readParams, or write the set of changed parameters using GRBEnv::writeParams.

We also include an automated parameter tuning tool that explores many different sets of parameter changes in order to find a set that improves performance. You can call GRBModel::tune to invoke the tuning tool on a model. Refer to the parameter tuning tool section for more information.

The full list of Gurobi parameters can be found in the Parameters section.

Memory Management

Memory management must always be considered in C++ programs. In particular, the Gurobi library and the user program share the same C++ heap, so the user must be aware of certain aspects of how the Gurobi library uses this heap. The basic rules for managing memory when using the Gurobi Optimizer are as follows:

- As with other dynamically allocated C++ objects, GRBEnv or GRBModel objects should be freed using the associated destructors. In other words, given a `GRBModel` object `m`, you should call `delete m` when you are no longer using `m`.
- Objects that are associated with a model (e.g., GRBConstr, GRBQConstr, GRBSOS, GRBGenConstr, and GRBVar objects) are managed by the model. In particular, deleting a model will delete all of the associated objects. Similarly, removing an object from a model (using GRBModel::remove) will also delete the object.
- Some Gurobi methods return an array of objects or values. For example, GRBModel::addVars returns an array of `GRBVar` objects. It is the user's responsibility to free the returned array (using `delete[]`). The reference manual indicates when a method returns a heap-allocated result.

One consequence of these rules is that you must be careful not to use an object once it has been freed. This is no doubt quite clear for environments and models, where you call the destructors explicitly, but may be less clear for constraints and variables, which are implicitly deleted when the associated model is deleted.

Monitoring Progress - Logging and Callbacks

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging

behavior. If you would like to direct output to a file as well as to the screen, specify the log file name in the GRBEnv constructor. You can modify the LogFile parameter if you wish to redirect the log to a different file after creating the environment object. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter. A detailed description of the Gurobi log file can be found in the Logging section.

More detailed progress monitoring can be done through the GRBCallback class. The GRBModel::setCallback method allows you to receive a periodic callback from the Gurobi Optimizer. You do this by sub-classing the GRBCallback abstract class, and writing your own `callback()` method on this class. You can call GRBCallback::getDoubleInfo, GRBCallback::getIntInfo, GRBCallback::getStringInfo, or GRBCallback::getSolution from within the callback to obtain additional information about the state of the optimization.

Modifying Solver Behavior - Callbacks

Callbacks can also be used to modify the behavior of the Gurobi optimizer. The simplest control callback is GRBCallback::abort, which asks the optimizer to terminate at the earliest convenient point. Method GRBCallback::setSolution allows you to inject a feasible solution (or partial solution) during the solution of a MIP model. Methods GRBCallback::addCut and GRBCallback::addLazy allow you to add *cutting planes* and *lazy constraints* during a MIP optimization, respectively. Method GRBCallback::stopOneMultiObj allows you to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

Batch Optimization

Gurobi Compute Server enables programs to offload optimization computations onto dedicated servers. The Gurobi Cluster Manager adds a number of additional capabilities on top of this. One important one, *batch optimization*, allows you to build an optimization model with your client program, submit it to a Compute Server cluster (through the Cluster Manager), and later check on the status of the model and retrieve its solution. You can use a Batch object to make it easier to work with batches. For details on batches, please refer to the Batch Optimization section.

Error Handling

All of the methods in the Gurobi C++ library can throw an exception of type GRBException. When an exception occurs, additional information on the error can be obtained by retrieving the error code (using method `GRBException::getErrorCode`), or by retrieving the exception message (using method `GRBException::getMessage`). The list of possible error return codes can be found in the Error Codes section.

4.1 GRBEnv

Gurobi environment object. Gurobi models are always associated with an environment. You must create an environment before can you create and populate a model. You will generally only need a single environment object in your program.

The methods on environment objects are mainly used to manage Gurobi parameters (e.g., get, getParamInfo, set).

GRBEnv()

Constructor for `GRBEnv` object. You have the option of constructing either a local environment, which solves Gurobi models on the local machine, a client environment for a Gurobi Compute Server, which will solve Gurobi models on a server machine, or an Instant Cloud environment, which will launch a Gurobi Cloud server and solve models on that server. Choose the appropriate signature for the type of environment you wish to launch.

```
GRBEnv  GRBEnv ( )
```

Create a Gurobi environment (with logging disabled). This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Return value:

An environment object (with no associated log file).

```
GRBEnv  GRBEnv (  bool  empty )
```

Create an empty Gurobi environment. Use GRBEnv::start to start the environment.

If the environment is not empty, This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`empty`: Indicates whether the environment should be empty. You should use `empty=true` if you want to set parameters before actually starting the environment. This can be useful if you want to connect to a Compute Server, a Token Server, the Gurobi Instant Cloud, a Cluster Manager or use a WLS license. See the Environment Section for more details.

Return value:

An environment object.

```
GRBEnv  GRBEnv (  const string&  logFileName )
```

Create a Gurobi environment (with logging enabled). This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`logFileName`: The desired log file name.

Return value:

An environment object.

GRBEnv::get()

Query the value of a parameter.

```
double  get (  GRB_DoubleParam  param )
```

Query the value of a double-valued parameter.

Arguments:

`param`: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

```
int  get (  GRB_IntParam  param )
```

Query the value of an int-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

```
string  get ( GRB_StringParam  param )
```

Query the value of a string-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

GRBEnv::getErrorMsg()

Query the error message for the most recent exception associated with this environment.

```
const string  getErrorMsg ( )
```

Return value:

The error string.

GRBEnv::getParamInfo()

Obtain information about a parameter.

```
void  getParamInfo ( GRB_DoubleParam  param,
                     double*          valP,
                     double*          minP,
                     double*          maxP,
                     double*          *defP )
```

Obtain detailed information about a double parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valP: The current value of the parameter.

minP: The minimum allowed value of the parameter.

maxP: The maximum allowed value of the parameter.

defP: The default value of the parameter.

```
void  getParamInfo (  GRB_IntParam  param,
                      int*          valP,
                      int*          minP,
                      int*          maxP,
                      int*          defP )
```

Obtain detailed information about an integer parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valP: The current value of the parameter.

minP: The minimum allowed value of the parameter.

maxP: The maximum allowed value of the parameter.

defP: The default value of the parameter.

```
void  getParamInfo (  GRB_StringParam  param,
                      string*          valP,
                      string*          defP )
```

Obtain detailed information about a string parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valP: The current value of the parameter.

defP: The default value of the parameter.

GRBEnv::message()

Write a message to the console and the log file.

```
void  message (  const string&  message )
```

Arguments:

message: Print a message to the console and to the log file. Note that this call has no effect unless the `OutputFlag` parameter is set.

GRBEnv::readParams()

Read new parameter settings from a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  readParams (  const string&  paramfile )
```

Arguments:

paramfile: Name of the file containing parameter settings. Parameters should be listed one per line, with the parameter name first and the desired value second. For example:

```
# Gurobi parameter file
Threads 1
MIPGap 0
```

Blank lines and lines that begin with the hash symbol are ignored.

GRBEnv::resetParams()

Reset all parameters to their default values.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  resetParams ( )
```

GRBEnv::set()

Set the value of a parameter.

Important notes:

Note that a model gets its own copy of the environment when it is created. Changes to the original environment have no effect on the copy. Use GRBModel::set to change a parameter on an existing model.

```
void  set ( GRB_DoubleParam  param,
            double           newvalue )
```

Set the value of a double-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  set ( GRB_IntParam  param,
            int           newvalue )
```

Set the value of an int-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  set ( GRB_StringParam  param,
            const string&    newvalue )
```

Set the value of a string-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  set ( const string&  param,
            const string&  newvalue )
```

Set the value of any parameter using strings alone.

Arguments:

param: The name of the parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

GRBEnv::start()

Start an empty environment. If the environment has already been started, this method will do nothing. If the call fails, the environment will have the same state as it had before the call to this method.

This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter). After that, it will apply all parameter changes specified by the user prior to this call. Note that this might overwrite parameters set in the license file, or in the `gurobi.env` file, if present.

After all these changes are performed, the code will actually activate the environment, and make it ready to work with models.

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  start ( )
```

GRBEnv::writeParams()

Write all non-default parameter settings to a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  writeParams (  const string&  paramfile )
```

Arguments:

paramfile: Name of the file to which non-default parameter settings should be written. The previous contents are overwritten.

4.2 GRBModel

Gurobi model object. Commonly used methods include addVar (adds a new decision variable to the model), addConstr (adds a new constraint to the model), optimize (optimizes the current model), and get (retrieves the value of an attribute).

GRBModel()

Constructor for `GRBModel`. The simplest version creates an empty model. You can then call addVar and addConstr to populate the model with variables and constraints. The more complex constructors can read a model from a file, or make a copy of an existing model.

```
GRBModel  GRBModel (  const GRBEnv&  env )
```

Model constructor.

Arguments:

`env`: Environment for new model.

Return value:

New model object. Model initially contains no variables or constraints.

```
GRBModel  GRBModel (  const GRBEnv&  env,
                      const string&  filename )
```

Read a model from a file. Note that the type of the file is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, `.rlp`, `.dua`, `.dlp`, `.ilp`, or `.opb`. The files can be compressed, so additional suffixes of `.zip`, `.gz`, `.bz2`, or `.7z` are accepted.

Arguments:

`env`: Environment for new model.

`modelname`: Name of the file containing the model.

Return value:

New model object.

```
GRBModel  GRBModel (  const GRBModel&  model )
```

Create a copy of an existing model. Note that due to the lazy update approach in Gurobi, you have to call update before copying it.

Arguments:

`model`: Model to copy.

Return value:

New model object. Model is a clone of the original.

```
GRBModel  GRBModel (  const GRBModel&  model,
                      const GRBEnv&    targetenv )
```

Copy an existing model to a different environment. Multiple threads can not work simultaneously within the same environment. Copies of models must therefore reside in different environments for multiple threads to operate on them simultaneously.

Note that this method itself is not thread safe, so you should either call it from the main thread or protect access to it with a lock.

Note that pending updates will not be applied to the model, so you should call update before copying if you would like those to be included in the copy.

For Compute Server users, note that you can copy a model from a client to a Compute Server environment, but it is not possible to copy models from a Compute Server environment to another (client or Compute Server) environment.

Arguments:

`model`: Model to copy.

`targetenv`: Environment to copy model into.

Return value:

New model object. Model is a clone of the original.

GRBModel::addConstr()

Add a single linear constraint to a model. Multiple signatures are available.

```
GRBConstr  addConstr (  const GRBLinExpr&  lhsExpr,
                        char               sense,
                        const GRBLinExpr&  rhsExpr,
                        string             name="" )
```

Add a single linear constraint to a model.

Arguments:

`lhsExpr`: Left-hand side expression for new linear constraint.

`sense`: Sense for new linear constraint (`GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`).

`rhsExpr`: Right-hand side expression for new linear constraint.

`name (optional)`: Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  const GRBLinExpr&  lhsExpr,
                        char               sense,
                        GRBVar             rhsVar,
                        string             name="" )
```

Add a single linear constraint to a model.

Arguments:

`lhsExpr`: Left-hand side expression for new linear constraint.

`sense`: Sense for new linear constraint (`GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`).

`rhsVar`: Right-hand side variable for new linear constraint.

`name (optional)`: Name for new constraint.

Return value:
New constraint object.

```
GRBConstr  addConstr (  const GRBLinExpr&  lhsExpr,
                        char               sense,
                        double             rhsVal,
                        string             name="" )
```

Add a single linear constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new linear constraint.

sense: Sense for new linear constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVal: Right-hand side value for new linear constraint.

name (optional): Name for new constraint.

Return value:
New constraint object.

```
GRBConstr  addConstr (  GRBVar  lhsVar,
                        char    sense,
                        GRBVar  rhsVar,
                        string  name="" )
```

Add a single linear constraint to a model.

Arguments:

lhsVar: Left-hand side variable for new linear constraint.

sense: Sense for new linear constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVar: Right-hand side variable for new linear constraint.

name (optional): Name for new constraint.

Return value:
New constraint object.

```
GRBConstr  addConstr (  GRBVar  lhsVar,
                        char    sense,
                        double  rhsVal,
                        string  name="" )
```

Add a single linear constraint to a model.

Arguments:

lhsVar: Left-hand side variable for new linear constraint.

sense: Sense for new linear constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVal: Right-hand side value for new linear constraint.

name (optional): Name for new constraint.

Return value:
New constraint object.

```
GRBConstr  addConstr (  GRBTempConstr&  tc,
                        string          name="" )
```

Add a single linear constraint to a model.

Arguments:

tc: Temporary constraint object, created using an overloaded comparison operator. See GRBTempConstr for more information.

name (optional): Name for new constraint.

Return value:

New constraint object.

GRBModel::addConstrs()

Add new linear constraints to a model.

We recommend that you build your model one constraint at a time (using addConstr), since it introduces no significant overhead and we find that it produces simpler code. Feel free to use these methods if you disagree, though.

```
GRBConstr*  addConstrs (  int  count )
```

Add `count` new linear constraints to a model.

Arguments:

count: Number of constraints to add to the model. The new constraints are all of the form `0 <= 0`.

Return value:

Array of new constraint objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
GRBConstr*  addConstrs (  const GRBLinExpr*  lhsExprs,
                          const char*        senses,
                          const double*      rhsVals,
                          const string*      names,
                          int                count )
```

Add `count` new linear constraints to a model.

Arguments:

lhsExprs: Left-hand side expressions for the new linear constraints.

senses: Senses for new linear constraints (`GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`).

rhsVals: Right-hand side values for the new linear constraints.

names: Names for new constraints.

count: Number of constraints to add.

Return value:

Array of new constraint objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBModel::addGenConstrXxx()

Each of the functions described below adds a new *general constraint* to a model.

Mathematical programming has traditionally defined a set of fundamental constraint types: variable bound constraints, linear constraints, quadratic constraints, integrality constraints, and SOS constraints. These are typically treated directly by the underlying solver (although not always), and are fundamental to the overall algorithm.

Gurobi accepts a number of additional constraint types, which we collectively refer to as *general (function) constraints*. These are typically *not* treated directly by the solver. Rather, they are transformed by presolve into constraints (and variables) chosen from among the fundamental types listed above. In some cases, the resulting constraint or constraints are mathematically equivalent to the original; in others, they are approximations. If such constraints appear in your model, but if you prefer to reformulate them yourself using fundamental constraint types instead, you can certainly do so. However, note that Gurobi can sometimes exploit information contained in the other constraints in the model to build a more efficient formulation than what you might create.

The additional constraint types that fall under this *general constraint* umbrella are:

- addGenConstrMax: $y = \max(x_1, x_2, ..., c)$
- addGenConstrMin: $y = \min(x_1, x_2, ..., c)$
- addGenConstrAbs: $y = |x|$
- addGenConstrAnd: $y = x_1 \wedge x_2 \wedge x_3 ...$
- addGenConstrOr: $y = x_1 \vee x_2 \vee x_3 ...$
- addGenConstrNorm: $y = norm(x_1, x_2, x_3 ...)$
- addGenConstrIndicator: $y = 1 \rightarrow a'x \leq b$ (an indicator constraint)
- addGenConstrPWL: $y = pwl(x)$ (a piecewise-linear function, specified using breakpoints)
- addGenConstrPoly: $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$
- addGenConstrExp: $y = e^x$
- addGenConstrExpA: $y = a^x$
- addGenConstrLog: $y = \log_e(x)$
- addGenConstrLogA: $y = \log_a(x)$
- addGenConstrLogistic: $y = \frac{1}{1+e^{-x}}$
- addGenConstrPow: $y = x^a$
- addGenConstrSin: $y = \sin(x)$
- addGenConstrCos: $y = \cos(x)$
- addGenConstrTan: $y = \tan(x)$

Please refer to this section for additional details on general constraints.

GRBModel::addGenConstrMax()

Add a new general constraint of type `GRB_GENCONSTR_MAX` to a model.

A MAX constraint $r = \max\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the maximum of the operand variables $x_1, \ldots, x_n$ and the constant c.

```
GRBGenConstr  addGenConstrMax (  GRBVar         resvar,
                                 const GRBVar*  vars,
                                 int            len,
                                 double         constant=-GRB_INFINITY,
                                 string         name="" )
```

Arguments:

`resvar`: The resultant variable of the new constraint.

`vars`: Array of variables that are the operands of the new constraint.

`len`: Number of operands in the new constraint (length of **`vars`** array).

`constant (optional)`: The additional constant operand of the new constraint.

`name (optional)`: Name for the new general constraint.

Return value:

New general constraint.

GRBModel::addGenConstrMin()

Add a new general constraint of type `GRB_GENCONSTR_MIN` to a model.

A MIN constraint $r = \min\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the minimum of the operand variables $x_1, \ldots, x_n$ and the constant c.

```
GRBGenConstr  addGenConstrMin (  GRBVar         resvar,
                                 const GRBVar*  vars,
                                 int            len,
                                 double         constant=GRB_INFINITY,
                                 string         name="" )
```

Arguments:

`resvar`: The resultant variable of the new constraint.

`vars`: Array of variables that are the operands of the new constraint.

`len`: Number of operands in the new constraint (length of **`vars`** array).

`constant (optional)`: The additional constant operand of the new constraint.

`name (optional)`: Name for the new general constraint.

Return value:

New general constraint.

GRBModel::addGenConstrAbs()

Add a new general constraint of type `GRB_GENCONSTR_ABS` to a model.

An ABS constraint $r = \text{abs}\{x\}$ states that the resultant variable r should be equal to the absolute value of the argument variable x.

```
GRBGenConstr  addGenConstrAbs (  GRBVar  resvar,
                                 GRBVar  argvar,
                                 string  name="" )
```

Arguments:

resvar: The resultant variable of the new constraint.

argvar: The argument variable of the new constraint.

name (optional): Name for the new general constraint.

Return value:

New general constraint.

GRBModel::addGenConstrAnd()

Add a new general constraint of type GRB_GENCONSTR_AND to a model.

An AND constraint $r = \text{and}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if all of the operand variables $x_1, \ldots, x_n$ are equal to 1. If any of the operand variables is 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

```
GRBGenConstr  addGenConstrAnd (  GRBVar         resvar,
                                 const GRBVar*  vars,
                                 int            len,
                                 string         name="" )
```

Arguments:

resvar: The resultant binary variable of the new constraint.

vars: Array of binary variables that are the operands of the new constraint.

len: Number of operands in the new constraint (length of **vars** array).

name (optional): Name for the new general constraint.

Return value:

New general constraint.

GRBModel::addGenConstrOr()

Add a new general constraint of type GRB_GENCONSTR_OR to a model.

An OR constraint $r = \text{or}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if any of the operand variables $x_1, \ldots, x_n$ is equal to 1. If all operand variables are 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

```
GRBGenConstr  addGenConstrOr (  GRBVar         resvar,
                                const GRBVar*  vars,
                                int            len,
                                string         name="" )
```

Arguments:

resvar: The resultant binary variable of the new constraint.
vars: Array of binary variables that are the operands of the new constraint.
len: Number of operands in the new constraint (length of **vars** array).
name (optional): Name for the new general constraint.

Return value:

New general constraint.

GRBModel::addGenConstrNorm()

Add a new general constraint of type GRB_GENCONSTR_NORM to a model.

A NORM constraint $r = \text{norm}\{x_1, \ldots, x_n\}$ states that the resultant variable r should be equal to the vector norm of the argument vector $x_1, \ldots, x_n$.

```
GRBGenConstr  addGenConstrNorm (  GRBVar         resvar,
                                  const GRBVar*  vars,
                                  int            len,
                                  double         which,
                                  string         name="" )
```

Arguments:

resvar: The resultant variable of the new constraint.
vars: Array of variables that are the operands of the new constraint. Note that this array may not contain duplicates.
len: Number of operands in the new constraint (length of **vars** array).
which: Which norm to use. Options are 0, 1, 2, and GRB_INFINITY.
name (optional): Name for the new general constraint.

Return value:

New general constraint.

GRBModel::addGenConstrIndicator()

Add a new general constraint of type GRB_GENCONSTR_INDICATOR to a model.

An INDICATOR constraint $z = f \rightarrow a^T x \leq b$ states that if the binary indicator variable z is equal to f, where $f \in \{0, 1\}$, then the linear constraint $a^T x \leq b$ should hold. On the other hand, if $z = 1 - f$, the linear constraint may be violated. The sense of the linear constraint can also be specified to be $=$ or $\geq$.

Note that the indicator variable z of a constraint will be forced to be binary, independent of how it was created.

Multiple signatures are available.

```
GRBGenConstr  addGenConstrIndicator (  GRBVar             binvar,
                                       int                binval,
                                       const GRBLinExpr&  expr,
                                       char               sense,
                                       double             rhs,
                                       string             name="" )
```

Arguments:

binvar: The binary indicator variable.

binval: The value for the binary indicator variable that would force the linear constraint to be satisfied (0 or 1).

expr: Left-hand side expression for the linear constraint triggered by the indicator.

sense: Sense for the linear constraint. Options are `GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`.

rhs: Right-hand side value for the linear constraint.

name (optional): Name for the new general constraint.

Return value:

New general constraint.

```
GRBGenConstr  addGenConstrIndicator (  GRBVar                binvar,
                                       int                   binval,
                                       const GRBTempConstr&  constr,
                                       string                name="" )
```

Arguments:

binvar: The binary indicator variable.

binval: The value for the binary indicator variable that would force the linear constraint to be satisfied (0 or 1).

constr: Temporary constraint object defining the linear constraint that is triggered by the indicator. The temporary constraint object is created using an overloaded comparison operator. See GRBTempConstr for more information.

name (optional): Name for the new general constraint.

Return value:

New general constraint.

GRBModel::addGenConstrPWL()

Add a new general constraint of type `GRB_GENCONSTR_PWL` to a model.

A piecewise-linear (PWL) constraint states that the relationship $y = f(x)$ must hold between variables x and y, where f is a piecewise-linear function. The breakpoints for f are provided as arguments. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined.

```
GRBGenConstr  addGenConstrPWL (  GRBVar         xvar,
                                 GRBVar         yvar,
                                 int            npts,
                                 const double*  xpts,
                                 const double*  ypts,
                                 std::string    name="" )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

npts: The number of points that define the piecewise-linear function.

xpts: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

ypts: The y values for the points that define the piecewise-linear function.

name (optional): Name for the new general constraint.

Return value:

New general constraint.

GRBModel::addGenConstrPoly()

Add a new general constraint of type `GRB_GENCONSTR_POLY` to a model.

A polynomial function constraint states that the relationship $y = p_0x^d + p_1x^{d-1} + ... + p_{d-1}x + p_d$ should hold between variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrPoly (  GRBVar         xvar,
                                  GRBVar         yvar,
                                  int            plen,
                                  const double*  p,
                                  std::string    name="",
                                  std::string    options="" )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

plen: The length of coefficient array **p**. If x^d is the highest power term, then **plen** should be $d + 1$.

p: The coefficients for the polynomial function (starting with the coefficient for the highest power).

name (optional): Name for the new general constraint.

options (optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrExp()

Add a new general constraint of type `GRB_GENCONSTR_EXP` to a model.

A natural exponential function constraint states that the relationship $y = exp(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrExp (  GRBVar       xvar,
                                 GRBVar       yvar,
                                 std::string  name="",
                                 std::string  options="" )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name (optional): Name for the new general constraint.

options (optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrExpA()

Add a new general constraint of type GRB_GENCONSTR_EXPA to a model.

An exponential function constraint states that the relationship $y = a^x$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrExpA (  GRBVar       xvar,
                                  GRBVar       yvar,
                                  double       a,
                                  std::string  name="",
                                  std::string  options="" )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

a: The base of the function, $a > 0$.

name (optional): Name for the new general constraint.

options (optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces).

Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrLog()

Add a new general constraint of type `GRB_GENCONSTR_LOG` to a model.

A natural logarithmic function constraint states that the relationship $y = log(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrLog (  GRBVar       xvar,
                                 GRBVar       yvar,
                                 std::string  name="",
                                 std::string  options="" )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

`name (optional)`: Name for the new general constraint.

`options (optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrLogA()

Add a new general constraint of type `GRB_GENCONSTR_LOGA` to a model.

A logarithmic function constraint states that the relationship $y = log_a(x)$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrLogA (  GRBVar       xvar,
                                  GRBVar       yvar,
                                  double       a,
                                  std::string  name="",
                                  std::string  options="" )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

`a`: The base of the function, $a > 0$.

`name (optional)`: Name for the new general constraint.

`options (optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrLogistic()

Add a new general constraint of type `GRB_GENCONSTR_LOGISTIC` to a model.

A logistic function constraint states that the relationship $y = \frac{1}{1+e^{-x}}$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrLogistic (  GRBVar       xvar,
                                      GRBVar       yvar,
                                      std::string  name="",
                                      std::string  options="" )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

`name (optional)`: Name for the new general constraint.

`options (optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrPow()

Add a new general constraint of type `GRB_GENCONSTR_POW` to a model.

A power function constraint states that the relationship $y = x^a$ should hold for variables x and y, where a is the (constant) exponent.

If the exponent a is negative, the lower bound on x must be strictly positive. If the exponent isn't an integer, the lower bound on x must be non-negative.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrPow (  GRBVar       xvar,
                                 GRBVar       yvar,
                                 double       a,
                                 std::string  name="",
                                 std::string  options="" )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

a: The exponent of the function.

name (optional): Name for the new general constraint.

options (optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrSin()

Add a new general constraint of type `GRB_GENCONSTR_SIN` to a model.

A sine function constraint states that the relationship $y = sin(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrSin (  GRBVar       xvar,
                                 GRBVar       yvar,
                                 std::string  name="",
                                 std::string  options="" )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name (optional): Name for the new general constraint.

options (optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrCos()

Add a new general constraint of type GRB_GENCONSTR_COS to a model.

A cosine function constraint states that the relationship $y = cos(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrCos (  GRBVar       xvar,
                                 GRBVar       yvar,
                                 std::string  name="",
                                 std::string  options="" )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name (optional): Name for the new general constraint.

options (optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addGenConstrTan()

Add a new general constraint of type GRB_GENCONSTR_TAN to a model.

A tangent function constraint states that the relationship $y = tan(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the

same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrTan (  GRBVar       xvar,
                                 GRBVar       yvar,
                                 std::string  name="",
                                 std::string  options="" )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name (optional): Name for the new general constraint.

options (optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel::addQConstr()

Add a quadratic constraint to a model. Multiple signatures are available.

Important note: Gurobi can handle both convex and non-convex quadratic constraints. The differences between them can be both important and subtle. Refer to this discussion for additional information.

```
GRBQConstr  addQConstr (  const GRBQuadExpr&  lhsExpr,
                          char                sense,
                          const GRBQuadExpr&  rhsExpr,
                          string              name="" )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsExpr: Right-hand side expression for new quadratic constraint.

name (optional): Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  const GRBQuadExpr&  lhsExpr,
                          char                sense,
                          GRBVar              rhsVar,
                          string              name="" )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVar: Right-hand side variable for new quadratic constraint.

name (optional): Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  GRBTempConstr&  tc,
                          string          name="" )
```

Add a quadratic constraint to a model.

Arguments:

tc: Temporary constraint object, created using an overloaded comparison operator. See GRBTempConstr for more information.

name (optional): Name for new constraint.

Return value:

New quadratic constraint object.

GRBModel::addRange()

Add a single range constraint to a model. A range constraint states that the value of the input expression must be between the specified **lower** and **upper** bounds in any solution.

Note that range constraints are stored internally as equality constraints. We add an extra variable to the model to capture the range information. Thus, the Sense attribute on a range constraint will always be GRB_EQUAL. In particular introducing a range constraint

$$L \leq a^T x \leq U$$

is equivalent to adding a slack variable s and the following constraints

$$\begin{aligned} a^T x - s &= L \\ 0 \leq s &\leq U - L. \end{aligned}$$

```
GRBConstr  addRange (  const GRBLinExpr&  expr,
                       double             lower,
                       double             upper,
                       string             name="" )
```

Arguments:

expr: Linear expression for new range constraint.

lower: Lower bound for linear expression.

upper: Upper bound for linear expression.

name (optional): Name for new constraint.

Return value:

New constraint object.

GRBModel::addRanges()

Add new range constraints to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution.

```
GRBConstr*  addRanges (  const GRBLinExpr*  exprs,
                         const double*      lower,
                         const double*      upper,
                         const string*      names,
                         int                count )
```

Arguments:

`exprs`: Linear expressions for the new range constraints.
`lower`: Lower bounds for linear expressions.
`upper`: Upper bounds for linear expressions.
`name`: Names for new range constraints.
`count`: Number of range constraints to add.

Return value:

Array of new constraint objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBModel::addSOS()

Add an SOS constraint to the model. Please refer to this section for details on SOS constraints.

```
GRBSOS  addSOS (  const GRBVar*  vars,
                  const double*  weights,
                  int            len,
                  int            type )
```

Arguments:

`vars`: Array of variables that participate in the SOS constraint.
`weights`: Weights for the variables in the SOS constraint.
`len`: Number of members in the new SOS set (length of `vars` and `weights` arrays).
`type`: SOS type (can be `GRB_SOS_TYPE1` or `GRB_SOS_TYPE2`).

Return value:

New SOS constraint.

GRBModel::addVar()

Add a single decision variable to a model.

```
GRBVar  addVar (  double  lb,
                  double  ub,
                  double  obj,
                  char    type,
                  string  name="" )
```

Add a variable; non-zero entries will be added later.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT).

name (optional): Name for new variable.

Return value:

New variable object.

```
GRBVar  addVar (  double            lb,
                  double            ub,
                  double            obj,
                  char              type,
                  int               numnz,
                  const GRBConstr*  constrs,
                  const double*     coeffs,
                  string            name="" )
```

Add a variable, and the associated non-zero coefficients.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT).

numnz: Number of constraints in which this new variable participates.

constrs: Array of constraints in which the variable participates.

coeffs: Array of coefficients for each constraint in which the variable participates.

name (optional): Name for new variable.

Return value:

New variable object.

```
GRBVar  addVar (  double            lb,
                  double            ub,
                  double            obj,
                  char              type,
                  const GRBColumn&  col,
                  string            name="" )
```

Add a variable, and the associated non-zero coefficients.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT).

col: GRBColumn object for specifying a set of constraints to which new variable belongs.

name (optional): Name for new variable.

Return value:

New variable object.

GRBModel::addVars()

Add new decision variables to a model.

```
GRBVar*  addVars (  int   count,
                    char  type=GRB_CONTINUOUS )
```

Add **count** new decision variables to a model. All associated attributes take their default values, except the variable **type**, which is specified as an argument.

Arguments:

count: Number of variables to add.

type (optional): Variable type for new variables (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT).

Return value:

Array of new variable objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
GRBVar*  addVars (  const double*  lb,
                    const double*  ub,
                    const double*  obj,
                    const char*    type,
                    const string*  names,
                    int            count )
```

Add **count** new decision variables to a model. This signature allows you to use arrays to hold the various variable attributes (lower bound, upper bound, etc.).

Arguments:

lb: Lower bounds for new variables. Can be NULL, in which case the variables get lower bounds of 0.0.

ub: Upper bounds for new variables. Can be NULL, in which case the variables get infinite upper bounds.

obj: Objective coefficients for new variables. Can be NULL, in which case the variables get objective coefficients of 0.0.

type: Variable types for new variables (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT). Can be NULL, in which case the variables are assumed to be continuous.

names: Names for new variables. Can be NULL, in which case all variables are given default names.

count: The number of variables to add.

Return value:

Array of new variable objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
GRBVar*  addVars (  const double*     lb,
                    const double*     ub,
                    const double*     obj,
                    const char*       type,
                    const string*     names,
                    const GRBColumn*  cols,
                    int               count )
```

Add new decision variables to a model. This signature allows you to specify the set of constraints to which each new variable belongs using an array of GRBColumn objects.

Arguments:

`lb`: Lower bounds for new variables. Can be `NULL`, in which case the variables get lower bounds of 0.0.

`ub`: Upper bounds for new variables. Can be `NULL`, in which case the variables get infinite upper bounds.

`obj`: Objective coefficients for new variables. Can be `NULL`, in which case the variables get objective coefficients of 0.0.

`type`: Variable types for new variables (`GRB_CONTINUOUS`, `GRB_BINARY`, `GRB_INTEGER`, `GRB_SEMICONT`, or `GRB_SEMIINT`). Can be `NULL`, in which case the variables are assumed to be continuous.

`names`: Names for new variables. Can be `NULL`, in which case all variables are given default names.

`cols`: GRBColumn objects for specifying a set of constraints to which each new column belongs.

`count`: The number of variables to add.

Return value:

Array of new variable objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBModel::chgCoeff()

Change one coefficient in the model. The desired change is captured using a `GRBVar` object, a `GRBConstr` object, and a desired coefficient for the specified variable in the specified constraint. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel::update), optimize the model (using GRBModel::optimize), or write the model to disk (using GRBModel::write).

```
void  chgCoeff (  GRBConstr  constr,
                  GRBVar     var,
                  double     newvalue )
```

Arguments:

`constr`: Constraint for coefficient to be changed.

var: Variable for coefficient to be changed.
newvalue: Desired new value for coefficient.

GRBModel::chgCoeffs()

Change a list of coefficients in the model. Each desired change is captured using a `GRBVar` object, a `GRBConstr` object, and a desired coefficient for the specified variable in the specified constraint. The entries in the input arrays each correspond to a single desired coefficient change. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel::update), optimize the model (using GRBModel::optimize), or write the model to disk (using GRBModel::write).

```
void  chgCoeffs (  const GRBConstr*  constrs,
                   const GRBVar*     vars,
                   const double*     vals,
                   int               len )
```

Arguments:

constrs: Constraints for coefficients to be changed.
vars: Variables for coefficients to be changed.
vals: Desired new values for coefficients.
len: Number of coefficients to change (length of `vars`, `constrs`, and `vals` arrays).

GRBModel::computeIIS()

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the smallest one; there may exist others with fewer constraints or bounds.

IIS results are returned in a number of attributes: IISConstr, IISLB, IISUB, IISSOS, IISQConstr, and IISGenConstr. Each indicates whether the corresponding model element is a member of the computed IIS.

Note that for models with general function constraints, piecewise-linear approximation of the constraints may cause unreliable IIS results.

The IIS log provides information about the progress of the algorithm, including a guess at the eventual IIS size.

If an IIS computation is interrupted before completion, Gurobi will return the smallest infeasible subsystem found to that point.

The IISConstrForce, IISLBForce, IISUBForce, IISSOSForce, IISQConstrForce, and IISGenConstrForce attributes allow you mark model elements to either include or exclude from the computed IIS. Setting the attribute to 1 forces the corresponding element into the IIS, setting it to 0 forces it out of the IIS, and setting it to -1 allows the algorithm to decide.

To give an example of when these attributes might be useful, consider the case where an initial model is known to be feasible, but it becomes infeasible after adding constraints or tightening bounds. If you are only interested in knowing which of the changes caused the infeasibility, you can force the unmodified bounds and constraints into the IIS. That allows the IIS algorithm to focus exclusively on the new constraints, which will often be substantially faster.

Note that setting any of the `Force` attributes to 0 may make the resulting subsystem feasible, which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

This method populates the IISConstr, IISQConstr, and IISGenConstr constraint attributes, the IISSOS, SOS attribute, and the IISLB and IISUB variable attributes. You can also obtain information about the results of the IIS computation by writing a `.ilp` format file (see GRBModel::write). This file contains only the IIS from the original model.

Use the IISMethod parameter to adjust the behavior of the IIS algorithm.

Note that this method can be used to compute IISs for both continuous and MIP models.

```
void  computeIIS ( )
```

GRBModel::discardConcurrentEnvs()

Discard concurrent environments for a model.

The concurrent environments created by getConcurrentEnv will be used by every subsequent call to the concurrent optimizer until the concurrent environments are discarded.

```
void  discardConcurrentEnvs ( )
```

GRBModel::discardMultiobjEnvs()

Discard all multi-objective environments associated with the model, thus restoring multi objective optimization to its default behavior.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Use getMultiobjEnv to create a multi-objective environment.

```
void  discardMultiobjEnvs ( )
```

GRBModel::feasRelax()

Modifies the `GRBModel` object to create a feasibility relaxation. Note that you need to call optimize on the result to compute the actual relaxed solution.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This method provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost per unit violation in the lower bounds, upper bounds, and linear constraints, respectively.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the coefficients on the squares of the lower bound, upper bound, and linear constraint violations, respectively.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the weighted count of bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost of violating a lower bound, upper bound, and linear constraint, respectively.

To give an example, if a constraint with `rhspen` value `p` is violated by 2.0, it would contribute `2*p` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2*2*p` for `relaxobjtype=1`, and it would contribute `p` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=false`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=true`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `feasRelax` must solve an optimization problem to find the minimum possible relaxation when `minrelax=true`, which can be quite expensive.

There are two signatures for this method. The more complex one takes a list of variables and constraints, as well as penalties associated with relaxing the corresponding lower bounds, upper bounds, and constraints. If a variable or constraint is not included in one of these lists, the associated bounds or constraints may not be violated. The simpler signature takes a pair of boolean arguments, `vrelax` and `crelax`, that indicate whether variable bounds and/or constraints can be violated. If `vrelax`/`crelax` is `true`, then every bound/constraint is allowed to be violated, respectively, and the associated cost is 1.0.

Note that this is a destructive method: it modifies the model on which it is invoked. If you don't want to modify your original model, use the GRBModel constructor to create a copy before invoking this method.

```
double  feasRelax (  int               relaxobjtype,
                     bool              minrelax,
                     int               vlen,
                     const GRBVar*     vars,
                     const double*     lbpen,
                     const double*     ubpen,
                     int               clen,
                     const GRBConstr*  constrs,
                     const double*     rhspen )
```

Create a feasibility relaxation model.

Arguments:

`relaxobjtype`: The cost function used when finding the minimum cost relaxation.

`minrelax`: The type of feasibility relaxation to perform.

`vlen`: The length of the list of variables whose bounds are allowed to be violated.

vars: Variables whose bounds are allowed to be violated.

lbpen: Penalty for violating a variable lower bound. One entry for each variable in argument vars.

ubpen: Penalty for violating a variable upper bound. One entry for each variable in argument vars.

clen: The length of the list of linear constraints that are allowed to be violated.

constrs: Linear constraints that are allowed to be violated.

rhspen: Penalty for violating a linear constraint. One entry for each constraint in argument constrs.

Return value:

Zero if minrelax is false. If minrelax is true, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

```
double  feasRelax (  int   relaxobjtype,
                     bool  minrelax,
                     bool  vrelax,
                     bool  crelax )
```

Simplified method for creating a feasibility relaxation model.

Arguments:

relaxobjtype: The cost function used when finding the minimum cost relaxation.

minrelax: The type of feasibility relaxation to perform.

vrelax: Indicates whether variable bounds can be relaxed (with a cost of 1.0 for any violations.

crelax: Indicates whether linear constraints can be relaxed (with a cost of 1.0 for any violations.

Return value:

Zero if minrelax is false. If minrelax is true, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

GRBModel::fixedModel()

Create the fixed model associated with a MIP model. The MIP model must have a solution loaded (e.g., after a call to the optimize method). In the fixed model, each integer variable is fixed to the value that variable takes in the MIP solution. In addition, continuous variables may be fixed to satisfy SOS or general constraints. The result is a model without any integrality constraints, SOS constraints, or general constraints.

Note that, while the fixed problem is always a continuous model, it may contain a non-convex quadratic objective or non-convex quadratic constraints. As a result, it may still be solved using the MIP algorithm.

```
GRBModel  fixedModel ( )
```

Return value:

Fixed model associated with calling object.

GRBModel::get()

Query the value(s) of a parameter or attribute. Use this method for parameters, for scalar model attributes, or for arrays of constraint or variable attributes.

```
double  get (  GRB_DoubleParam  param )
```

Query the value of a double-valued parameter.
Arguments:
param: The parameter being queried.
Return value:
The current value of the requested parameter.

```
int  get (  GRB_IntParam  param )
```

Query the value of an int-valued parameter.
Arguments:
param: The parameter being queried.
Return value:
The current value of the requested parameter.

```
string  get (  GRB_StringParam  param )
```

Query the value of a string-valued parameter.
Arguments:
param: The parameter being queried.
Return value:
The current value of the requested parameter.

```
char*  get (  GRB_CharAttr   attr,
              const GRBVar*  vars,
              int            count )
```

Query a char-valued variable attribute for an array of variables.
Arguments:
attr: The attribute being queried.
vars: An array of variables whose attribute values are being queried.
count: The number of variable attributes to retrieve.
Return value:
The current values of the requested attribute for each input variable. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
char*  get (  GRB_CharAttr      attr,
              const GRBConstr*  constrs,
              int               count )
```

Query a char-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of constraints whose attribute values are being queried.

count: The number of constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
char*  get (  GRB_CharAttr       attr,
              const GRBQConstr*  qconstrs,
              int                count )
```

Query a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of quadratic constraints whose attribute values are being queried.

count: The number of quadratic constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input quadratic constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
double  get (  GRB_DoubleAttr  attr )
```

Query the value of a double-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
double*  get (  GRB_DoubleAttr  attr,
                const GRBVar*   vars,
                int             count )
```

Query a double-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: An array of variables whose attribute values are being queried.

count: The number of variable attributes to retrieve.

Return value:

The current values of the requested attribute for each input variable. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
double*  get (  GRB_DoubleAttr    attr,
                const GRBConstr*  constrs,
                int               count )
```

Query a double-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of constraints whose attribute values are being queried.

count: The number of constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
double*  get (  GRB_DoubleAttr     attr,
                const GRBQConstr*  qconstrs,
                int                count )
```

Query a double-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of quadratic constraints whose attribute values are being queried.

count: The number of quadratic constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input quadratic constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
int  get (  GRB_IntAttr  attr )
```

Query the value of an int-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
int*  get (  GRB_IntAttr    attr,
             const GRBVar*  vars,
             int            count )
```

Query an int-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: An array of variables whose attribute values are being queried.

count: The number of variable attributes to retrieve.

Return value:

The current values of the requested attribute for each input variable. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
int*  get (  GRB_IntAttr       attr,
             const GRBConstr*  constrs,
             int               count )
```

Query an int-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of constraints whose attribute values are being queried.

count: The number of constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
string  get (  GRB_StringAttr  attr )
```

Query the value of a string-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
string*  get (  GRB_StringAttr  attr,
                const GRBVar*   vars,
                int             count )
```

Query a string-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: An array of variables whose attribute values are being queried.

count: The number of variable attributes to retrieve.

Return value:

The current values of the requested attribute for each input variable. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
string*  get (  GRB_StringAttr    attr,
                const GRBConstr*  constrs,
                int               count )
```

Query a string-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of constraints whose attribute values are being queried.

count: The number of constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
string*  get (  GRB_StringAttr     attr,
                const GRBQConstr*  qconstrs,
                int                count )
```

Query a string-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

`attr`: The attribute being queried.

`constrs`: An array of quadratic constraints whose attribute values are being queried.

`count`: The number of quadratic constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input quadratic constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBModel::getCoeff()

Query the coefficient of variable `var` in linear constraint `constr` (note that the result can be zero).

```
double  getCoeff (  GRBConstr  constr,
                    GRBVar     var )
```

Arguments:

`constr`: The requested constraint.

`var`: The requested variable.

Return value:

The current value of the requested coefficient.

GRBModel::getCol()

Retrieve the list of constraints in which a variable participates, and the associated coefficients. The result is returned as a GRBColumn object.

```
GRBColumn  getCol (  GRBVar  var )
```

Arguments:

`var`: The variable of interest.

Return value:

A GRBColumn object that captures the set of constraints in which the variable participates.

GRBModel::getConcurrentEnv()

Create/retrieve a concurrent environment for a model.

This method provides fine-grained control over the concurrent optimizer. By creating your own concurrent environments and setting appropriate parameters on these environments (e.g., the Method parameter), you can control exactly which strategies the concurrent optimizer employs. For example, if you create two concurrent environments, and set `Method` to primal simplex for one and dual simplex for the other, subsequent concurrent optimizer runs will use the two simplex algorithms rather than the default choices.

Note that you must create contiguously numbered concurrent environments, starting with `num=0`. For example, if you want three concurrent environments, they must be numbered 0, 1, and 2.

Once you create concurrent environments, they will be used for every subsequent concurrent optimization on that model. Use discardConcurrentEnvs to revert back to default concurrent optimizer behavior.

```
GRBEnv  getConcurrentEnv (  int  num )
```

Arguments:

`num`: The concurrent environment number.

Return value:

The concurrent environment for the model.

GRBModel::getConstrByName()

Retrieve a linear constraint from its name. If multiple linear constraints have the same name, this method chooses one arbitrarily.

```
GRBConstr  getConstrByName (  const string&  name )
```

Arguments:

`name`: The name of the desired linear constraint.

Return value:

The requested linear constraint.

GRBModel::getConstrs()

Retrieve an array of all linear constraints in the model.

```
GRBConstr*  getConstrs ( )
```

Return value:

An array of all linear constraints in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::getGenConstrXxx()

The following methods allow you to retrieve general constraints from your model.

GRBModel::getGenConstrMax

Retrieve the data associated with a general constraint of type MAX. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `NULL` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `lenP`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also addGenConstrMax for a description of the semantics of this general constraint type.

```
void  getGenConstrMax (  GRBGenConstr  genc,
                         GRBVar*       resvarP,
                         GRBVar*       vars,
                         int*          lenP,
                         double*       constantP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

resvarP: Pointer to store the resultant variable of the constraint.

vars: Array to store the operand variables of the constraint.

lenP: Pointer to store the number of operand variables of the constraint.

constantP: Pointer to store the additional constant operand of the constraint.

GRBModel::getGenConstrMin

Retrieve the data associated with a general constraint of type MIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a NULL value for the vars argument. The routine returns the total number of operand variables in the specified general constraint in lenP. That allows you to make certain that the vars array is of sufficient size to hold the result of the second call.

See also addGenConstrMin for a description of the semantics of this general constraint type.

```
void  getGenConstrMin (  GRBGenConstr  genc,
                         GRBVar*       resvarP,
                         GRBVar*       vars,
                         int*          lenP,
                         double*       constantP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

resvarP: Pointer to store the resultant variable of the constraint.

vars: Array to store the operand variables of the constraint.

lenP: Pointer to store the number of operand variables of the constraint.

constantP: Pointer to store the additional constant operand of the constraint.

GRBModel::getGenConstrAbs

Retrieve the data associated with a general constraint of type ABS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrAbs for a description of the semantics of this general constraint type.

```
void  getGenConstrAbs (  GRBGenConstr  genc,
                         GRBVar*       resvarP,
                         GRBVar*       argvarP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

resvarP: Pointer to store the resultant variable of the constraint.

argvarP: Pointer to store the argument variable of the constraint.

GRBModel::getGenConstrAnd

Retrieve the data associated with a general constraint of type AND. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a NULL value for the vars argument. The routine returns the total number of operand variables in the specified general constraint in lenP. That allows you to make certain that the vars array is of sufficient size to hold the result of the second call.

See also addGenConstrAnd for a description of the semantics of this general constraint type.

```
void  getGenConstrAnd (  GRBGenConstr  genc,
                         GRBVar*       resvarP,
                         GRBVar*       vars,
                         int*          lenP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

resvarP: Pointer to store the resultant variable of the constraint.

vars: Array to store the operand variables of the constraint.

lenP: Pointer to store the number of operand variables of the constraint.

GRBModel::getGenConstrOr

Retrieve the data associated with a general constraint of type OR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a NULL value for the vars argument. The routine returns the total number of operand variables in the specified general constraint in lenP. That allows you to make certain that the vars array is of sufficient size to hold the result of the second call.

See also addGenConstrOr for a description of the semantics of this general constraint type.

```
void  getGenConstrOr (  GRBGenConstr  genc,
                        GRBVar*       resvarP,
                        GRBVar*       vars,
                        int*          lenP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

resvarP: Pointer to store the resultant variable of the constraint.

vars: Array to store the operand variables of the constraint.

lenP: Pointer to store the number of operand variables of the constraint.

GRBModel::getGenConstrNorm

Retrieve the data associated with a general constraint of type NORM. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a NULL value for the vars argument. The routine returns the total number of operand variables in the specified general constraint in lenP. That allows you to make certain that the vars array is of sufficient size to hold the result of the second call.

See also addGenConstrNorm for a description of the semantics of this general constraint type.

```
void  getGenConstrNorm (  GRBGenConstr  genc,
                          GRBVar*       resvarP,
                          GRBVar*       vars,
                          int*          lenP,
                          double*       whichP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

resvarP: Pointer to store the resultant variable of the constraint.

vars: Array to store the operand variables of the constraint.

lenP: Pointer to store the number of operand variables of the constraint.

whichP: Pointer to store the norm type (possible values are 0, 1, 2, or GRB_INFINITY).

GRBModel::getGenConstrIndicator

Retrieve the data associated with a general constraint of type INDICATOR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrIndicator for a description of the semantics of this general constraint type.

```
void  getGenConstrIndicator (  GRBGenConstr  genc,
                               GRBVar*       binvarP,
                               int*          binvalP,
                               GRBLinExpr*   exprP,
                               char*         senseP,
                               double*       rhsP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

binvarP: Pointer to store the binary indicator variable of the constraint.

binvalP: Pointer to store the value that the indicator variable has to take in order to trigger the linear constraint.

exprP: Pointer to a GRBLinExpr object to store the left-hand side expression of the linear constraint that is triggered by the indicator.

senseP: Pointer to store the sense for the linear constraint. Options are GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL.

rhsP: Pointer to store the right-hand side value for the linear constraint.

GRBModel::getGenConstrPWL

Retrieve the data associated with a general constraint of type PWL. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a NULL value for the **xpts** and **ypts** arguments. The routine returns the length for the **xpts** and **ypts** arrays in **nptsP**. That allows you to make certain that the **xpts** and **ypts** arrays are of sufficient size to hold the result of the second call.

See also addGenConstrPWL for a description of the semantics of this general constraint type.

```
void  getGenConstrPWL (  GRBGenConstr  genc,
                         GRBVar*       xvarP,
                         GRBVar*       yvarP,
                         int*          nptsP,
                         double*       xpts,
                         double*       ypts )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

nptsP: Pointer to store the number of points that define the piecewise-linear function.

xpts: The x values for the points that define the piecewise-linear function.

ypts: The y values for the points that define the piecewise-linear function.

GRBModel::getGenConstrPoly

Retrieve the data associated with a general constraint of type POLY. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a NULL value for the **p** argument. The routine returns the length of the **p** array in

plenP. That allows you to make certain that the p array is of sufficient size to hold the result of the second call.

See also addGenConstrPoly for a description of the semantics of this general constraint type.

```
void  getGenConstrPoly (  GRBGenConstr  genc,
                          GRBVar*       xvarP,
                          GRBVar*       yvarP,
                          int*          plenP,
                          double*       p )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

plenP: Pointer to store the array length for p. If x^d is the highest power term, then $d+1$ will be returned.

p: The coefficients for polynomial function.

GRBModel::getGenConstrExp

Retrieve the data associated with a general constraint of type EXP. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrExp for a description of the semantics of this general constraint type.

```
void  getGenConstrExp (  GRBGenConstr  genc,
                         GRBVar*       xvarP,
                         GRBVar*       yvarP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

GRBModel::getGenConstrExpA

Retrieve the data associated with a general constraint of type EXPA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrExpA for a description of the semantics of this general constraint type.

```
void  getGenConstrExpA (  GRBGenConstr  genc,
                          GRBVar*       xvarP,
                          GRBVar*       yvarP,
                          double*       aP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

aP: Pointer to store the base of the function.

GRBModel::getGenConstrLog

Retrieve the data associated with a general constraint of type LOG. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLog for a description of the semantics of this general constraint type.

```
void  getGenConstrLog (  GRBGenConstr  genc,
                         GRBVar*       xvarP,
                         GRBVar*       yvarP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

GRBModel::getGenConstrLogA

Retrieve the data associated with a general constraint of type LOGA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLogA for a description of the semantics of this general constraint type.

```
void  getGenConstrLogA (  GRBGenConstr  genc,
                          GRBVar*       xvarP,
                          GRBVar*       yvarP,
                          double*       aP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

aP: Pointer to store the base of the function.

GRBModel::getGenConstrLogistic

Retrieve the data associated with a general constraint of type LOGISTIC. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLogistic for a description of the semantics of this general constraint type.

```
void  getGenConstrLogistic (  GRBGenConstr  genc,
                              GRBVar*       xvarP,
                              GRBVar*       yvarP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

GRBModel::getGenConstrPow

Retrieve the data associated with a general constraint of type POW. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrPow for a description of the semantics of this general constraint type.

```
void  getGenConstrPow (  GRBGenConstr  genc,
                         GRBVar*       xvarP,
                         GRBVar*       yvarP,
                         double*       aP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

aP: Pointer to store the exponent of the function.

GRBModel::getGenConstrSin

Retrieve the data associated with a general constraint of type SIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrSin for a description of the semantics of this general constraint type.

```
void  getGenConstrSin (  GRBGenConstr  genc,
                         GRBVar*       xvarP,
                         GRBVar*       yvarP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

GRBModel::getGenConstrCos

Retrieve the data associated with a general constraint of type COS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrCos for a description of the semantics of this general constraint type.

```
void  getGenConstrCos (  GRBGenConstr  genc,
                         GRBVar*       xvarP,
                         GRBVar*       yvarP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

GRBModel::getGenConstrTan

Retrieve the data associated with a general constraint of type TAN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrTan for a description of the semantics of this general constraint type.

```
void  getGenConstrTan (  GRBGenConstr  genc,
                         GRBVar*       xvarP,
                         GRBVar*       yvarP )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be NULL.

xvarP: Pointer to store the x variable.

yvarP: Pointer to store the y variable.

GRBModel::getGenConstrs()

Retrieve an array of all general constraints in the model.

```
GRBGenConstr*  getGenConstrs ( )
```

Return value:

An array of all general constraints in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::getJSONSolution()

After a call to optimize, this method returns the resulting solution and related model attributes as a JSON string. Please refer to the JSON solution format section for details.

```
getJSONSolution  GRBModel ( )
```

Return value:

A JSON string.

GRBModel::getMultiobjEnv()

Create/retrieve a multi-objective environment for the optimization pass with the given index. This environment enables fine-grained control over the multi-objective optimization process. Specifically, by changing parameters on this environment, you modify the behavior of the optimization that occurs during the corresponding pass of the multi-objective optimization.

Each multi-objective environment starts with a copy of the current model environment.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Please refer to the discussion on Combining Blended and Hierarchical Objectives for information on the optimization passes to solve multi-objective models.

Use discardMultiobjEnvs to discard multi-objective environments and return to standard behavior.

```
GRBEnv  getMultiobjEnv (  int  index )
```

Arguments:

index: The optimization pass index, starting from 0.

Return value:

The multi-objective environment for that optimization pass when solving the model.

GRBModel::getObjective()

Retrieve the optimization objective(s).

```
GRBQuadExpr  getObjective ( )
```

Retrieve the optimization objective.

Note that the constant and linear portions of the objective can also be retrieved using the ObjCon and Obj attributes.

Return value:

The model objective.

```
GRBLinExpr  getObjective (  int  index )
```

Retrieve an alternative optimization objective. Alternative objectives will always be linear. You can also use this routine to retrieve the primary objective (using `index` = 0), but you will get an exception if the primary objective contains quadratic terms.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

Note that alternative objectives can also be retrieved using the ObjNCon and ObjN attributes.

Arguments:

`index`: The index for the requested alternative objective.

Return value:

The requested alternate objective.

GRBModel::getPWLObj()

Retrieve the piecewise-linear objective function for a variable. The return value gives the number of points that define the function, and the x and y arguments give the coordinates of the points, respectively. The x and y arguments must be large enough to hold the result. Call this method with `NULL` values for x and y if you just want the number of points.

Refer to this discussion for additional information on what the values in x and y mean.

```
int  getPWLObj ( GRBVar    var,
                 double[]  x,
                 double[]  y )
```

Arguments:

`var`: The variable whose objective function is being retrieved.

`x`: The x values for the points that define the piecewise-linear function. These will always be in non-decreasing order.

`y`: The y values for the points that define the piecewise-linear function.

Return value:

The number of points that define the piecewise-linear objective function.

GRBModel::getQCRow()

Retrieve the left-hand side expression from a quadratic constraint. The result is returned as a GRBQuadExpr object.

```
GRBQuadExpr  getQCRow ( GRBQConstr  qconstr )
```

Arguments:

`qconstr`: The quadratic constraint of interest.

Return value:

A GRBQuadExpr object that captures the left-hand side of the quadratic constraint.

GRBModel::getQConstrs()

Retrieve an array of all quadratic constraints in the model.

```
GRBQConstr*  getQConstrs ( )
```

Return value:

An array of all quadratic constraints in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::getRow()

Retrieve a list of variables that participate in a constraint, and the associated coefficients. The result is returned as a GRBLinExpr object.

```
GRBLinExpr  getRow (  GRBConstr  constr )
```

Arguments:

constr: The constraint of interest. A GRBConstr object, typically obtained from addConstr or getConstrs.

Return value:

A GRBLinExpr object that captures the set of variables that participate in the constraint.

GRBModel::getSOS()

Retrieve the list of variables that participate in an SOS constraint, and the associated coefficients. The return value is the length of this list. If you would like to allocate space for the result before retrieving the result, call the method first with NULL array arguments to determine the appropriate array lengths.

```
int  getSOS (  GRBSOS   sos,
               GRBVar*  vars,
               double*  weights,
               int*     typeP )
```

Arguments:

sos: The SOS set of interest.

vars: A list of variables that participate in **sos**.

weights: The SOS weights for each participating variable.

typeP: The type of the SOS set (either GRB_SOS_TYPE1 or GRB_SOS_TYPE2).

Return value:

The length of the result arrays.

GRBModel::getSOSs()

Retrieve an array of all SOS constraints in the model.

```
GRBSOS*  getSOSs ( )
```

Return value:

An array of all SOS constraints in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::getTuneResult()

Use this method to retrieve the results of a previous tune call. Calling this method with argument `n` causes tuned parameter set `n` to be copied into the model. Parameter sets are stored in order of decreasing quality, with parameter set 0 being the best. The number of available sets is stored in attribute TuneResultCount.

Once you have retrieved a tuning result, you can call optimize to use these parameter settings to optimize the model, or write to write the changed parameters to a `.prm` file.

Please refer to the parameter tuning section for details on the tuning tool.

```
void  getTuneResult (  int  n )
```

`n`: The index of the tuning result to retrieve. The best result is available as index 0. The number of stored results is available in attribute TuneResultCount.

GRBModel::getVarByName()

Retrieve a variable from its name. If multiple variables have the same name, this method chooses one arbitrarily.

```
GRBVar  getVarByName (  const string&  name )
```

Arguments:

`name`: The name of the desired variable.

Return value:

The requested variable.

GRBModel::getVars()

Retrieve an array of all variables in the model.

```
GRBVar*  getVars ( )
```

Return value:

An array of all variables in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::optimize()

Optimize the model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion, this method will populate the solution related attributes of the model. See the Attributes section for more information on attributes.

Please consult this section for a discussion of some of the practical issues associated with solving a precisely defined mathematical model using finite-precision floating-point arithmetic.

Note that this method will process all pending model modifications.

```
void  optimize ( )
```

GRBModel::optimizeasync()

Optimize a model asynchronously. This routine returns immediately. Your program can perform other computations while optimization proceeds in the background. To check the state of the asynchronous optimization, query the Status attribute for the model. A value of `IN_PROGRESS` indicates that the optimization has not yet completed. When you are done with your foreground tasks, you must call sync to sync your foreground program with the asynchronous optimization task.

Note that the set of Gurobi calls that you are allowed to make while optimization is running in the background is severely limited. Specifically, you can only perform attribute queries, and only for a few attributes (listed below). Any other calls on the running model, *or on any other models that were built within the same Gurobi environment*, will fail with error code `OPTIMIZATION_IN_PROGRESS`.

Note that there are no such restrictions on models built in other environments. Thus, for example, you could create multiple environments, and then have a single foreground program launch multiple simultaneous asynchronous optimizations, each in its own environment.

As already noted, you are allowed to query the value of the `Status` attribute while an asynchronous optimization is in progress. The other attributes that can be queried are: ObjVal, ObjBound, IterCount, NodeCount, and BarIterCount. In each case, the returned value reflects progress in the optimization to that point. Any attempt to query the value of an attribute not on this list will return an `OPTIMIZATION_IN_PROGRESS` error.

```
void  optimizeasync ( )
```

GRBModel::optimizeBatch()

Submit a new batch request to the Cluster Manager. Returns the `BatchID` (a string), which uniquely identifies the job in the Cluster Manager and can be used to query the status of this request (from this program or from any other). Once the request has completed, the `BatchID` can also be used to retrieve the associated solution. To submit a batch request, you must tag at least one element of the model by setting one of the VTag, CTag or QCTag attributes. For more details on batch optimization, please refer to the Batch Optimization section.

Note that this routine will process all pending model modifications.

```
string  optimizeBatch ( )
```

Example usage:

```
// submit batch request
batchID = model->optimizeBatch();
```

GRBModel::presolve()

Perform presolve on a model.

Please note that the presolved model computed by this function may be different from the presolved model computed when optimizing the model.

```
GRBModel  presolve ( )
```

Return value:

Presolved version of original model.

GRBModel::read()

This method is the general entry point for importing data from a file into a model. It can be used to read basis files for continuous models, start vectors for MIP models, or parameter settings. The type of data read is determined by the file suffix. File formats are described in the File Format section.

Note that reading a file does **not** process all pending model modifications. These modifications can be processed by calling GRBModel::update.

Note also that this is **not** the method to use if you want to read a new model from a file. For that, use the GRBModel constructor. One variant of the constructor takes the name of the file that contains the new model as its argument.

```
void  read ( const string&  filename )
```

Arguments:

`filename`: Name of the file to read. The suffix on the file must be either `.bas` (for an LP basis), `.mst` or `.sol` (for a MIP start), `.hnt` (for MIP hints), `.ord` (for a priority order), or `.prm` (for a parameter file). The suffix may optionally be followed by `.zip`, `.gz`, `.bz2`, or `.7z`.

GRBModel::remove()

Remove a variable, constraint, or SOS from a model.

```
void  remove ( GRBConstr  constr )
```

Remove a linear constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel::update), optimize the model (using GRBModel::optimize), or write the model to disk (using GRBModel::write).

Arguments:

constr: The linear constraint to remove.

```
void  remove ( GRBGenConstr  genconstr )
```

Remove a general constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel::update), optimize the model (using GRBModel::optimize), or write the model to disk (using GRBModel::write).

Arguments:

genconstr: The general constraint to remove.

```
void  remove ( GRBQConstr  qconstr )
```

Remove a quadratic constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel::update), optimize the model (using GRBModel::optimize), or write the model to disk (using GRBModel::write).

Arguments:

qconstr: The quadratic constraint to remove.

```
void  remove ( GRBSOS  sos )
```

Remove an SOS constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel::update), optimize the model (using GRBModel::optimize), or write the model to disk (using GRBModel::write).

Arguments:

sos: The SOS constraint to remove.

```
void  remove ( GRBVar  var )
```

Remove a variable from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel::update), optimize the model (using GRBModel::optimize), or write the model to disk (using GRBModel::write).

Arguments:

var: The variable to remove.

GRBModel::reset()

```
void  reset ( int  clearall=0 )
```

Reset the model to an unsolved state, discarding any previously computed solution information. Note that, due to our lazy update approach, the change won't actually take effect until you update

the model (using GRBModel::update), optimize the model (using GRBModel::optimize), or write the model to disk (using GRBModel::write).

clearall (optional): A value of 1 discards additional information that affects the solution process but not the actual model (currently MIP starts, variable hints, branching priorities, lazy flags, and partition information). The default value of 0 just discards the solution.

Arguments:

GRBModel::setCallback()

Set the callback object for a model. The `callback()` method on this object will be called periodically from the Gurobi solver. You will have the opportunity to obtain more detailed information about the state of the optimization from this callback. See the documentation for GRBCallback for additional information.

Note that a model can only have a single callback method, so this call will replace an existing callback. To disable a previously set callback, call this method with a `NULL` argument.

```
void  setCallback ( GRBCallback*  cb )
```

GRBModel::set()

Set the value(s) of a parameter or attribute. Use this method for parameters, for scalar model attributes, and for arrays of constraint or variable attributes.

```
void  set ( GRB_DoubleParam  param,
            double           newvalue )
```

Set the value of a double-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv::set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newvalue: The desired new value for the parameter.

```
void  set ( GRB_IntParam  param,
            int           newvalue )
```

Set the value of an int-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv::set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newvalue: The desired new value for the parameter.

```
void  set (  GRB_StringParam  param,
             string           newvalue )
```

Set the value of a string-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv::set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newvalue: The desired new value for the parameter.

```
void  set (  GRB_CharAttr   attr,
             const GRBVar*  vars,
             char*          newvalues,
             int            count )
```

Set a char-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: An array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

count: The number of variable attributes to set.

```
void  set (  GRB_CharAttr      attr,
             const GRBConstr*  constrs,
             char*             newvalues,
             int               count )
```

Set a char-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

count: The number of constraint attributes to set.

```
void  set (  GRB_CharAttr       attr,
             const GRBQConstr*  qconstrs,
             char*              newvalues,
             int                count )
```

Set a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

count: The number of quadratic constraint attributes to set.

```
void  set (  GRB_DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued model attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value for the attribute.

```
void  set (  GRB_DoubleAttr  attr,
             const GRBVar*   vars,
             double*         newvalues,
             int             count )
```

Set a double-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: An array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

count: The number of variable attributes to set.

```
void  set (  GRB_DoubleAttr    attr,
             const GRBConstr*  constrs,
             double*           newvalues,
             int               count )
```

Set a double-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

count: The number of constraint attributes to set.

```
void  set (  GRB_DoubleAttr     attr,
             const GRBQConstr*  qconstrs,
             double*            newvalues,
             int                count )
```

Set a double-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

count: The number of quadratic constraint attributes to set.

```
void  set (  GRB_IntAttr  attr,
             int          newvalue )
```

Set the value of an int-valued model attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value for the attribute.

```
void  set (  GRB_IntAttr    attr,
             const GRBVar*  vars,
             int*           newvalues,
             int            count )
```

Set an int-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: An array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

count: The number of variable attributes to set.

```
void  set (  GRB_IntAttr       attr,
             const GRBConstr*  constrs,
             int*              newvalues,
             int               count )
```

Set an int-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

count: The number of constraint attributes to set.

```
void  set (  GRB_StringAttr  attr,
             string          newvalue )
```

Set the value of a string-valued model attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value for the attribute.

```
void  set (  GRB_StringAttr  attr,
             const GRBVar*   vars,
             string*         newvalues,
             int             count )
```

Set a string-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: An array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

count: The number of variable attributes to set.

```
void  set (  GRB_StringAttr    attr,
             const GRBConstr*  constrs,
             string*           newvalues,
             int               count )
```

Set a string-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

count: The number of constraint attributes to set.

```
void  set (  GRB_StringAttr     attr,
             const GRBQConstr*  qconstrs,
             string*            newvalues,
             int                count )
```

Set a string-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

count: The number of quadratic constraint attributes to set.

GRBModel::setObjective()

Set the model objective equal to a linear or quadratic expression (for multi-objective optimization, see setObjectiveN).

Note that you can also modify the linear portion of a model objective using the Obj variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `Obj` attribute can be used to modify individual linear terms.

```
void  setObjective (  GRBLinExpr  linexpr,
                      int         sense=0 )
```

Arguments:

linexpr: New linear model objective.

sense (optional): Optimization sense (`GRB_MINIMIZE` for minimization, `GRB_MAXIMIZE` for maximization). Omit this argument to use the `ModelSense` attribute value to determine the sense.

```
void  setObjective (  GRBQuadExpr  quadexpr,
                      int          sense=0 )
```

Arguments:

quadexpr: New quadratic model objective.

sense (optional): Optimization sense (`GRB_MINIMIZE` for minimization, `GRB_MAXIMIZE` for maximization). Omit this argument to use the `ModelSense` attribute value.

GRBModel::setObjectiveN()

```
void  setObjectiveN (  GRBLinExpr  expr,
                       int         index,
                       int         priority=0,
                       double      weight=1,
                       double      abstol=0,
                       double      reltol=0,
                       string      name="" )
```

Set an alternative optimization objective equal to a linear expression.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

Note that you can also modify an alternative objective using the ObjN variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `ObjN` attribute can be used to modify individual terms.

Arguments:

`expr`: New alternative objective.

`index`: Index for new objective. If you use an index of 0, this routine will change the primary optimization objective.

`priority`: Priority for the alternative objective. This initializes the ObjNPriority attribute for this objective.

`weight`: Weight for the alternative objective. This initializes the ObjNWeight attribute for this objective.

`abstol`: Absolute tolerance for the alternative objective. This initializes the ObjNAbsTol attribute for this objective.

`reltol`: Relative tolerance for the alternative objective. This initializes the ObjNRelTol attribute for this objective.

`name`: Name of the alternative objective. This initializes the ObjNName attribute for this objective.

GRBModel::setPWLObj()

Set a piecewise-linear objective function for a variable.

The arguments to this method specify a list of points that define a piecewise-linear objective function for a single variable. Specifically, the x and y arguments give coordinates for the vertices of the function.

For additional details on piecewise-linear objective functions, refer to this discussion.

```
void  setPWLObj (  GRBvar    var,
                   int       npoints,
                   double[]  x,
                   double[]  y )
```

Set the piecewise-linear objective function for a variable.

Arguments:

var: The variable whose objective function is being set.

npoints: Number of points that define the piecewise-linear function.

x: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

y: The y values for the points that define the piecewise-linear function.

GRBModel::singleScenarioModel()

Capture a single scenario from a multi-scenario model. Use the ScenarioNumber parameter to indicate which scenario to capture.

The model on which this method is invoked must be a multi-scenario model, and the result will be a single-scenario model.

```
GRBModel  singleScenarioModel ( )
```

Return value:

Model for a single scenario.

GRBModel::sync()

Wait for a previous asynchronous optimization call to complete.

Calling optimizeasync returns control to the calling routine immediately. The caller can perform other computations while optimization proceeds, and can check on the progress of the optimization by querying various model attributes. The `sync` call forces the calling program to wait until the asynchronous optimization call completes. You *must* call `sync` before the corresponding model object is deleted.

The `sync` call throws an exception if the optimization itself ran into any problems. In other words, exceptions thrown by this method are those that `optimize` itself would have thrown, had the original method not been asynchronous.

Note that you need to call `sync` even if you know that the asynchronous optimization has already completed.

```
void  sync ( )
```

GRBModel::terminate()

Generate a request to terminate the current optimization. This method can be called at any time during an optimization (from a callback, from another thread, from an interrupt handler, etc.). Note that, in general, the request won't be acted upon immediately.

When the optimization stops, the Status attribute will be equal to `GRB_INTERRUPTED`.

```
void  terminate ( )
```

GRBModel::tune()

Perform an automated search for parameter settings that improve performance. Upon completion, this method stores the best parameter sets it found. The number of stored parameter sets can be determined by querying the value of the TuneResultCount attribute. The actual settings can be retrieved using getTuneResult.

Please refer to the parameter tuning section for details on the tuning tool.

```
void  tune ( )
```

GRBModel::update()

Process any pending model modifications.

```
void  update ( )
```

GRBModel::write()

This method is the general entry point for writing optimization data to a file. It can be used to write optimization models, solutions vectors, basis vectors, start vectors, or parameter settings. The type of data written is determined by the file suffix. File formats are described in the File Format section.

Note that writing a model to a file will process all pending model modifications. This is also true when writing other model information such as solutions, bases, etc.

Note also that when you write a Gurobi parameter file (PRM), both integer or double parameters not at their default value will be saved, but no string parameter will be saved into the file.

```
void  write (  const string&  filename )
```

Arguments:

`filename`: The name of the file to be written. The file type is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, or `.rlp` for writing the model itself, `.dua` or `.dlp` for writing the dualized model (only pure LP), `.ilp` for writing just the IIS associated with an infeasible model (see GRBModel::computeIIS for further information), `.sol` for writing the solution selected by the SolutionNumber parameter, `.mst` for writing a start vector, `.hnt` for writing a hint file, `.bas` for writing an LP basis, `.prm` for writing modified parameter settings, `.attr` for writing model attributes, or `.json` for writing solution information in JSON format. If your system has compression utilities installed (e.g., `7z` or `zip` for Windows, and `gzip`, `bzip2`, or `unzip` for Linux or macOS), then the files can be compressed, so additional suffixes of `.gz`, `.bz2`, or `.7z` are accepted.

4.3 GRBVar

Gurobi variable object. Variables are always associated with a particular model. You create a variable object by adding a variable to a model (using GRBModel::addVar), rather than by using a `GRBVar` constructor.

The methods on variable objects are used to get and set variable attributes. For example, solution information can be queried by calling get(`GRB_DoubleAttr_X`). Note that you can also query attributes for a set of variables at once. This is done using the attribute query method on the `GRBModel` object (GRBModel::get).

GRBVar::get()

Query the value of a variable attribute.

```
char  get (  GRB_CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get (  GRB_DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get (  GRB_IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
string  get (  GRB_StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBVar::index()

```
int  index ( )
```

This method returns the current index, or order, of the variable in the underlying constraint matrix.

Return value:

-2: removed, -1: not in model, otherwise: index of the variable in the model

Note that the index of a variable may change after subsequent model modifications.

GRBVar::sameAs()

```
bool  sameAs (  GRBVar  var2 )
```

Check whether two variable objects refer to the same variable.

Arguments:

var2: The other variable.

Return value:

Boolean result indicates whether the two variable objects refer to the same model variable.

GRBVar::set()

Set the value of a variable attribute.

```
void  set (  GRB_CharAttr  attr,
             char          newvalue )
```

Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_IntAttr  attr,
             int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_StringAttr  attr,
             const string&   newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

4.4 GRBConstr

Gurobi constraint object. Constraints are always associated with a particular model. You create a constraint object by adding a constraint to a model (using GRBModel::addConstr), rather than by using a `GRBConstr` constructor.

The methods on constraint objects are used to get and set constraint attributes. For example, constraint right-hand sides can be queried by calling get(`GRB_DoubleAttr_RHS`). Note that you can also query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel::get).

GRBConstr::get()

Query the value of a constraint attribute.

```
char  get (  GRB_CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get (  GRB_DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get (  GRB_IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
string  get (  GRB_StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBConstr::index()

```
int  index ( )
```

This method returns the current index, or order, of the constraint in the underlying constraint matrix.

Note that the index of a constraint may change after subsequent model modifications.

Return value:

-2: removed, -1: not in model, otherwise: index of the constraint in the model

GRBConstr::sameAs()

```
bool  sameAs ( GRBConstr  constr2 )
```

Check whether two constraint objects refer to the same constraint.

Arguments:

constr2: The other constraint.

Return value:

Boolean result indicates whether the two constraint objects refer to the same model constraint.

GRBConstr::set()

Set the value of a constraint attribute.

```
void  set ( GRB_CharAttr  attr,
            char          newvalue )
```

Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set ( GRB_DoubleAttr  attr,
            double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set ( GRB_IntAttr  attr,
            int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.
newvalue: The desired new value of the attribute.

```
void  set (  GRB_StringAttr  attr,
             const string&   newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.
newvalue: The desired new value of the attribute.

4.5 GRBQConstr

Gurobi quadratic constraint object. Quadratic constraints are always associated with a particular model. You create a quadratic constraint object by adding a constraint to a model (using GRBModel::addQConstr), rather than by using a `GRBQConstr` constructor.

The methods on quadratic constraint objects are used to get and set quadratic constraint attributes. For example, quadratic constraint right-hand sides can be queried by calling get(`GRB_DoubleAttr_QCRHS`). Note, however, that it is generally more efficient to query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel::get).

GRBQConstr::get()

Query the value of a quadratic constraint attribute.

```
char  get (  GRB_CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get (  GRB_DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get (  GRB_IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
string  get (  GRB_StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBQConstr::set()

Set the value of a quadratic constraint attribute.

```
void  set (  GRB_CharAttr  attr,
             char          newvalue )
```

Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_IntAttr  attr,
             int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_StringAttr  attr,
             const string&   newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

4.6 GRBSOS

Gurobi SOS constraint object. SOS constraints are always associated with a particular model. You create an SOS object by adding an SOS constraint to a model (using GRBModel::addSOS), rather than by using a `GRBSOS` constructor. Similarly, SOS constraints are removed using the GRBModel::remove method.

An SOS constraint can be of type 1 or 2 (`GRB_SOS_TYPE1` or `GRB_SOS_TYPE2`). A type 1 SOS constraint is a set of variables for which at most one variable in the set may take a value other than zero. A type 2 SOS constraint is an ordered set of variables where at most two variables in the set may take non-zero values. If two take non-zero values, they must be contiguous in the ordered set.

SOS constraint objects have one attribute, `IISSOS`, which can be queried with the GRBSOS::get method.

GRBSOS::get()

Query the value of an SOS attribute.

```
int  get ( GRB_IntAttr  attr )
```

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBSOS::set()

Set the value of an SOS attribute.

```
void  set ( GRB_IntAttr  attr,
            int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

4.7 GRBGenConstr

Gurobi general constraint object. General constraints are always associated with a particular model. You create a general constraint object by adding a constraint to a model (using one of the GRBModel::addGenConstrXxx methods), rather than by using a `GRBGenConstr` constructor.

The methods on general constraint objects are used to get and set general constraint attributes. For example, general constraint types can be queried by calling get(`GRB_IntAttr_GenConstrType`). Note, however, that it is generally more efficient to query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel::get).

GRBGenConstr::get()

Query the value of a general constraint attribute.

```
double  get (  GRB_DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get (  GRB_IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
string  get (  GRB_StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBGenConstr::set()

Set the value of a general constraint attribute.

```
void  set (  GRB_DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_IntAttr  attr,
             int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_StringAttr  attr,
             const string&   newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

4.8 GRBExpr

Abstract base class for the GRBLinExpr and GRBQuadExpr classes. Expressions are used to build objectives and constraints. They are temporary objects that typically have short lifespans.

GRBExpr::getValue()

Compute the value of an expression for the current solution.

```
double  getValue ( )
```

Return value:

Value of the expression for the current solution.

4.9 GRBLinExpr

Gurobi linear expression object. A linear expression consists of a constant term, plus a list of coefficient-variable pairs that capture the linear terms. Linear expressions are used to build constraints. They are temporary objects that typically have short lifespans.

The `GRBLinExpr` class is a sub-class of the abstract base class GRBExpr.

You generally build linear expressions using overloaded operators. For example, if `x` is a GRBVar object, then `x + 1` is a GRBLinExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x + 2 * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x`, or `expr2 -= expr1`).

Another option for building expressions is to use the addTerms method, which adds an array of new terms at once. Terms can also be removed from an expression, using remove.

Note that the cost of building expressions depends heavily on the approach you use. While you can generally ignore this issue when building small expressions, you should be aware of a few efficiency issues when building large expressions:

- You should avoid using `expr = expr + x` in a loop. It will lead to runtimes that are quadratic in the number of terms in the expression.
- Using `expr += x` (or `expr -= x`) is much more efficient than `expr = expr + x`. Building a large expression by looping over `+=` statements is reasonably efficient, but it isn't the most efficient approach.
- The most efficient way to build a large expression is to make a single call to addTerms.

To add a linear constraint to your model, you generally build one or two linear expression objects (`expr1` and `expr2`) and then use an overloaded comparison operator to build an argument for GRBModel::addConstr. To give a few examples:

$$\text{model.addConstr}(expr1 <= expr2)$$
$$\text{model.addConstr}(expr1 == 1)$$
$$\text{model.addConstr}(2 * x + 3 * y <= 4)$$

Once you add a constraint to your model, subsequent changes to the expression object you used to build the constraint will not change the constraint (you would use GRBModel::chgCoeff for that).

Individual terms in a linear expression can be queried using the getVar, getCoeff, and getConstant methods. You can query the number of terms in the expression using the size method.

Note that a linear expression may contain multiple terms that involve the same variable. These duplicate terms are merged when creating a constraint from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar).

GRBLinExpr()

Linear expression constructor. Create a constant expression or an expression with one term.

```
GRBLinExpr  GRBLinExpr (  double  constant=0.0 )
```

Create a constant linear expression.

Arguments:

`constant (optional)`: Constant value for expression.

Return value:

A constant expression object.

```
GRBLinExpr  GRBLinExpr (  GRBVar  var,
                          double  coeff=1.0 )
```

Create an expression with one term.

Arguments:

`var`: Variable for expression term.

`coeff (optional)`: Coefficient for expression term.

Return value:

An expression object containing one linear term.

GRBLinExpr::addTerms()

Add new terms into a linear expression.

```
void  addTerms (  const double*  coeffs,
                  const GRBVar*  vars,
                  int            count )
```

Arguments:

`coeffs`: Coefficients for new terms.

`vars`: Variables for new terms.

`count`: Number of terms to add to the expression.

GRBLinExpr::clear()

Set a linear expression to 0.

You should use the overloaded `expr = 0` instead. The `clear` method is mainly included for consistency with our interfaces to non-overloaded languages.

```
void  clear ( )
```

GRBLinExpr::getConstant()

Retrieve the constant term from a linear expression.

```
double  getConstant ( )
```

Return value:

Constant from expression.

GRBLinExpr::getCoeff()

Retrieve the coefficient from a single term of the expression.

```
double  getCoeff (  int  i )
```

Arguments:

`i`: Index for coefficient of interest.

Return value:

Coefficient for the term at index `i` in the expression.

GRBLinExpr::getValue()

Compute the value of a linear expression for the current solution.

```
double  getValue ( )
```

Return value:

Value of the expression for the current solution.

GRBLinExpr::getVar()

Retrieve the variable object from a single term of the expression.

```
GRBVar  getVar (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

Variable for the term at index `i` in the expression.

GRBLinExpr::operator=

Set an expression equal to another expression.

```
GRBLinExpr  operator= (  const GRBLinExpr&  rhs )
```

Arguments:

`rhs`: Source expression.

Return value:

New expression object.

GRBLinExpr::operator+

Add one expression into another, producing a result expression.

```
GRBLinExpr  operator+ (  const GRBLinExpr&  rhs )
```

Arguments:

rhs: Expression to add.

Return value:

Expression object which is equal the sum of the invoking expression and the argument expression.

GRBLinExpr::operator-

Subtract one expression from another, producing a result expression.

```
GRBLinExpr  operator- (  const GRBLinExpr&  rhs )
```

Arguments:

rhs: Expression to subtract.

Return value:

Expression object which is equal the invoking expression minus the argument expression.

GRBLinExpr::operator+=

Add an expression into the invoking expression.

```
void  operator+= (  const GRBLinExpr&  expr )
```

Arguments:

expr: Expression to add.

GRBLinExpr::operator-=

Subtract an expression from the invoking expression.

```
void  operator-= (  const GRBLinExpr&  expr )
```

Arguments:

expr: Expression to subtract.

GRBLinExpr::operator*=

Multiply the invoking expression by a constant.

```
void  operator*= (  double  multiplier )
```

Arguments:

`multiplier`: Constant multiplier.

GRBLinExpr::remove()

Remove a term from a linear expression.

```
void  remove (  int  i )
```

Remove the term stored at index `i` of the expression.

Arguments:

`i`: The index of the term to be removed.

```
boolean  remove (  GRBVar  var )
```

Remove all terms associated with variable `var` from the expression.

Arguments:

`var`: The variable whose term should be removed.

Return value:

Returns `true` if the variable appeared in the linear expression (and was removed).

GRBLinExpr::size()

Retrieve the number of terms in the linear expression (not including the constant).

```
unsigned int  size ( )
```

Return value:

Number of terms in the expression.

4.10 GRBQuadExpr

Gurobi quadratic expression object. A quadratic expression consists of a linear expression, plus a list of coefficient-variable-variable triples that capture the quadratic terms. Quadratic expressions are used to build quadratic objective functions and quadratic constraints. They are temporary objects that typically have short lifespans.

The `GRBQuadExpr` class is a sub-class of the abstract base class GRBExpr.

You generally build quadratic expressions using overloaded operators. For example, if `x` is a GRBVar object, then `x * x` is a GRBQuadExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x *x + 2 * x * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x * x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x * x`, or `expr2 -= expr1`).

The other option for building expressions is to start with an empty expression (using the GRBQuadExpr constructor), and then add terms. Terms can be added individually (using addTerm) or in groups (using addTerms). Terms can also be removed from an expression (using remove).

Note that the cost of building expressions depends heavily on the approach you use. While you can generally ignore this issue when building small expressions, you should be aware of a few efficiency issues when building large expressions:

- You should avoid using `expr = expr + x*x` in a loop. It will lead to runtimes that are quadratic in the number of terms in the expression.
- Using `expr += x*x` (or `expr -= x*x`) is much more efficient than `expr = expr + x*x`. Building a large expression by looping over `+=` statements is reasonably efficient, but it isn't the most efficient approach.
- The most efficient way to build a large expression is to make a single call addTerms.

To add a quadratic constraint to your model, you generally build one or two quadratic expression objects (`qexpr1` and `qexpr2`) and then use an overloaded comparison operator to build an argument for GRBModel::addQConstr. To give a few examples:

$$\text{model.addQConstr}(qexpr1 <= qexpr2)$$
$$\text{model.addQConstr}(qexpr1 == 1)$$
$$\text{model.addQConstr}(2 * x * x + 3 * y * y <= 4)$$

Once you add a constraint to your model, subsequent changes to the expression object you used to build the constraint will have no effect on that constraint.

Individual terms in a quadratic expression can be queried using the getVar1, getVar2, and getCoeff methods. You can query the number of quadratic terms in the expression using the size method. To query the constant and linear terms associated with a quadratic expression, first obtain the linear portion of the quadratic expression using getLinExpr, and then use the getConstant, getCoeff, or getVar on the resulting GRBLinExpr object.

Note that a quadratic expression may contain multiple terms that involve the same variable pair. These duplicate terms are merged when creating the model objective from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar1 and getVar2).

GRBQuadExpr()

Quadratic expression constructor. Create a constant expression or an expression with one term.

```
GRBQuadExpr  GRBQuadExpr (  double  constant=0.0 )
```

Create a constant quadratic expression.

Arguments:

`constant (optional)`: Constant value for expression.

Return value:

A constant expression object.

```
GRBQuadExpr  GRBQuadExpr (  GRBVar  var,
                            double  coeff=1.0 )
```

Create an expression with one term.

Arguments:

`var`: Variable for expression term.

`coeff (optional)`: Coefficient for expression term.

Return value:

An expression object containing one quadratic term.

```
GRBQuadExpr  GRBQuadExpr (  GRBLinExpr  linexpr )
```

Initialize a quadratic expression from an existing linear expression.

Arguments:

`orig`: Existing linear expression to copy.

Return value:

Quadratic expression object whose initial value is taken from the input linear expression.

GRBQuadExpr::addTerm()

Add a single new term into a quadratic expression.

```
void  addTerm (  double  coeff,
                 GRBVar  var )
```

Add a new linear term into a quadratic expression.

Arguments:

`coeff`: Coefficient for new linear term.

`var`: Variable for new linear term.

```
void  addTerm (  double  coeff,
                 GRBVar  var1,
                 GRBVar  var2 )
```

Add a new quadratic term into a quadratic expression.

Arguments:

`coeff`: Coefficient for new quadratic term.

`var1`: Variable for new quadratic term.

`var2`: Variable for new quadratic term.

GRBQuadExpr::addTerms()

Add new terms into a quadratic expression.

```
void  addTerms (  const double*  coeffs,
                  const GRBVar*  vars,
                  int            count )
```

Add new linear terms into a quadratic expression.

Arguments:

`coeffs`: Coefficients for new linear terms.

`vars`: Variables for new linear terms.

`count`: Number of linear terms to add to the quadratic expression.

```
void  addTerms (  const double*  coeffs,
                  const GRBVar*  vars1,
                  const GRBVar*  vars2,
                  int            count )
```

Add new quadratic terms into a quadratic expression.

Arguments:

`coeffs`: Coefficients for new quadratic terms.

`vars1`: First variables for new quadratic terms.

`vars2`: Second variables for new quadratic terms.

`count`: Number of quadratic terms to add to the quadratic expression.

GRBQuadExpr::clear()

Set a quadratic expression to 0.

You should use the overloaded `expr = 0` instead. The `clear` method is mainly included for consistency with our interfaces to non-overloaded languages.

```
void  clear ( )
```

GRBQuadExpr::getCoeff()

Retrieve the coefficient from a single quadratic term of the quadratic expression.

```
double  getCoeff (  int  i )
```

Arguments:

`i`: Index for coefficient of interest.

Return value:

Coefficient for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr::getLinExpr()

A quadratic expression is represented as a linear expression, plus a list of quadratic terms. This method retrieves the linear expression associated with the quadratic expression.

```
GRBLinExpr  getLinExpr ( )
```

Return value:

Linear expression associated with the quadratic expression.

GRBQuadExpr::getValue()

Compute the value of a quadratic expression for the current solution.

```
double  getValue ( )
```

Return value:

Value of the expression for the current solution.

GRBQuadExpr::getVar1()

Retrieve the first variable object associated with a single quadratic term from the expression.

```
GRBVar  getVar1 (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

First variable for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr::getVar2()

Retrieve the second variable object associated with a single quadratic term from the expression.

```
GRBVar  getVar2 (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

Second variable for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr::operator=

Set a quadratic expression equal to another quadratic expression.

```
GRBQuadExpr  operator= (  const GRBQuadExpr&  rhs )
```

Arguments:

rhs: Source quadratic expression.

Return value:

New quadratic expression object.

GRBQuadExpr::operator+

Add one expression into another, producing a result expression.

```
GRBQuadExpr  operator+ (  const GRBQuadExpr&  rhs )
```

Arguments:

rhs: Expression to add.

Return value:

Expression object which is equal the sum of the invoking expression and the argument expression.

GRBQuadExpr::operator-

Subtract one expression from another, producing a result expression.

```
GRBQuadExpr  operator- (  const GRBQuadExpr&  rhs )
```

Arguments:

rhs: Expression to subtract.

Return value:

Expression object which is equal the invoking expression minus the argument expression.

GRBQuadExpr::operator+=

Add an expression into the invoking expression.

```
void  operator+= (  const GRBQuadExpr&  expr )
```

Arguments:

expr: Expression to add.

GRBQuadExpr::operator-=

Subtract an expression from the invoking expression.

```
void  operator-= (  const GRBQuadExpr&  expr )
```

Arguments:

`expr`: Expression to subtract.

GRBQuadExpr::operator*=

Multiply the invoking expression by a constant.

```
void  operator*= (  double  multiplier )
```

Arguments:

`multiplier`: Constant multiplier.

GRBQuadExpr::remove()

Remove a quadratic term from a quadratic expression.

```
void  remove (  int  i )
```

Remove the quadratic term stored at index `i` of the expression.

Arguments:

`i`: The index of the term to be removed.

```
boolean  remove (  GRBVar  var )
```

Remove all quadratic terms associated with variable `var` from the quadratic expression.

Arguments:

`var`: The variable whose term should be removed.

Return value:

Returns `true` if the variable appeared in the quadratic expression (and was removed).

GRBQuadExpr::size()

Retrieve the number of quadratic terms in the quadratic expression.

```
unsigned int  size ( )
```

Return value:

Number of quadratic terms in the expression.

4.11 GRBTempConstr

Gurobi temporary constraint object. Objects of this class are created as intermediate results when building constraints using overloaded operators. There are no member functions on this class. Instead, `GRBTempConstr` objects are created by a set of non-member functions: ==, <=, and >=. You will generally never store objects of this class in your own variables.

Consider the following examples:

```
model.addConstr(x + y <= 1);
model.addQConstr(x*x + y*y <= 1);
```

The overloaded `<=` operator creates an object of type `GRBTempContr`, which is then immediately passed to method GRBModel::addConstr or GRBModel::addQConstr.

4.12 GRBColumn

Gurobi column object. A column consists of a list of coefficient, constraint pairs. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients. They are temporary objects that typically have short lifespans.

You generally build columns by starting with an empty column (using the GRBColumn constructor), and then adding terms. Terms can be added individually, using addTerm, or in groups, using addTerms. Terms can also be removed from a column, using remove.

Individual terms in a column can be queried using the getConstr, and getCoeff methods. You can query the number of terms in the column using the size method.

GRBColumn()

Column constructor. Create an empty column.

```
GRBColumn  GRBColumn ( )
```

Return value:

An empty column object.

GRBColumn::addTerm()

Add a single term into a column.

```
void  addTerm (  double     coeff,
                 GRBConstr  constr )
```

Arguments:

`coeff`: Coefficient for new term.

`constr`: Constraint for new term.

GRBColumn::addTerms()

Add new terms into a column.

```
void  addTerms (  const double*     coeffs,
                  const GRBConstr*  constrs,
                  int               count )
```

Add a list of terms into a column.

Arguments:

`coeffs`: Coefficients for new terms.

`constrs`: Constraints for new terms.

`count`: Number of terms to add to the column.

GRBColumn::clear()

Remove all terms from a column.

```
void  clear ( )
```

GRBColumn::getCoeff()

Retrieve the coefficient from a single term in the column.

```
double  getCoeff (  int  i )
```

Return value:
Coefficient for the term at index `i` in the column.

GRBColumn::getConstr()

Retrieve the constraint object from a single term in the column.

```
GRBConstr  getConstr (  int  i )
```

Return value:
Constraint for the term at index `i` in the column.

GRBColumn::remove()

Remove a single term from a column.

```
void  remove (  int  i )
```

Remove the term stored at index `i` of the column.
Arguments:
`i`: The index of the term to be removed.

```
boolean  remove (  GRBConstr  constr )
```

Remove the term associated with constraint `constr` from the column.
Arguments:
`constr`: The constraint whose term should be removed.
Return value:
Returns `true` if the constraint appeared in the column (and was removed).

GRBColumn::size()

Retrieve the number of terms in the column.

```
unsigned int  size ( )
```

Return value:
Number of terms in the column.

4.13 GRBCallback

Gurobi callback class. This is an abstract class. To implement a callback, you should create a subclass of this class and implement a `callback()` method. If you pass an object of this subclass to method GRBModel::setCallback before calling GRBModel::optimize or GRBModel::computeIIS, the `callback()` method of the class will be called periodically. Depending on where the callback is called from, you can obtain various information about the progress of the optimization.

Note that this class contains one protected *int* member variable: `where`. You can query this variable from your `callback()` method to determine where the callback was called from.

Gurobi callbacks can be used both to monitor the progress of the optimization and to modify the behavior of the Gurobi Optimizer. A simple user callback function might call the GRBCallback::getIntInfo or GRBCallback::getDoubleInfo methods to produce a custom display, or perhaps to terminate optimization early (using GRBCallback::abort) or to proceed to the next phase of the computation (using GRBCallback::proceed). More sophisticated MIP callbacks might use GRBCallback::getNodeRel or GRBCallback::getSolution to retrieve values from the solution to the current node, and then use GRBCallback::addCut or GRBCallback::addLazy to add a constraint to cut off that solution, or GRBCallback::setSolution to import a heuristic solution built from that solution. For multi-objective problems, you might use GRBCallback::stopOneMultiObj to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

When solving a model using multiple threads, the user callback is only ever called from a single thread, so you don't need to worry about the thread-safety of your callback.

Note that changing parameters from within a callback is not supported, doing so may lead to undefined behavior.

You can look at the `callback_c++.cpp` example for details of how to use Gurobi callbacks.

GRBCallback()

Callback constructor.

```
GRBCallback  GRBCallback ( )
```

Return value:

A callback object.

GRBCallback::abort()

Abort optimization. When the optimization stops, the Status attribute will be equal to `GRB_INTERRUPTED`.

```
void  abort ( )
```

GRBCallback::addCut()

Add a cutting plane to the MIP model from within a callback function. Note that this method can only be invoked when the `where` member variable is equal to `GRB_CB_MIPNODE` (see the Callback Codes section for more information).

Cutting planes can be added at any node of the branch-and-cut tree. However, they should be added sparingly, since they increase the size of the relaxation model that is solved at each node and can significantly degrade node processing speed.

Cutting planes are typically used to cut off the current relaxation solution. To retrieve the relaxation solution at the current node, you should first call getNodeRel.

You should consider setting parameter PreCrush to value 1 when adding your own cuts. This setting shuts off a few presolve reductions that can sometimes prevent your cut from being applied to the presolved model (which would result in your cut being silently ignored).

Note that cutting planes added through this method must truly be cutting planes -- they can cut off continuous solutions, but they may not cut off integer solutions that respect the original constraints of the model. Ignoring this restriction will lead to incorrect solutions.

```
void  addCut ( const GRBLinExpr&  lhsExpr,
               char               sense,
               double             rhsVal )
```

Arguments:

lhsExpr: Left-hand side expression for new cutting plane.

sense: Sense for new cutting plane (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVal: Right-hand side value for new cutting plane.

```
void  addCut ( GRBTempConstr&  tc )
```

Arguments:

tc: Temporary constraint object, created using an overloaded comparison operator. See GRBTempConstr for more information.

GRBCallback::addLazy()

Add a lazy constraint to the MIP model from within a callback function. Note that this method can only be invoked when the **where** member variable is equal to GRB_CB_MIPNODE or GRB_CB_MIPSOL (see the Callback Codes section for more information).

Lazy constraints are typically used when the full set of constraints for a MIP model is too large to represent explicitly. By only including the constraints that are actually violated by solutions found during the branch-and-cut search, it is sometimes possible to find a proven optimal solution while only adding a fraction of the full set of constraints.

You would typically add a lazy constraint by first querying the current node solution (by calling getSolution from a GRB_CB_MIPSOL callback, or getNodeRel from a GRB_CB_MIPNODE callback), and then calling addLazy() to add a constraint that cuts off the solution. Gurobi guarantees that you will have the opportunity to cut off any solutions that would otherwise be considered feasible.

MIP solutions may be generated outside of a MIP node. Thus, generating lazy constraints is optional when the **where** value in the callback function equals GRB_CB_MIPNODE. To avoid this, we recommend to always check when the **where** value equals GRB_CB_MIPSOL.

Your callback should be prepared to cut off solutions that violate any of your lazy constraints, including those that have already been added. Node solutions will usually respect previously added lazy constraints, but not always.

Note that you must set the LazyConstraints parameter if you want to use lazy constraints.

```
void  addLazy ( const GRBLinExpr&  lhsExpr,
                char               sense,
                double             rhsVal )
```

Arguments:

lhsExpr: Left-hand side expression for new lazy constraint.

sense: Sense for new lazy constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVal: Right-hand side value for new lazy constraint.

```
void  addLazy ( GRBTempConstr&  tc )
```

Arguments:

tc: Temporary constraint object, created using an overloaded comparison operator. See GRBTempConstr for more information.

GRBCallback::getDoubleInfo()

Request double-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the double-valued information that can be queried for different values of **where**, please refer to the Callback section.

```
double  getDoubleInfo ( int  what )
```

Arguments:

what: Information requested (refer the list of Gurobi Callback Codes for possible values).

Return value:

Value of requested callback information.

GRBCallback::getIntInfo()

Request int-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the int-valued information that can be queried for different values of **where**, please refer to the Callback section.

```
int  getIntInfo ( int  what )
```

Arguments:

what: Information requested (refer to the list of Gurobi Callback Codes for possible values).

Return value:

Value of requested callback information.

GRBCallback::getNodeRel()

Retrieve values from the node relaxation solution at the current node. Only available when the `where` member variable is equal to `GRB_CB_MIPNODE`, and `GRB_CB_MIPNODE_STATUS` is equal to `GRB_OPTIMAL`.

```
double  getNodeRel (  GRBVar  v )
```

Arguments:

`v`: The variable whose value is desired.

Return value:

The value of the specified variable in the node relaxation for the current node.

```
double*  getNodeRel (  const GRBVar*  xvars,
                       int            len )
```

Arguments:

`xvars`: The list of variables whose values are desired.

`len`: The number of variables in the list.

Return value:

The values of the specified variables in the node relaxation for the current node. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBCallback::getSolution()

Retrieve values from the current solution vector. Only available when the `where` member variable is equal to `GRB_CB_MIPSOL` or `GRB_CB_MULTIOBJ`.

```
double  getSolution (  GRBVar  v )
```

Arguments:

`v`: The variable whose value is desired.

Return value:

The value of the specified variable in the current solution vector.

```
double*  getSolution (  const GRBVar*  xvars,
                        int            len )
```

Arguments:

`xvars`: The list of variables whose values are desired.

`len`: The number of variables in the list.

Return value:

The values of the specified variables in the current solution. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBCallback::getStringInfo()

Request string-valued callback information. The available information depends on the value of the `where` member. For information on possible values of `where`, and the string-valued information that can be queried for different values of `where`, please refer to the Callback section.

```
string  getStringInfo (  int  what )
```

Arguments:

`what`: Information requested (refer to the list of Gurobi Callback Codes for possible values).

Return value:

Value of requested callback information.

GRBCallback::proceed()

Generate a request to proceed to the next phase of the computation. Note that the request is only accepted in a few phases of the algorithm, and it won't be acted upon immediately.

In the current Gurobi version, this callback allows you to proceed from the NoRel heuristic to the standard MIP search. You can determine the current algorithm phase using `MIP_PHASE`, `MIPNODE_PHASE`, or `MIPSOL_PHASE` queries from a callback.

```
void  proceed ( )
```

GRBCallback::setSolution()

Import solution values for a heuristic solution. Only available when the `where` member variable is equal to `GRB_CB_MIP`, `GRB_CB_MIPNODE`, or `GRB_CB_MIPSOL` (see the Callback Codes section for more information).

When you specify a heuristic solution from a callback, variables initially take undefined values. You should use this method to specify variable values. You can make multiple calls to `setSolution` from one callback invocation to specify values for multiple sets of variables. After the callback, if values have been specified for any variables, the Gurobi Optimizer will try to compute a feasible solution from the specified values, possibly filling in values for variables whose values were left undefined. You can also optionally call useSolution within your callback function to try to immediately compute a feasible solution from the specified values.

Note that this method is not supported in a Compute Server environment.

```
void  setSolution (  GRBVar  v,
                     double  val )
```

Arguments:

`v`: The variable whose values is being set.

`val`: The value of the variable in the new solution.

```
void  setSolution (  const GRBVar*  xvars,
                     const double*  sol,
                     int            len )
```

Arguments:

`xvars`: The variables whose values are being set.
`sol`: The values of the variables in the new solution.
`len`: The number of variables.

GRBCallback::stopOneMultiObj()

Interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process. Only available for multi-objective MIP models and when the `where` member variable is not equal to `GRB_CB_MULTIOBJ` (see the Callback Codes section for more information).

You would typically stop a multi-objective optimization step by querying the last finished number of multi-objectives steps, and using that number to stop the current step and move on to the next hierarchical objective (if any) as shown in the following example:

Example usage:

```
#include <ctime>

class mycallback: public GDBCallback
{
  public:
    int    objcnt    = 0;
    time_t starttime = time();

  protected:
    void callback () {
      if (where == GRB_CB_MULTIOBJ) {
        /* get current objective number */
        objcnt = getIntInfo(GRB_CB_MULTIOBJ_OBJCNT);

        /* reset start time to current time */
        starttime = time();
      } else if (time() - startime > BIG ||
          /* takes too long or good enough  */) {
        /* stop only this optimization step */
        stopOneMultiObj(objcnt);
      }
    }
}
```

You should refer to the section on Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

```
void  stopOneMultiObj ( int  objcnt )
```

Arguments:

objnum: The number of the multi-objective optimization step to interrupt. For processes running locally, this argument can have the special value -1, meaning to stop the current step.

GRBCallback::useSolution()

Once you have imported solution values using setSolution, you can optionally call `useSolution` in a `GRB_CB_MIPNODE` callback to immediately use these values to try to compute a heuristic solution. Alternatively, you can call `useSolution` in a `GRB_CB_MIP` or `GRB_CB_MIPSOL` callback, which will store the solution until it can be processed internally.

```
double  useSolution ( )
```

Return value:

The objective value for the solution obtained from your solution values. It equals `GRB_INFINITY` if no improved solution is found or the method has been called from a callback other than `GRB_CB_MIPNODE` as, in these contexts, the solution is stored instead of being processed immediately.

4.14 GRBException

Gurobi exception object. Exceptions can be thrown by nearly every method in the Gurobi C++ API.

GRBException()

Exception constructor.

```
GRBException  GRBException (  int  errcode=0 )
```

Create a Gurobi exception.

Arguments:

errcode (optional): Error code for exception.

Return value:

An exception object.

```
GRBException  GRBException (  string  errmsg,
                              int     errcode=0 )
```

Create a Gurobi exception.

Arguments:

errmsg: Error message for exception.

errcode (optional): Error code for exception.

Return value:

An exception object.

GRBException::getErrorCode()

Retrieve the error code associated with a Gurobi exception.

```
int  getErrorCode ( )
```

Return value:

The error code associated with the exception.

GRBException::getMessage()

Retrieve the error message associated with a Gurobi exception.

```
const string  getMessage ( )
```

Return value:

The error message associated with the exception.

4.15 GRBBatch

Gurobi batch object. Batch optimization is a feature available with the Gurobi Cluster Manager. It allows a client program to build an optimization model, submit it to a Compute Server cluster (through a Cluster Manager), and later check on the status of the model and retrieve its solution. For more information, please refer to the Batch Optimization section.

Commonly used methods on batch objects include update (refresh attributes from the Cluster Manager), abort (abort execution of a batch request), retry (retry optimization for an interrupted or failed batch), discard (remove the batch request and all related information from the Cluster Manager), and getJSONSolution (query solution information for the batch request).

These methods are built on top of calls to the Cluster Manager REST API. They are meant to simplify such calls, but note that you always have the option of calling the REST API directly.

Batch objects have four attributes:

- BatchID: Unique ID for the batch request.
- BatchStatus: Last batch status.
- BatchErrorCode: Last error code.
- BatchErrorMessage: Last error message.

You can access their values by using get. Note that all Batch attributes are locally cached, and are only updated when you create a client-side batch object or when you explicitly update this cache, which can done by calling update.

GRBBatch()

Constructor for `GRBBatch`.

Given a `BatchID`, as returned by optimizeBatch, and a Gurobi environment that can connect to the appropriate Cluster Manager (i.e., one where parameters CSManager, UserName, and ServerPassword have been set appropriately), this function returns a GRBBatch object. With it, you can query the current status of the associated batch request and, once the batch request has been processed, you can query its solution. Please refer to the Batch Optimization section for details and examples.

```
GRBBatch  GRBBatch (  GRBEnv&  env,
                      string&  batchID )
```

Arguments:

`env`: The environment in which the new batch object should be created.

`batchID`: ID of the batch request for which you want to access status and other information.

Return value:

New batch object.

Example usage:

```
GRBBatch batch = GRBBatch(env, batchID);
```

GRBBatch::abort()

This method instructs the Cluster Manager to abort the processing of this batch request, changing its status to ABORTED. Please refer to the Batch Status Codes section for further details.
`void abort ( )`

Example usage:

```
// Abort this batch if it is taking too long
time_t curtime = time(NULL);
if (curtime - starttime > maxwaittime) {
  batch->abort();
  break;
}
```

GRBBatch::discard()

This method instructs the Cluster Manager to remove all information related to the batch request in question, including the stored solution if available. Further queries for the associated batch request will fail with error code GRB_ERROR_DATA_NOT_AVAILABLE. Use this function with care, as the removed information can not be recovered later on.
`void discard ( )`

Example usage:

```
void
batchdiscard(string batchID)
```

GRBBatch::getJSONSolution()

This method retrieves the solution of a completed batch request from a Cluster Manager. The solution is returned as a JSON solution string. For this call to succeed, the status of the batch request must be COMPLETED. Please refer to the Batch Status Codes section for further details.
`string getJSONSolution ( )` **Return value:**

The requested solution in JSON format.

Example usage:

```
// Pretty printing the general solution information
cout << "JSON solution:" << batch->getJSONSolution() << endl;
```

GRBBatch::get()

Query the value of an attribute.

`int get ( GRB_IntAttr attr )`

Query the value of an int-valued batch attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
string  get ( GRB_StringAttr  attr )
```

Query the value of a string-valued batch attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBBatch::retry()

This method instructs the Cluster Manager to retry optimization of a failed or aborted batch request, changing its status to SUBMITTED. Please refer to the Batch Status Codes section for further details. `void  retry ( )`

Example usage:

```
// If the batch failed, we try again
if (BatchStatus == GRB_BATCH_FAILED)
  batch->retry();
```

GRBBatch::update()

All Batch attribute values are cached locally, so queries return the value received during the last communication with the Cluster Manager. This method refreshes the values of all attributes with the values currently available in the Cluster Manager (which involves network communication). `void  update ( )`

Example usage:

```
// Update the resident attribute cache of the Batch object with the
// latest values from the cluster manager.
batch->update();
BatchStatus = batch->get(GRB_IntAttr_BatchStatus);
```

GRBBatch::writeJSONSolution()

This method returns the stored solution of a completed batch request from a Cluster Manager. The solution is returned in a gzip-compressed JSON file. The file name you provide must end with a `.json.gz` extension. The JSON format is described in the JSON solution format section. Note that for this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details. `void writeJSONSolution ( string& filename )`

Arguments:

`filename`: Name of file where the solution should be stored (in JSON format).

Example usage:

```
// Write the full JSON solution string to a file
batch->writeJSONSolution("batch-sol.json.gz");
```

4.16 Non-Member Functions

Several Gurobi C++ interface functions aren't member functions on a particular object.

operator==

Create an equality constraint

```
GRBTempConstr  operator== (  GRBQuadExpr  lhsExpr,
                             GRBQuadExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of equality constraint.

rhsExpr: Right-hand side of equality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to GRBModel::addConstr.

operator<=

Create an inequality constraint

```
GRBTempConstr  operator<= (  GRBQuadExpr  lhsExpr,
                             GRBQuadExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of inequality constraint.

rhsExpr: Right-hand side of inequality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to GRBModel::addConstr or GRBModel::addQConstr.

operator>=

Create an inequality constraint

```
GRBTempConstr  operator>= (  GRBQuadExpr  lhsExpr,
                             GRBQuadExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of inequality constraint.

rhsExpr: Right-hand side of inequality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to GRBModel::addConstr or GRBModel::addQConstr.

operator+

Overloaded operator on expression objects.

```
GRBLinExpr  operator+ (  const GRBLinExpr&  expr1,
                         const GRBLinExpr&  expr2 )
```

Add a pair of expressions.

Arguments:

expr1: First expression to be added.

expr2: Second expression to be added.

Return value:

Sum expression.

```
GRBLinExpr  operator+ (  const GRBLinExpr&  expr )
```

Allow plus sign to be used before an expression.

Arguments:

expr: Expression.

Return value:

Result expression.

```
GRBLinExpr  operator+ (  GRBVar  x,
                         GRBVar  y )
```

Add a pair of variables.

Arguments:

x: First variable to be added.

y: Second variable to be added.

Return value:

Sum expression.

```
GRBQuadExpr  operator+ (  const GRBQuadExpr&  expr1,
                          const GRBQuadExpr&  expr2 )
```

Add a pair of expressions.

Arguments:

expr1: First expression to be added.

expr2: Second expression to be added.

Return value:

Sum expression.

```
GRBQuadExpr  operator+ (  const GRBQuadExpr&  expr )
```

Allow plus sign to be used before an expression.

Arguments:

expr: Expression.

Return value:

Result expression.

operator-

Overloaded operator on expression objects.

```
GRBLinExpr  operator- (  const GRBLinExpr&  expr1,
                         const GRBLinExpr&  expr2 )
```

Subtract one expression from another.

Arguments:

expr1: Start expression.

expr2: Expression to be subtracted.

Return value:

Difference expression.

```
GRBLinExpr  operator- (  const GRBLinExpr&  expr )
```

Negate an expression.

Arguments:

expr: Expression.

Return value:

Negation of expression.

```
GRBQuadExpr  operator- (  const GRBQuadExpr&  expr1,
                          const GRBQuadExpr&  expr2 )
```

Subtract one expression from another.

Arguments:

expr1: Start expression.

expr2: Expression to be subtracted.

Return value:

Difference expression.

```
GRBQuadExpr  operator- (  const GRBQuadExpr&  expr )
```

Negate an expression.

Arguments:

expr: Expression.

Return value:

Negation of expression.

operator*

Overloaded operator on expression objects.

```
GRBLinExpr  operator* (  GRBVar  x,
                         double  a )
```

Multiply a variable and a constant.

Arguments:

`x`: Variable.

`a`: Constant multiplier.

Return value:

Expression that represents the result of multiplying the variable by a constant.

```
GRBLinExpr  operator* (  double  a,
                         GRBVar  x )
```

Multiply a variable and a constant.

Arguments:

`a`: Constant multiplier.

`x`: Variable.

Return value:

Expression that represents the result of multiplying the variable by a constant.

```
GRBLinExpr  operator* (  const GRBLinExpr&  expr,
                         double             a )
```

Multiply an expression and a constant.

Arguments:

`expr`: Expression.

`a`: Constant multiplier.

Return value:

Expression that represents the result of multiplying the expression by a constant.

```
GRBLinExpr  operator* (  double             a,
                         const GRBLinExpr&  expr )
```

Multiply an expression and a constant.

Arguments:

`a`: Constant multiplier.

`expr`: Expression.

Return value:

Expression that represents the result of multiplying the expression by a constant.

```
GRBQuadExpr  operator* (  const GRBQuadExpr&  expr,
                          double              a )
```

Multiply an expression and a constant.

Arguments:

`expr`: Expression.

`a`: Constant multiplier.

Return value:

Expression that represents the result of multiplying the expression by a constant.

```
GRBQuadExpr  operator* (  double              a,
                          const GRBQuadExpr&  expr )
```

Multiply an expression and a constant.

Arguments:

`a`: Constant multiplier.

`expr`: Expression.

Return value:

Expression that represents the result of multiplying the expression by a constant.

```
GRBQuadExpr  operator* (  GRBVar  x,
                          GRBVar  y )
```

Multiply a pair of variables.

Arguments:

`x`: First variable.

`y`: Second variable.

Return value:

Expression that represents the result of multiplying the argument variables.

```
GRBQuadExpr  operator* (  GRBVar              var,
                          const GRBLinExpr&   expr )
```

Multiply an expression and a variable.

Arguments:

`var`: Variable.

`expr`: Expression.

Return value:

Expression that represents the result of multiplying the expression by a variable.

```
GRBQuadExpr  operator* (  const GRBLinExpr&  expr,
                          GRBVar             var )
```

Multiply an expression and a variable.

Arguments:

`var`: Variable.

`expr`: Expression.

Return value:

Expression that represents the result of multiplying the expression by a variable.

```
GRBQuadExpr  operator* (  const GRBLinExpr&  expr1,
                          const GRBLinExpr&  expr2 )
```

Multiply a pair of expressions.

Arguments:

`expr1`: First expression.

`expr2`: Second expression.

Return value:

Expression that represents the result of multiplying the argument expressions.

operator/

Overloaded operator to divide a variable or expression by a constant.

```
GRBLinExpr  operator/ (  GRBVar  x,
                         double  a )
```

Arguments:

`x`: Variable.

`a`: Constant divisor.

Return value:

Expression that represents the result of dividing the variable by a constant.

```
GRBLinExpr  operator/ (  const GRBLinExpr&  expr,
                         double             a )
```

Arguments:

`expr`: Expression.

`a`: Constant divisor.

Return value:

Expression that represents the result of dividing the expression by a constant.

```
GRBLinExpr  operator/ (  const GRBQuadExpr&  expr,
                         double              a )
```

Arguments:

`expr`: Expression.

`a`: Constant divisor.

Return value:

Expression that represents the result of dividing the expression by a constant.

4.17 Attribute Enums

These `enums` are used to get or set Gurobi attributes. The complete list of attributes can be found in the Attributes section.

GRB_CharAttr

This enum is used to get or set char-valued attributes (through GRBModel::get or GRBModel::set). Please refer to the Attributes section to see a list of all char attributes and their functions.

GRB_DoubleAttr

This enum is used to get or set double-valued attributes (through GRBModel::get or GRBModel::set). Please refer to the Attributes section to see a list of all double attributes and their functions.

GRB_IntAttr

This enum is used to get or set int-valued attributes (through GRBModel::get or GRBModel::set). Please refer to the Attributes section to see a list of all int attributes and their functions.

GRB_StringAttr

This enum is used to get or set string-valued attributes (through GRBModel::get or GRBModel::set). Please refer to the Attributes section to see a list of all string attributes and their functions.

4.18 Parameter Enums

These `enums` are used to get or set Gurobi parameters. The complete of parameters can be found in the Parameters section.

GRB_DoubleParam

This enum is used to get or set double-valued parameters (through GRBModel::get, GRBModel::set. GRBEnv::get, or GRBEnv::set). Please refer to the Parameters section to see a list of all double parameters and their functions.

GRB_IntParam

This enum is used to get or set int-valued parameters (through GRBModel::get, GRBModel::set. GRBEnv::get, or GRBEnv::set). Please refer to the Parameters section to see a list of all int parameters and their functions.

GRB_StringParam

This enum is used to get or set string-valued parameters (through GRBModel::get, GRBModel::set, GRBEnv::get, or GRBEnv::set). Please refer to the Parameters section to see a list of all int parameters and their functions.

Java API Overview

This section documents the Gurobi Java interface. This manual begins with a quick overview of the classes exposed in the interface and the most important methods on those classes. It then continues with a comprehensive presentation of all of the available classes and methods.

If you prefer Javadoc format, documentation for the Gurobi Java interface is also available in file `gurobi-javadoc.jar`. Javadoc format is particularly helpful when used from an integrated development environment like Eclipse®. Please consult the documentation for your IDE for information on how to import Javadoc files.

If you are new to the Gurobi Optimizer, we suggest that you start with the Getting Started Knowledge Base article or the Example Tour. These documents provide concrete examples of how to use the classes and methods described here.

Environments

The first step in using the Gurobi Java interface is to create an environment object. Environments are represented using the GRBEnv class. An environment acts as the container for all data associated with a set of optimization runs. You will generally only need one environment object in your program.

For more advanced usecases, you can use an empty environment to create an uninitialized environment and then, programmatically, set all required options for your specific requirements. For further details see the Environment section.

Models

You can create one or more optimization models within an environment. Each model is represented as an object of class GRBModel. A model consists of a set of decision variables (objects of class GRBVar), a linear or quadratic objective function on these variables (specified using GRBModel.setObjective), and a set of constraints on these variables (objects of class GRBConstr, GRBQConstr, GRBSOS, or GRBGenConstr). Each variable has an associated lower bound, upper bound, and type (continuous, binary, etc.). Each linear or quadratic constraint has an associated sense (less-than-or-equal, greater-than-or-equal, or equal), and right-hand side value. Refer to this section in the Reference Manual for more information on variables, constraints, and objectives.

Linear constraints are specified by building linear expressions (objects of class GRBLinExpr), and then specifying relationships between these expressions (for example, requiring that one expression be equal to another). Quadratic constraints are built in a similar fashion, but using quadratic expressions (objects of class GRBQuadExpr) instead.

An optimization model may be specified all at once, by loading the model from a file (using the appropriate GRBModel constructor), or built incrementally, by first constructing an empty object of class GRBModel and then subsequently calling GRBModel.addVar or GRBModel.addVars to add additional variables, and GRBModel.addConstr, GRBModel.addQConstr, GRBModel.addSOS, or any of the GRBModel.addGenConstrXxx methods to add additional constraints. Models are dynamic entities; you can always add or remove variables or constraints.

We often refer to the *class* of an optimization model. At the highest level, a model can be

continuous or discrete, depending on whether the modeling elements present in the model require discrete decisions to be made. Among continuous models...

- A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*.
- If the objective is quadratic, the model is a *Quadratic Program (QP)*.
- If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We sometimes refer to a few special cases of QCP: QCPs with convex constraints, QCPs with non-convex constraints, *bilinear programs*, and *Second-Order Cone Programs (SOCP)*.
- If any of the constraints are non-linear (chosen from among the available general constraints), the model is a *Non-Linear Program (NLP)*.

A model that contains any integer variables, semi-continuous variables, semi-integer variables, Special Ordered Set (SOS) constraints, or general constraints, is discrete, and is referred to as a *Mixed Integer Program (MIP)*. The special cases of MIP, which are the discrete versions of the continuous models types we've already described, are...

- *Mixed Integer Linear Programs (MILP)*
- *Mixed Integer Quadratic Programs (MIQP)*
- *Mixed Integer Quadratically-Constrained Programs (MIQCP)*
- *Mixed Integer Second-Order Cone Programs (MISOCP)*
- *Mixed Integer Non-Linear Programs (MINLP)*

The Gurobi Optimizer handles all of these model classes. Note that the boundaries between them aren't as clear as one might like, because we are often able to transform a model from one class to a simpler class.

Solving a Model

Once you have built a model, you can call GRBModel.optimize to compute a solution. By default, optimize will use the concurrent optimizer to solve LP models, the barrier algorithm to solve QP models with convex objectives and QCP models with convex constraints, and the branch-and-cut algorithm otherwise. The solution is stored in a set of *attributes* of the model. These attributes can be queried using a set of attribute query methods on the GRBModel, GRBVar, GRBConstr, GRBQConstr, GRBSOS, and GRBGenConstr, and classes.

The Gurobi algorithms keep careful track of the state of the model, so calls to GRBModel.optimize will only perform further optimization if relevant data has changed since the model was last optimized. If you would like to discard previously computed solution information and restart the optimization from scratch without changing the model, you can call GRBModel.reset.

After a MIP model has been solved, you can call GRBModel.fixedModel to compute the associated *fixed* model. This model is identical to the original, except that the integer variables are fixed to their values in the MIP solution. If your model contains SOS constraints, some continuous variables that appear in these constraints may be fixed as well. In some applications, it can be useful to compute information on this fixed model (e.g., dual variables, sensitivity information, etc.), although you should be careful in how you interpret this information.

Multiple Solutions, Objectives, and Scenarios

By default, the Gurobi Optimizer assumes that your goal is to find one proven optimal solution to a single model with a single objective function. Gurobi provides the following features that allow you to relax these assumptions:

- Solution Pool: Allows you to find more solutions.
- Multiple Scenarios: Allows you to find solutions to multiple, related models.
- Multiple Objectives: Allows you to specify multiple objective functions and control the trade-off between them.

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call GRBModel.computeIIS to compute an Irreducible Inconsistent Subsystem (IIS). This method can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. This method populates a set of IIS attributes.

To attempt to repair an infeasibility, call GRBModel.feasRelax to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Querying and Modifying Attributes

Most of the information associated with a Gurobi model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. To give a simple example, solving an optimization model causes the `X` variable attribute to be populated. Attributes such as `X` that are computed by the Gurobi Optimizer cannot be modified directly by the user, while others, such as the variable lower bound (the `LB` attribute) can.

Attributes are queried using GRBVar.get, GRBConstr.get, GRBQConstr.get, GRBSOS.get, GRBGenConstr.get, or GRBModel.get, and modified using GRBVar.set, GRBConstr.set, GRBQConstr.set, GRBGenConstr.set, or GRBModel.set. Attributes are grouped into a set of enums by type (GRB.CharAttr, GRB.DoubleAttr, GRB.IntAttr, GRB.StringAttr). The `get()` and `set()` methods are overloaded, so the type of the attribute determines the type of the returned value. Thus, `constr.get(GRB.DoubleAttr.RHS)` returns a double, while `constr.get(GRB.CharAttr.Sense)` returns a char.

If you wish to retrieve attribute values for a set of variables or constraints, it is usually more efficient to use the array methods on the associated GRBModel object. Method GRBModel.get includes signatures that allow you to query or modify attribute values for one-, two-, and three-dimensional arrays of variables or constraints.

The full list of attributes can be found in the Attributes section.

Additional Model Modification Information

Most modifications to an existing model are done through the attribute interface (e.g., changes to variable bounds, constraint right-hand sides, etc.). The main exceptions are modifications to the constraint matrix and to the objective function.

The constraint matrix can be modified in a few ways. The first is to call the chgCoeff method on a GRBModel object to change individual matrix coefficients. This method can be used to modify the value of an existing non-zero, to set an existing non-zero to zero, or to create a new non-zero. The constraint matrix is also modified when you remove a variable or constraint from the model (through the GRBModel.remove method). The non-zero values associated with the deleted constraint or variable are removed along with the constraint or variable itself.

The model objective function can also be modified in a few ways. The easiest is to build an expression that captures the objective function (a GRBLinExpr or GRBQuadExpr object), and then pass that expression to method GRBModel.setObjective. If you wish to modify the objective, you can simply call `setObjective` again with a new `GRBLinExpr` or `GRBQuadExpr` object.

For linear objective functions, an alternative to `setObjective` is to use the `Obj` variable attribute to modify individual linear objective coefficients.

If your variables have piecewise-linear objectives, you can specify them using the GRBModel.setPWLObj method. Call this method once for each relevant variable. The Gurobi simplex solver includes algorithmic support for convex piecewise-linear objective functions, so for continuous models you should see a substantial performance benefit from using this feature. To clear a previously specified piecewise-linear objective function, simply set the `Obj` attribute on the corresponding variable to 0.

Lazy Updates

One important item to note about model modification in the Gurobi optimizer is that it is performed in a *lazy* fashion, meaning that modifications don't affect the model immediately. Rather, they are queued and applied later. If your program simply creates a model and solves it, you will probably never notice this behavior. However, if you ask for information about the model before your modifications have been applied, the details of the lazy update approach may be relevant to you.

As we just noted, model modifications (bound changes, right-hand side changes, objective changes, etc.) are placed in a queue. These queued modifications can be applied to the model in three different ways. The first is by an explicit call to GRBModel.update. The second is by a call to GRBModel.optimize. The third is by a call to GRBModel.write to write out the model. The first case gives you fine-grained control over when modifications are applied. The second and third make the assumption that you want all pending modifications to be applied before you optimize your model or write it to disk.

Why does the Gurobi interface behave in this manner? There are a few reasons. The first is that this approach makes it much easier to perform multiple modifications to a model, since the model remains unchanged between modifications. The second is that processing model modifications can be expensive, particularly in a Compute Server environment, where modifications require communication between machines. Thus, it is useful to have visibility into exactly when these modifications are applied. In general, if your program needs to make multiple modifications to the model, you should aim to make them in phases, where you make a set of modifications, then update, then make more modifications, then update again, etc. Updating after each individual modification can be extremely expensive.

If you forget to call update, your program won't crash. Your query will simply return the value of the requested data from the point of the last update. If the object you tried to query didn't exist then, you'll get a `NOT_IN_MODEL` exception instead.

The semantics of lazy updates have changed since earlier Gurobi versions. While the vast

majority of programs are unaffected by this change, you can use the UpdateMode parameter to revert to the earlier behavior if you run into an issue.

Managing Parameters

The Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization. Parameters can be of type *int*, *double*, or *string*.

The simplest way to set parameters is through the GRBModel.set method on the model object. Similarly, parameter values can be queried with GRBModel.get.

Parameters can also be set on the Gurobi environment object, using GRBEnv.set. Note that each model gets its own copy of the environment when it is created, so parameter changes to the original environment have no effect on existing models.

You can read a set of parameter settings from a file using GRBEnv.readParams, or write the set of changed parameters using GRBEnv.writeParams.

We also include an automated parameter tuning tool that explores many different sets of parameter changes in order to find a set that improves performance. You can call GRBModel.tune to invoke the tuning tool on a model. Refer to the parameter tuning tool section for more information.

The full list of Gurobi parameters can be found in the Parameters section.

Memory Management

Users typically do not need to concern themselves with memory management in Java, since it is handled automatically by the garbage collector. The Gurobi Java interface utilizes the same garbage collection mechanism as other Java programs, but there are a few specifics of our memory management that users should be aware of.

In general, Gurobi objects live in the same Java heap as other Java objects. When they are no longer referenced, they become candidates for garbage collection, and are returned to the pool of free space at the next invocation of the garbage collector. Two important exceptions are the GRBEnv and GRBModel objects. A `GRBModel` object has a small amount of memory associated with it in the Java heap, but the majority of the space associated with a model lives in the heap of the Gurobi native code library (the Gurobi DLL in Windows, or the Gurobi shared library in Linux or Mac). The Java heap manager is unaware of the memory associated with the model in the native code library, so it does not consider this memory usage when deciding whether to invoke the garbage collector. When the garbage collector eventually collects the Java `GRBModel` object, the memory associated with the model in the Gurobi native code library will be freed, but this collection may come later than you might want. Similar considerations apply to the `GRBEnv` object.

If you are writing a Java program that makes use of multiple Gurobi models or environments, we recommend that you call GRBModel.dispose when you are done using the associated `GRBModel` object, and GRBEnv.dispose when you are done using the associated `GRBEnv` object and after you have called GRBModel.dispose on all of the models created using that `GRBEnv` object.

Native Code

As noted earlier, the Gurobi Java interface is a thin layer that sits on top of our native code library (the Gurobi DLL on Windows, and the Gurobi shared library on Linux or Mac). Thus, an application that uses the Gurobi Java library will load the Gurobi native code library at runtime. In order for this happen, you need to make sure that two things are true. First, you need to make

sure that the native code library is available in the search path of the target machine (`PATH` on Windows, `LD_LIBRARY_PATH` on Linux, or `DYLD_LIBRARY_PATH` on Mac). These paths are set up as part of the installation of the Gurobi Optimizer, but may not be configured appropriately on a machine where the full Gurobi Optimizer has not been installed. Second, you need to be sure that the Java JVM and the Gurobi native library use the same object format. In particular, you need to use a 64-bit Java JVM to use the 64-bit Gurobi native library.

Monitoring Progress - Logging and Callbacks

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging behavior. If you would like to direct output to a file as well as to the screen, specify the log file name in the GRBEnv constructor. You can modify the LogFile parameter if you wish to redirect the log to a different file after creating the environment object. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter. A detailed description of the Gurobi log file can be found in the Logging section.

More detailed progress monitoring can be done through the GRBCallback class. The GRBModel.setCallback method allows you to receive a periodic callback from the Gurobi Optimizer. You do this by sub-classing the GRBCallback abstract class, and writing your own `Callback()` method on this class. You can call GRBCallback.getDoubleInfo, GRBCallback.getIntInfo, GRBCallback.getStringInfo, or GRBCallback.getSolution from within the callback to obtain additional information about the state of the optimization.

In addition, you can add a logging callback function to an environment object (GRBEnv.setLogCallback) or a model object (GRBModelEnv.setLogCallback). With that you catch output posted by an environment object or a model object.

Modifying Solver Behavior - Callbacks

Callbacks can also be used to modify the behavior of the Gurobi optimizer. The simplest control callback is GRBCallback.abort, which asks the optimizer to terminate at the earliest convenient point. Method GRBCallback.setSolution allows you to inject a feasible solution (or partial solution) during the solution of a MIP model. Methods GRBCallback.addCut and GRBCallback.addLazy allow you to add *cutting planes* and *lazy constraints* during a MIP optimization, respectively. Method GRBCallback.stopOneMultiObj allows you to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

Batch Optimization

Gurobi Compute Server enables programs to offload optimization computations onto dedicated servers. The Gurobi Cluster Manager adds a number of additional capabilities on top of this. One important one, *batch optimization*, allows you to build an optimization model with your client program, submit it to a Compute Server cluster (through the Cluster Manager), and later check on the status of the model and retrieve its solution. You can use a Batch object to make it easier to work with batches. For details on batches, please refer to the Batch Optimization section.

Error Handling

All of the methods in the Gurobi Java library can throw an exception of type GRBException. When an exception occurs, additional information on the error can be obtained by retrieving the

error code (using method GRBException.getErrorCode), or by retrieving the exception message (using method `GRBException.getMessage` from the parent class). The list of possible error return codes can be found in the Error Codes section.

5.1 GRBEnv

Gurobi environment object. Gurobi models are always associated with an environment. You must create an environment before can you create and populate a model. You will generally only need a single environment object in your program.

The methods on environment objects are mainly used to manage Gurobi parameters (e.g., get, getParamInfo, set).

While the Java garbage collector will eventually collect an unused `GRBEnv` object, an environment will hold onto resources (Gurobi licenses, file descriptors, etc.) until that collection occurs. If your program creates multiple `GRBEnv` objects, we recommend that you call GRBEnv.dispose when you are done using one.

GRBEnv()

Constructor for `GRBEnv` object. You have the option of constructing either a local environment, which solves Gurobi models on the local machine, or a client environment for a Gurobi Compute Server, which will solve Gurobi models on a server machine. For the latter, choose the signature that allows you to specify the names of the Gurobi Compute Servers and the priority of the associated job.

```
GRBEnv  GRBEnv ( )
```

Create a Gurobi environment (with logging disabled). This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Return value:

An environment object (with no associated log file).

```
GRBEnv  GRBEnv (  boolean  empty )
```

Create an empty Gurobi environment. Use start to start the environment.

If the environment is not empty, This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating

and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`empty`: Indicates whether the environment should be empty. You should use `empty=true` if you want to set parameters before actually starting the environment. This can be useful if you want to connect to a Compute Server, a Token Server, the Gurobi Instant Cloud, a Cluster Manager or use a WLS license. See the Environment Section for more details.

Return value:

An environment object.

```
GRBEnv  GRBEnv ( String logFileName )
```

Create a Gurobi environment (with logging enabled). This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`logFileName`: The desired log file name.

Return value:

An environment object.

GRBEnv.dispose()

Release the resources associated with a `GRBEnv` object. While the Java garbage collector will eventually reclaim these resources, we recommend that you call the `dispose` method when you are done using an environment if your program creates more than one.

The `dispose` method on a `GRBEnv` should be called only after you have called `dispose` on all of the models that were created within that environment. You should not attempt to use a `GRBEnv` object after calling `dispose`.

```
void  dispose ( )
```

GRBEnv.get()

Query the value of a parameter.

```
double  get ( GRB.DoubleParam  param )
```

Query the value of a double-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

```
int  get ( GRB.IntParam  param )
```

Query the value of an int-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

```
String  get ( GRB.StringParam  param )
```

Query the value of a string-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

GRBEnv.getErrorMsg()

Query the error message for the most recent exception associated with this environment.

```
String  getErrorMsg ( )
```

Return value:

The error string.

GRBEnv.getParamInfo()

Obtain information about a parameter.

```
void  getParamInfo ( GRB.DoubleParam  param,
                     double[]         info )
```

Obtain detailed information about a double parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

info: The returned information. The result will contain four entries: the current value of the parameter, the minimum allowed value, the maximum allowed value, and the default value.

```
void  getParamInfo ( GRB.IntParam  param,
                     int[]         info )
```

Obtain detailed information about an integer parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

info: The returned information. The result will contain four entries: the current value of the parameter, the minimum allowed value, the maximum allowed value, and the default value.

```
void  getParamInfo ( GRB.StringParam  param,
                     String[]         info )
```

Obtain detailed information about a string parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

info: The returned information. The result will contain two entries: the current value of the parameter and the default value.

GRBEnv.message()

Write a message to the console and the log file.

```
void  message ( String  message )
```

Arguments:

message: Print a message to the console and to the log file. Note that this call has no effect unless the `OutputFlag` parameter is set.

GRBEnv.readParams()

Read new parameter settings from a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  readParams ( String paramFile )
```

Arguments:

paramFile: Name of the file containing parameter settings. Parameters should be listed one per line, with the parameter name first and the desired value second. For example:

```
# Gurobi parameter file
Threads 1
MIPGap 0
```

Blank lines and lines that begin with the hash symbol are ignored.

GRBEnv.release()

Release the license associated with this environment. You will no longer be able to call optimize on models created with this environment after the license has been released.

```
void  release ( )
```

GRBEnv.resetParams()

Reset all parameters to their default values.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  resetParams ( )
```

GRBEnv.set()

Set the value of a parameter.

Important notes:

Note that a model gets its own copy of the environment when it is created. Changes to the original environment have no effect on the copy. Use GRBModel.set to change a parameter on an existing model.

```
void  set ( GRB.DoubleParam param,
            double          newval )
```

Set the value of a double-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newval: The desired new value of the parameter.

```
void  set ( GRB.IntParam param,
            int          newval )
```

Set the value of an int-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newval: The desired new value of the parameter.

```
void  set (  GRB.StringParam  param,
             String           newval )
```

Set the value of a string-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newval: The desired new value of the parameter.

```
void  set (  String  param,
             String  newval )
```

Set the value of any parameter using strings alone.

Arguments:

param: The name of the parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newval: The desired new value of the parameter.

GRBEnv.setLogCallback()

Sets a logging callback function to query all output posted by the environment object. Can be set after an empty environment was created.

```
void  setLogCallback (  java.util.function.Consumer<String>  logCallback )
```

Arguments:

logCallback: The logging callback function.

GRBEnv.start()

Start an empty environment. If the environment has already been started, this method will do nothing. If the call fails, the environment will have the same state as it had before the call to this method.

This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by

the desired value for that parameter). After that, it will apply all parameter changes specified by the user prior to this call. Note that this might overwrite parameters set in the license file, or in the `gurobi.env` file, if present.

After all these changes are performed, the code will actually activate the environment, and make it ready to work with models.

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  start ( )
```

GRBEnv.writeParams()

Write all non-default parameter settings to a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  writeParams (  String  paramFile )
```

Arguments:

`paramFile`: Name of the file to which non-default parameter settings should be written. The previous contents are overwritten.

5.2 GRBModel

Gurobi model object. Commonly used methods include addVar (adds a new decision variable to the model), addConstr (adds a new constraint to the model), optimize (optimizes the current model), and get (retrieves the value of an attribute).

While the Java garbage collector will eventually collect an unused `GRBModel` object, the vast majority of the memory associated with a model is stored outside of the Java heap. As a result, the garbage collector can't see this memory usage, and thus it can't take this quantity into account when deciding whether collection is necessary. We recommend that you call GRBModel.dispose when you are done using a model.

GRBModel()

Constructor for `GRBModel`. The simplest version creates an empty model. You can then call addVar and addConstr to populate the model with variables and constraints. The more complex constructors can read a model from a file, or make a copy of an existing model.

```
GRBModel  GRBModel ( GRBEnv  env )
```

Model constructor.

Arguments:

`env`: Environment for new model.

Return value:

New model object. Model initially contains no variables or constraints.

```
GRBModel  GRBModel ( GRBEnv  env,
                     String  filename )
```

Read a model from a file. Note that the type of the file is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, `.rlp`, `.dua`, `.dlp`, `.ilp`, or `.opb`. The files can be compressed, so additional suffixes of `.zip`, `.gz`, `.bz2`, or `.7z` are accepted.

Arguments:

`env`: Environment for new model.

`filename`: Name of the file containing the model.

Return value:

New model object.

```
GRBModel  GRBModel ( GRBModel  model )
```

Create a copy of an existing model. Note that due to the lazy update approach in Gurobi, you have to call update before copying it.

Arguments:

`model`: Model to copy.

Return value:

New model object. Model is a clone of the original.

```
GRBModel  GRBModel (  GRBModel  model,
                      GRBEnv    targetenv )
```

Copy an existing model to a different environment. Multiple threads can not work simultaneously within the same environment. Copies of models must therefore reside in different environments for multiple threads to operate on them simultaneously.

Note that this method itself is not thread safe, so you should either call it from the main thread or protect access to it with a lock.

Note that pending updates will not be applied to the model, so you should call update before copying if you would like those to be included in the copy.

For Compute Server users, note that you can copy a model from a client to a Compute Server environment, but it is not possible to copy models from a Compute Server environment to another (client or Compute Server) environment.

Arguments:

model: Model to copy.

targetenv: Environment to copy model into.

Return value:

New model object. Model is a clone of the original.

GRBModel.addConstr()

Add a single linear constraint to a model. Multiple signatures are available.

```
GRBConstr  addConstr (  GRBLinExpr  lhsExpr,
                        char        sense,
                        GRBLinExpr  rhsExpr,
                        String      name )
```

Add a single linear constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new linear constraint.

sense: Sense for new linear constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsExpr: Right-hand side expression for new linear constraint.

name: Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  GRBLinExpr  lhsExpr,
                        char        sense,
                        GRBVar      rhsVar,
                        String      name )
```

Add a single linear constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new linear constraint.

sense: Sense for new linear constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsVar: Right-hand side variable for new linear constraint.

name: Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  GRBLinExpr  lhsExpr,
                        char        sense,
                        double      rhs,
                        String      name )
```

Add a single linear constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new linear constraint.

sense: Sense for new linear constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhs: Right-hand side value for new linear constraint.

name: Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  GRBVar      lhsVar,
                        char        sense,
                        GRBLinExpr  rhsExpr,
                        String      name )
```

Add a single linear constraint to a model.

Arguments:

lhsVar: Left-hand side variable for new linear constraint.

sense: Sense for new linear constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsExpr: Right-hand side expression for new linear constraint.

name: Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  GRBVar  lhsVar,
                        char    sense,
                        GRBVar  rhsVar,
                        String  name )
```

Add a single linear constraint to a model.

Arguments:

lhsVar: Left-hand side variable for new linear constraint.

sense: Sense for new linear constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsVar: Right-hand side variable for new linear constraint.

name: Name for new constraint.

Return value:
New constraint object.

```
GRBConstr  addConstr (  GRBVar  lhsVar,
                        char    sense,
                        double  rhs,
                        String  name )
```

Add a single linear constraint to a model.

Arguments:

lhsVar: Left-hand side variable for new linear constraint.

sense: Sense for new linear constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhs: Right-hand side value for new linear constraint.

name: Name for new constraint.

Return value:
New constraint object.

```
GRBConstr  addConstr (  double  lhs,
                        char    sense,
                        GRBVar  rhsVar,
                        String  name )
```

Add a single linear constraint to a model.

Arguments:

lhs: Left-hand side value for new linear constraint.

sense: Sense for new linear constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsVar: Right-hand side variable for new linear constraint.

name: Name for new constraint.

Return value:
New constraint object.

```
GRBConstr  addConstr (  double      lhs,
                        char        sense,
                        GRBLinExpr  rhsExpr,
                        String      name )
```

Add a single linear constraint to a model.

Arguments:

lhs: Left-hand side value for new linear constraint.

sense: Sense for new linear constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsExpr: Right-hand side expression for new linear constraint.

name: Name for new constraint.

Return value:
New constraint object.

GRBModel.addConstrs()

Add new linear constraints to a model.

We recommend that you build your model one constraint at a time (using addConstr), since it introduces no significant overhead and we find that it produces simpler code. Feel free to use these methods if you disagree, though.

```
GRBConstr[]  addConstrs (  int  count )
```

Add `count` new linear constraints to a model. The new constraints are all of the form `0 <= 0`.

Arguments:

`count`: Number of constraints to add.

Return value:

Array of new constraint objects.

```
GRBConstr[]  addConstrs (  GRBLinExpr[]  lhsExprs,
                           char[]        senses,
                           double[]      rhss,
                           String[]      names )
```

Add new linear constraints to a model. The number of added constraints is determined by the length of the input arrays (which must be consistent across all arguments).

Arguments:

`lhsExprs`: Left-hand side expressions for the new linear constraints.

`senses`: Senses for new linear constraints (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

`rhss`: Right-hand side values for the new linear constraints.

`names`: Names for new constraints.

Return value:

Array of new constraint objects.

```
GRBConstr[]  addConstrs (  GRBLinExpr[]  lhsExprs,
                           char[]        senses,
                           double[]      rhss,
                           String[]      names,
                           int           start,
                           int           len )
```

Add new linear constraints to a model. This signature allows you to use arrays to hold the various constraint attributes (left-hand side, sense, etc.), without forcing you to add one constraint for each entry in the array. The `start` and `len` arguments allow you to specify which constraints to add.

Arguments:

`lhsExprs`: Left-hand side expressions for the new linear constraints.

`senses`: Senses for new linear constraints (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

rhss: Right-hand side values for the new linear constraints.
names: Names for new constraints.
start: The first constraint in the list to add.
len: The number of constraints to add.

Return value:

Array of new constraint objects.

GRBModel.addGenConstrXxx()

Each of the functions described below adds a new *general constraint* to a model.

Mathematical programming has traditionally defined a set of fundamental constraint types: variable bound constraints, linear constraints, quadratic constraints, integrality constraints, and SOS constraints. These are typically treated directly by the underlying solver (although not always), and are fundamental to the overall algorithm.

Gurobi accepts a number of additional constraint types, which we collectively refer to as *general (function) constraints*. These are typically *not* treated directly by the solver. Rather, they are transformed by presolve into constraints (and variables) chosen from among the fundamental types listed above. In some cases, the resulting constraint or constraints are mathematically equivalent to the original; in others, they are approximations. If such constraints appear in your model, but if you prefer to reformulate them yourself using fundamental constraint types instead, you can certainly do so. However, note that Gurobi can sometimes exploit information contained in the other constraints in the model to build a more efficient formulation than what you might create.

The additional constraint types that fall under this *general constraint* umbrella are:

- addGenConstrMax: $y = \max(x_1, x_2, ..., c)$
- addGenConstrMin: $y = \min(x_1, x_2, ..., c)$
- addGenConstrAbs: $y = |x|$
- addGenConstrAnd: $y = x_1 \wedge x_2 \wedge x_3 ...$
- addGenConstrOr: $y = x_1 \vee x_2 \vee x_3 ...$
- addGenConstrNorm: $y = norm(x_1, x_2, x_3 ...)$
- addGenConstrIndicator: $y = 1 \rightarrow a'x \leq b$ (an indicator constraint)
- addGenConstrPWL: $y = pwl(x)$ (a piecewise-linear function, specified using breakpoints)
- addGenConstrPoly: $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$
- addGenConstrExp: $y = e^x$
- addGenConstrExpA: $y = a^x$
- addGenConstrLog: $y = \log_e(x)$
- addGenConstrLogA: $y = \log_a(x)$
- addGenConstrLogistic: $y = \frac{1}{1+e^{-x}}$

- addGenConstrPow: $y = x^a$
- addGenConstrSin: $y = \sin(x)$
- addGenConstrCos: $y = \cos(x)$
- addGenConstrTan: $y = \tan(x)$

For additional details please refer to the General Constraints section in the Reference Manual.

GRBModel.addGenConstrMax()

Add a new general constraint of type `GRB.GENCONSTR_MAX` to a model.

A MAX constraint $r = \max\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the maximum of the operand variables $x_1, \ldots, x_n$ and the constant c.

```
GRBGenConstr  addGenConstrMax (  GRBVar    resvar,
                                 GRBVar[]  vars,
                                 double    constant,
                                 String    name )
```

Arguments:

`resvar`: The resultant variable of the new constraint.
`vars`: Array of variables that are the operands of the new constraint.
`constant`: The additional constant operand of the new constraint.
`name`: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.addGenConstrMin()

Add a new general constraint of type `GRB.GENCONSTR_MIN` to a model.

A MIN constraint $r = \min\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the minimum of the operand variables $x_1, \ldots, x_n$ and the constant c.

```
GRBGenConstr  addGenConstrMin (  GRBVar    resvar,
                                 GRBVar[]  vars,
                                 double    constant,
                                 String    name )
```

Arguments:

`resvar`: The resultant variable of the new constraint.
`vars`: Array of variables that are the operands of the new constraint.
`constant`: The additional constant operand of the new constraint.
`name`: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.addGenConstrAbs()

Add a new general constraint of type `GRB.GENCONSTR_ABS` to a model.

An ABS constraint $r = \text{abs}\{x\}$ states that the resultant variable r should be equal to the absolute value of the argument variable x.

```
GRBGenConstr  addGenConstrAbs (  GRBVar  resvar,
                                 GRBVar  argvar,
                                 String  name )
```

Arguments:

resvar: The resultant variable of the new constraint.

argvar: The argument variable of the new constraint.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.addGenConstrAnd()

Add a new general constraint of type `GRB.GENCONSTR_AND` to a model.

An AND constraint $r = \text{and}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if all of the operand variables $x_1, \ldots, x_n$ are equal to 1. If any of the operand variables is 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

```
GRBGenConstr  addGenConstrAnd (  GRBVar    resvar,
                                 GRBVar[]  vars,
                                 String    name )
```

Arguments:

resvar: The resultant variable of the new constraint.

vars: Array of variables that are the operands of the new constraint.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.addGenConstrOr()

Add a new general constraint of type `GRB.GENCONSTR_OR` to a model.

An OR constraint $r = \text{or}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if any of the operand variables $x_1, \ldots, x_n$ is equal to 1. If all operand variables are 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

```
GRBGenConstr  addGenConstrOr (  GRBVar    resvar,
                                GRBVar[]  vars,
                                String    name )
```

Arguments:

resvar: The resultant variable of the new constraint.

vars: Array of variables that are the operands of the new constraint.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.addGenConstrNorm()

Add a new general constraint of type GRB.GENCONSTR_NORM to a model.

A NORM constraint $r = \text{norm}\{x_1, \ldots, x_n\}$ states that the resultant variable r should be equal to the vector norm of the argument vector $x_1, \ldots, x_n$.

```
GRBGenConstr  addGenConstrNorm (  GRBVar    resvar,
                                  GRBVar[]  vars,
                                  double    which,
                                  String    name )
```

Arguments:

resvar: The resultant variable of the new constraint.

vars: Array of variables that are the operands of the new constraint. Note that this array may not contain duplicates.

which: Which norm to use. Options are 0, 1, 2, and GRB.INFINITY.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.addGenConstrIndicator()

Add a new general constraint of type GRB.GENCONSTR_INDICATOR to a model.

An INDICATOR constraint $z = f \rightarrow a^T x \leq b$ states that if the binary indicator variable z is equal to f, where $f \in \{0, 1\}$, then the linear constraint $a^T x \leq b$ should hold. On the other hand, if $z = 1 - f$, the linear constraint may be violated. The sense of the linear constraint can also be specified to be $=$ or $\geq$.

Note that the indicator variable z of a constraint will be forced to be binary, independent of how it was created.

```
GRBGenConstr  addGenConstrIndicator (  GRBVar      binvar,
                                       int         binval,
                                       GRBLinExpr  expr,
                                       char        sense,
                                       double      rhs,
                                       String      name )
```

Arguments:

binvar: The binary indicator variable.

binval: The value for the binary indicator variable that would force the linear constraint to be satisfied (0 or 1).

expr: Left-hand side expression for the linear constraint triggered by the indicator.

sense: Sense for the linear constraint. Options are `GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`.

rhs: Right-hand side value for the linear constraint.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.addGenConstrPWL()

Add a new general constraint of type `GRB.GENCONSTR_PWL` to a model.

A piecewise-linear (PWL) constraint states that the relationship $y = f(x)$ must hold between variables x and y, where f is a piecewise-linear function. The breakpoints for f are provided as arguments. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined.

```
GRBGenConstr  addGenConstrPWL (  GRBVar    xvar,
                                 GRBVar    yvar,
                                 double[]  xpts,
                                 double[]  ypts,
                                 String    name )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

xpts: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

ypts: The y values for the points that define the piecewise-linear function.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.addGenConstrPoly()

Add a new general constraint of type `GRB.GENCONSTR_POLY` to a model.

A polynomial function constraint states that the relationship $y = p_0x^d + p_1x^{d-1} + ... + p_{d-1}x + p_d$ should hold between variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrPoly (  GRBVar    xvar,
                                  GRBVar    yvar,
                                  double[]  p,
                                  String    name,
                                  String    options )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

`p`: The coefficients for the polynomial function (starting with the coefficient for the highest power).

`name`: Name for the new general constraint.

`options`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrExp()

Add a new general constraint of type `GRB.GENCONSTR_EXP` to a model.

A natural exponential function constraint states that the relationship $y = exp(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrExp (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 String  name,
                                 String  options )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

`name`: Name for the new general constraint.

`options`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrExpA()

Add a new general constraint of type `GRB.GENCONSTR_EXPA` to a model.

An exponential function constraint states that the relationship $y = a^x$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrExpA (  GRBVar  xvar,
                                  GRBVar  yvar,
                                  double  a,
                                  String  name,
                                  String  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

a: The base of the function, $a > 0$.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrLog()

Add a new general constraint of type GRB.GENCONSTR_LOG to a model.

A natural logarithmic function constraint states that the relationship $y = log(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrLog (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 String  name,
                                 String  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the

attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrLogA()

Add a new general constraint of type `GRB.GENCONSTR_LOGA` to a model.

A logarithmic function constraint states that the relationship $y = log_a(x)$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrLogA (  GRBVar  xvar,
                                  GRBVar  yvar,
                                  double  a,
                                  String  name,
                                  String  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

a: The base of the function, $a > 0$.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrLogistic()

Add a new general constraint of type `GRB.GENCONSTR_LOGISTIC` to a model.

A logistic function constraint states that the relationship $y = \frac{1}{1+e^{-x}}$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrLogistic (  GRBVar  xvar,
                                      GRBVar  yvar,
                                      String  name,
                                      String  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrPow()

Add a new general constraint of type `GRB.GENCONSTR_POW` to a model.

A power function constraint states that the relationship $y = x^a$ should hold for variables x and y, where a is the (constant) exponent.

If the exponent a is negative, the lower bound on x must be strictly positive. If the exponent isn't an integer, the lower bound on x must be non-negative.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrPow (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 double  a,
                                 String  name,
                                 String  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

a: The exponent of the function.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrSin()

Add a new general constraint of type `GRB.GENCONSTR_SIN` to a model.

A sine function constraint states that the relationship $y = sin(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrSin (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 String  name,
                                 String  options )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

`name`: Name for the new general constraint.

`options`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrCos()

Add a new general constraint of type `GRB.GENCONSTR_COS` to a model.

A cosine function constraint states that the relationship $y = cos(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrCos (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 String  name,
                                 String  options )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addGenConstrTan()

Add a new general constraint of type `GRB.GENCONSTR_TAN` to a model.

A tangent function constraint states that the relationship $y = tan(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  addGenConstrTan (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 String  name,
                                 String  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.addQConstr()

Add a quadratic constraint to a model. Multiple signatures are available.

Important note: Gurobi can handle both convex and non-convex quadratic constraints. The differences between them can be both important and subtle. Refer to this discussion for additional information.

```
GRBQConstr  addQConstr (  GRBQuadExpr  lhsExpr,
                          char         sense,
                          GRBQuadExpr  rhsExpr,
                          String       name )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side quadratic expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsExpr: Right-hand side quadratic expression for new quadratic constraint.

name: Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  GRBQuadExpr  lhsExpr,
                          char         sense,
                          GRBVar       rhsVar,
                          String       name )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side quadratic expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsVar: Right-hand side variable for new quadratic constraint.

name: Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  GRBQuadExpr  lhsExpr,
                          char         sense,
                          GRBLinExpr   rhsExpr,
                          String       name )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side quadratic expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsExpr: Right-hand side linear expression for new quadratic constraint.

name: Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  GRBQuadExpr  lhsExpr,
                          char         sense,
                          double       rhs,
                          String       name )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side quadratic expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhs: Right-hand side value for new quadratic constraint.

name: Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  GRBLinExpr   lhsExpr,
                          char         sense,
                          GRBQuadExpr  rhsExpr,
                          String       name )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side linear expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsExpr: Right-hand side quadratic expression for new quadratic constraint.

name: Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  GRBVar       lhsVar,
                          char         sense,
                          GRBQuadExpr  rhsExpr,
                          String       name )
```

Add a quadratic constraint to a model.

Arguments:

lhsVar: Left-hand side variable for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsExpr: Right-hand side quadratic expression for new quadratic constraint.

name: Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  double       lhs,
                          char         sense,
                          GRBQuadExpr  rhsExpr,
                          String       name )
```

Add a quadratic constraint to a model.

Arguments:

lhs: Left-hand side value for new quadratic constraint.

sense: Sense for new quadratic constraint (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

rhsExpr: Right-hand side quadratic expression for new quadratic constraint.

name: Name for new constraint.

Return value:

New quadratic constraint object.

GRBModel.addRange()

Add a single range constraint to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution.

Note that range constraints are stored internally as equality constraints. We add an extra variable to the model to capture the range information. Thus, the Sense attribute on a range constraint will always be `GRB.EQUAL`. In particular introducing a range constraint

$$L \leq a^T x \leq U$$

is equivalent to adding a slack variable s and the following constraints

$$\begin{aligned} a^T x - s &= L \\ 0 \leq s &\leq U - L. \end{aligned}$$

```
GRBConstr  addRange (  GRBLinExpr  expr,
                       double      lower,
                       double      upper,
                       String      name )
```

Arguments:

expr: Linear expression for new range constraint.

lower: Lower bound for linear expression.

upper: Upper bound for linear expression.

name: Name for new constraint.

Return value:

New constraint object.

GRBModel.addRanges()

Add new range constraints to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution.

```
GRBConstr[]  addRanges (  GRBLinExpr[]  exprs,
                          double[]      lower,
                          double[]      upper,
                          String[]      names )
```

Arguments:

exprs: Linear expressions for the new range constraints.

lower: Lower bounds for linear expressions.

upper: Upper bounds for linear expressions.

names: Names for new range constraints.

Return value:

Array of new constraint objects.

GRBModel.addSOS()

Add an SOS constraint to the model. Please refer to the SOS Constraints section in the Reference Manual for additional details.

```
GRBSOS  addSOS (  GRBVar[]  vars,
                  double[]  weights,
                  int       type )
```

Arguments:

vars: Array of variables that participate in the SOS constraint.

weights: Weights for the variables in the SOS constraint.

type: SOS type (can be GRB.SOS_TYPE1 or GRB.SOS_TYPE2).

Return value:

New SOS constraint.

GRBModel.addVar()

Add a single decision variable to a model.

```
GRBVar  addVar (  double  lb,
                  double  ub,
                  double  obj,
                  char    type,
                  String  name )
```

Add a variable to a model; non-zero entries will be added later.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB.CONTINUOUS, GRB.BINARY, GRB.INTEGER, GRB.SEMICONT, or GRB.SEMIINT).

name: Name for new variable.

Return value:

New variable object.

```
GRBVar  addVar (  double       lb,
                  double       ub,
                  double       obj,
                  char         type,
                  GRBConstr[]  constrs,
                  double[]     coeffs,
                  String       name )
```

Add a variable to a model, and the associated non-zero coefficients.

Arguments:

`lb`: Lower bound for new variable.

`ub`: Upper bound for new variable.

`obj`: Objective coefficient for new variable.

`type`: Variable type for new variable (`GRB.CONTINUOUS`, `GRB.BINARY`, `GRB.INTEGER`, `GRB.SEMICONT`, or `GRB.SEMIINT`).

`constrs`: Array of constraints in which the variable participates.

`coeffs`: Array of coefficients for each constraint in which the variable participates. The lengths of the `constrs` and `coeffs` arrays must be identical.

`name`: Name for new variable.

Return value:

New variable object.

```
GRBVar  addVar (  double     lb,
                  double     ub,
                  double     obj,
                  char       type,
                  GRBColumn  col,
                  String     name )
```

Add a variable to a model. This signature allows you to specify the set of constraints to which the new variable belongs using a GRBColumn object.

Arguments:

`lb`: Lower bound for new variable.

`ub`: Upper bound for new variable.

`obj`: Objective coefficient for new variable.

`type`: Variable type for new variable (`GRB.CONTINUOUS`, `GRB.BINARY`, `GRB.INTEGER`, `GRB.SEMICONT`, or `GRB.SEMIINT`).

`col`: GRBColumn object for specifying a set of constraints to which new variable belongs.

`name`: Name for new variable.

Return value:

New variable object.

GRBModel.addVars()

Add new decision variables to a model.

```
GRBVar[]  addVars (  int   count,
                     char  type )
```

Add `count` new decision variables to a model. All associated attributes take their default values, except the variable `type`, which is specified as an argument.

Arguments:

`count`: Number of variables to add.

`type`: Variable type for new variables (`GRB.CONTINUOUS`, `GRB.BINARY`, `GRB.INTEGER`, `GRB.SEMICONT`, or `GRB.SEMIINT`).

Return value:

Array of new variable objects.

```
GRBVar[]  addVars (  double[]  lb,
                     double[]  ub,
                     double[]  obj,
                     char[]    type,
                     String[]  names )
```

Add new decision variables to a model. The number of added variables is determined by the length of the input arrays (which must be consistent across all arguments).

Arguments:

`lb`: Lower bounds for new variables. Can be `null`, in which case the variables get lower bounds of 0.0.

`ub`: Upper bounds for new variables. Can be `null`, in which case the variables get infinite upper bounds.

`obj`: Objective coefficients for new variables. Can be `null`, in which case the variables get objective coefficients of 0.0.

`type`: Variable types for new variables (`GRB.CONTINUOUS`, `GRB.BINARY`, `GRB.INTEGER`, `GRB.SEMICONT`, or `GRB.SEMIINT`). Can be `null`, in which case the variables are assumed to be continuous.

`names`: Names for new variables. Can be `null`, in which case all variables are given default names.

Return value:

Array of new variable objects.

```
GRBVar[]  addVars (  double[]  lb,
                     double[]  ub,
                     double[]  obj,
                     char[]    type,
                     String[]  names,
                     int       start,
                     int       len )
```

Add new decision variables to a model. This signature allows you to use arrays to hold the various variable attributes (lower bound, upper bound, etc.), without forcing you to add a variable for each entry in the array. The `start` and `len` arguments allow you to specify which variables to add.

Arguments:

`lb`: Lower bounds for new variables. Can be `null`, in which case the variables get lower bounds of 0.0.

ub: Upper bounds for new variables. Can be **null**, in which case the variables get infinite upper bounds.

obj: Objective coefficients for new variables. Can be **null**, in which case the variables get objective coefficients of 0.0.

type: Variable types for new variables (**GRB.CONTINUOUS**, **GRB.BINARY**, **GRB.INTEGER**, **GRB.SEMICONT**, or **GRB.SEMIINT**). Can be **null**, in which case the variables are assumed to be continuous.

names: Names for new variables. Can be **null**, in which case all variables are given default names.

start: The first variable in the list to add.

len: The number of variables to add.

Return value:

Array of new variable objects.

```
GRBVar[]  addVars (  double[]     lb,
                     double[]     ub,
                     double[]     obj,
                     char[]       type,
                     String[]     names,
                     GRBColumn[]  cols )
```

Add new decision variables to a model. This signature allows you to specify the list of constraints to which each new variable belongs using an array of GRBColumn objects.

Arguments:

lb: Lower bounds for new variables. Can be **null**, in which case the variables get lower bounds of 0.0.

ub: Upper bounds for new variables. Can be **null**, in which case the variables get infinite upper bounds.

obj: Objective coefficients for new variables. Can be **null**, in which case the variables get objective coefficients of 0.0.

type: Variable types for new variables (**GRB.CONTINUOUS**, **GRB.BINARY**, **GRB.INTEGER**, **GRB.SEMICONT**, or **GRB.SEMIINT**). Can be **null**, in which case the variables are assumed to be continuous.

names: Names for new variables. Can be **null**, in which case all variables are given default names.

cols: GRBColumn objects for specifying a set of constraints to which each new column belongs.

Return value:

Array of new variable objects.

GRBModel.chgCoeff()

Change one coefficient in the model. The desired change is captured using a GRBVar object, a GRBConstr object, and a desired coefficient for the specified variable in the specified constraint. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.update), optimize the model (using GRBModel.optimize), or write the model to disk (using GRBModel.write).

```
void  chgCoeff (  GRBConstr  constr,
                  GRBVar     var,
                  double     newval )
```

Arguments:

constr: Constraint for coefficient to be changed.
var: Variable for coefficient to be changed.
newval: Desired new value for coefficient.

GRBModel.chgCoeffs()

Change a list of coefficients in the model. Each desired change is captured using a `GRBVar` object, a `GRBConstr` object, and a desired coefficient for the specified variable in the specified constraint. The entries in the input arrays each correspond to a single desired coefficient change. The lengths of the input arrays must all be the same. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.update), optimize the model (using GRBModel.optimize), or write the model to disk (using GRBModel.write).

```
void  chgCoeffs (  GRBConstr[]  constrs,
                   GRBVar[]     vars,
                   double[]     newvals )
```

Arguments:

constrs: Constraints for coefficients to be changed.
vars: Variables for coefficients to be changed.
newvals: Desired new values for coefficients.

GRBModel.computeIIS()

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the smallest one; there may exist others with fewer constraints or bounds.

IIS results are returned in a number of attributes: IISConstr, IISLB, IISUB, IISSOS, IISQConstr, and IISGenConstr. Each indicates whether the corresponding model element is a member of the computed IIS.

Note that for models with general function constraints, piecewise-linear approximation of the constraints may cause unreliable IIS results.

The IIS log provides information about the progress of the algorithm, including a guess at the eventual IIS size.

If an IIS computation is interrupted before completion, Gurobi will return the smallest infeasible subsystem found to that point.

The IISConstrForce, IISLBForce, IISUBForce, IISSOSForce, IISQConstrForce, and IISGenConstrForce attributes allow you mark model elements to either include or exclude from the computed IIS. Setting the attribute to 1 forces the corresponding element into the IIS, setting it to 0 forces it out of the IIS, and setting it to -1 allows the algorithm to decide.

To give an example of when these attributes might be useful, consider the case where an initial model is known to be feasible, but it becomes infeasible after adding constraints or tightening bounds. If you are only interested in knowing which of the changes caused the infeasibility, you can force the unmodified bounds and constraints into the IIS. That allows the IIS algorithm to focus exclusively on the new constraints, which will often be substantially faster.

Note that setting any of the `Force` attributes to 0 may make the resulting subsystem feasible, which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

This method populates the IISConstr, IISQConstr, and IISGenConstr constraint attributes, the IISSOS, SOS attribute, and the IISLB and IISUB variable attributes. You can also obtain information about the results of the IIS computation by writing a `.ilp` format file (see GRBModel.write). This file contains only the IIS from the original model.

Use the IISMethod parameter to adjust the behavior of the IIS algorithm.

Note that this method can be used to compute IISs for both continuous and MIP models.

```
void  computeIIS ( )
```

GRBModel.discardConcurrentEnvs()

Discard concurrent environments for a model.

The concurrent environments created by getConcurrentEnv will be used by every subsequent call to the concurrent optimizer until the concurrent environments are discarded.

Use getMultiobjEnv to create a multi-objective environment.

```
void  discardConcurrentEnvs ( )
```

GRBModel.discardMultiobjEnvs()

Discard all multi-objective environments associated with the model, thus restoring multi objective optimization to its default behavior.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Use getMultiobjEnv to create a multi-objective environments.

```
void  discardMultiobjEnvs ( )
```

GRBModel.dispose()

Release the resources associated with a `GRBModel` object. While the Java garbage collector will eventually reclaim these resources, we recommend that you call the `dispose` method when you are done using a model.

You should not attempt to use a `GRBModel` object after calling `dispose` on it.

```
void  dispose ( )
```

GRBModel.feasRelax()

Modifies the `GRBModel` object to create a feasibility relaxation. Note that you need to call optimize on the result to compute the actual relaxed solution.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This method provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost per unit violation in the lower bounds, upper bounds, and linear constraints, respectively.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the coefficients on the squares of the lower bound, upper bound, and linear constraint violations, respectively.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the weighted count of bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost of violating a lower bound, upper bound, and linear constraint, respectively.

To give an example, if a constraint with `rhspen` value `p` is violated by 2.0, it would contribute `2*p` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2*2*p` for `relaxobjtype=1`, and it would contribute `p` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=false`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=true`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `feasRelax` must solve an optimization problem to find the minimum possible relaxation when `minrelax=true`, which can be quite expensive.

There are two signatures for this method. The more complex one takes a list of variables and constraints, as well as penalties associated with relaxing the corresponding lower bounds, upper bounds, and constraints. If a variable or constraint is not included in one of these lists, the associated bounds or constraints may not be violated. The simpler signature takes a pair of boolean arguments, `vrelax` and `crelax`, that indicate whether variable bounds and/or constraints can be violated. If `vrelax`/`crelax` is `true`, then every bound/constraint is allowed to be violated, respectively, and the associated cost is 1.0.

Note that this is a destructive method: it modifies the model on which it is invoked. If you don't want to modify your original model, use the GRBModel constructor to create a copy before invoking this method.

```
double  feasRelax (  int          relaxobjtype,
                     boolean      minrelax,
                     GRBVar[]     vars,
                     double[]     lbpen,
                     double[]     ubpen,
                     GRBConstr[]  constrs,
                     double[]     rhspen )
```

Create a feasibility relaxation model.

Arguments:

relaxobjtype: The cost function used when finding the minimum cost relaxation.

minrelax: The type of feasibility relaxation to perform.

vars: Variables whose bounds are allowed to be violated.

lbpen: Penalty for violating a variable lower bound. One entry for each variable in argument vars.

ubpen: Penalty for violating a variable upper bound. One entry for each variable in argument vars.

constrs: Linear constraints that are allowed to be violated.

rhspen: Penalty for violating a linear constraint. One entry for each constraint in argument constrs.

Arguments:

Return value:

Zero if minrelax is false. If minrelax is true, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

```
double  feasRelax (  int      relaxobjtype,
                     boolean  minrelax,
                     boolean  vrelax,
                     boolean  crelax )
```

Simplified method for creating a feasibility relaxation model.

Arguments:

relaxobjtype: The cost function used when finding the minimum cost relaxation.

minrelax: The type of feasibility relaxation to perform.

vrelax: Indicates whether variable bounds can be relaxed (with a cost of 1.0 for any violations.

crelax: Indicates whether linear constraints can be relaxed (with a cost of 1.0 for any violations.

Return value:

Zero if minrelax is false. If minrelax is true, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

GRBModel.fixedModel()

Create the fixed model associated with a MIP model. The MIP model must have a solution loaded (e.g., after a call to the optimize method). In the fixed model, each integer variable is fixed to the value that variable takes in the MIP solution. In addition, continuous variables may be fixed to satisfy SOS or general constraints. The result is a model without any integrality constraints, SOS constraints, or general constraints.

Note that, while the fixed problem is always a continuous model, it may contain a non-convex quadratic objective or non-convex quadratic constraints. As a result, it may still be solved using the MIP algorithm.

```
GRBModel  fixedModel ( )
```

Return value:

Fixed model associated with calling object.

GRBModel.get()

Query the value(s) of a parameter or attribute. Use this method for parameters, for scalar model attributes, and for arrays of constraint or variable attributes.

```
double  get (  GRB.DoubleParam  param )
```

Query the value of a double-valued parameter.

Arguments:

param: The parameter being queried.

Return value:

The current value of the requested parameter.

```
int  get (  GRB.IntParam  param )
```

Query the value of an int-valued parameter.

Arguments:

param: The parameter being queried.

Return value:

The current value of the requested parameter.

```
String  get (  GRB.StringParam  param )
```

Query the value of a string-valued parameter.

Arguments:

param: The parameter being queried.

Return value:

The current value of the requested parameter.

```
char[]  get (  GRB.CharAttr  attr,
               GRBVar[]      vars )
```
Query a char-valued variable attribute for an array of variables.

Arguments:

`attr`: The attribute being queried.

`vars`: The variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
char[]  get (  GRB.CharAttr  attr,
               GRBVar[]      vars,
               int           start,
               int           len )
```
Query a char-valued variable attribute for a sub-array of variables.

Arguments:

`attr`: The attribute being queried.

`vars`: A one-dimensional array of variables whose attribute values are being queried.

`start`: The index of the first variable of interest in the list.

`len`: The number of variables.

Return value:

The current values of the requested attribute for each input variable.

```
char[][]  get (  GRB.CharAttr  attr,
                 GRBVar[][]    vars )
```
Query a char-valued variable attribute for a two-dimensional array of variables.

Arguments:

`attr`: The attribute being queried.

`vars`: A two-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
char[][][]  get (  GRB.CharAttr  attr,
                   GRBVar[][][]  vars )
```
Query a char-valued variable attribute for a three-dimensional array of variables.

Arguments:

`attr`: The attribute being queried.

`vars`: A three-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
char[]  get (  GRB.CharAttr  attr,
               GRBConstr[]   constrs )
```
Query a char-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: The constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
char[]  get (  GRB.CharAttr  attr,
               GRBConstr[]   constrs,
               int           start,
               int           len )
```

Query a char-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A one-dimensional array of constraints whose attribute values are being queried.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

Return value:

The current values of the requested attribute for each input constraint.

```
char[][]  get (  GRB.CharAttr   attr,
                 GRBConstr[][]  constrs )
```

Query a char-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A two-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
char[][][]  get (  GRB.CharAttr     attr,
                   GRBConstr[][][]  constrs )
```

Query a char-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A three-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
char[]  get (  GRB.CharAttr  attr,
               GRBQConstr[]  qconstrs )
```

Query a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: The quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
char[]  get (  GRB.CharAttr  attr,
               GRBQConstr[]  qconstrs,
               int           start,
               int           len )
```

Query a char-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being queried.

start: The index of the first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
char[][]  get (  GRB.CharAttr    attr,
                 GRBQConstr[][]  qconstrs )
```

Query a char-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
char[][][]  get (  GRB.CharAttr      attr,
                   GRBQConstr[][][]  qconstrs )
```

Query a char-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
double  get (  GRB.DoubleAttr  attr )
```

Query the value of a double-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
double[]  get (  GRB.DoubleAttr  attr,
                 GRBVar[]        vars )
```

Query a double-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: The variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
double[]  get (  GRB.DoubleAttr  attr,
                 GRBVar[]        vars,
                 int             start,
                 int             len )
```

Query a double-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being queried.

vars: A one-dimensional array of variables whose attribute values are being queried.

start: The index of the first variable of interest in the list.

len: The number of variables.

Return value:

The current values of the requested attribute for each input variable.

```
double[][]  get (  GRB.DoubleAttr  attr,
                   GRBVar[][]      vars )
```

Query a double-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A two-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
double[][][]  get (  GRB.DoubleAttr  attr,
                     GRBVar[][][]    vars )
```

Query a double-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A three-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
double[]  get (  GRB.DoubleAttr  attr,
                 GRBConstr[]     constrs )
```

Query a double-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: The constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
double[]  get (  GRB.DoubleAttr  attr,
                 GRBConstr[]     constrs,
                 int             start,
                 int             len )
```

Query a double-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A one-dimensional array of constraints whose attribute values are being queried.

start: The first constraint of interest in the list.

len: The number of constraints.

Return value:

The current values of the requested attribute for each input constraint.

```
double[][]  get (  GRB.DoubleAttr  attr,
                   GRBConstr[][]   constrs )
```

Query a double-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A two-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
double[][][]  get (  GRB.DoubleAttr   attr,
                     GRBConstr[][][]  constrs )
```

Query a double-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A three-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
double[]  get (  GRB.DoubleAttr  attr,
                 GRBQConstr[]    qconstrs )
```

Query a double-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: The quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
double[]  get (  GRB.DoubleAttr  attr,
                 GRBQConstr[]    qconstrs,
                 int             start,
                 int             len )
```

Query a double-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being queried.

start: The first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
double[][]  get (  GRB.DoubleAttr  attr,
                   GRBQConstr[][]  qconstrs )
```

Query a double-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
double[][][]  get (  GRB.DoubleAttr    attr,
                     GRBQConstr[][][]  qconstrs )
```

Query a double-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
int  get (  GRB.IntAttr  attr )
```

Query the value of an int-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
int[]  get (  GRB.IntAttr  attr,
              GRBVar[]     vars )
```

Query an int-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: The variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
int[]  get (  GRB.IntAttr  attr,
              GRBVar[]     vars,
              int          start,
              int          len )
```

Query an int-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being queried.

vars: A one-dimensional array of variables whose attribute values are being queried.

start: The index of the first variable of interest in the list.

len: The number of variables.

Return value:

The current values of the requested attribute for each input variable.

```
int[][]  get (  GRB.IntAttr  attr,
                GRBVar[][]   vars )
```

Query an int-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A two-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
int[][][]  get (  GRB.IntAttr   attr,
                  GRBVar[][][]  vars )
```

Query an int-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A three-dimensional array of variables whose attribute values are being queried.

Return value:
The current values of the requested attribute for each input variable.

```
int[]  get (  GRB.IntAttr  attr,
              GRBConstr[]  constrs )
```

Query an int-valued constraint attribute for an array of constraints.
Arguments:
attr: The attribute being queried.
constrs: The constraints whose attribute values are being queried.
Return value:
The current values of the requested attribute for each input constraint.

```
int[]  get (  GRB.IntAttr  attr,
              GRBConstr[]  constrs,
              int          start,
              int          len )
```

Query an int-valued constraint attribute for a sub-array of constraints.
Arguments:
attr: The attribute being queried.
constrs: A one-dimensional array of constraints whose attribute values are being queried.
start: The index of the first constraint of interest in the list.
len: The number of constraints.
Return value:
The current values of the requested attribute for each input constraint.

```
int[][]  get (  GRB.IntAttr    attr,
                GRBConstr[][]  constrs )
```

Query an int-valued constraint attribute for a two-dimensional array of constraints.
Arguments:
attr: The attribute being queried.
constrs: A two-dimensional array of constraints whose attribute values are being queried.
Return value:
The current values of the requested attribute for each input constraint.

```
int[][][]  get (  GRB.IntAttr      attr,
                  GRBConstr[][][]  constrs )
```

Query an int-valued constraint attribute for a three-dimensional array of constraints.
Arguments:
attr: The attribute being queried.
constrs: A three-dimensional array of constraints whose attribute values are being queried.
Return value:
The current values of the requested attribute for each input constraint.

```
int[]  get (  GRB.IntAttr   attr,
              GRBQConstr[]  qconstrs )
```

Query an int-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: The quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
int[]  get (  GRB.IntAttr   attr,
              GRBQConstr[]  qconstrs,
              int           start,
              int           len )
```

Query an int-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being queried.

start: The index of the first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
int[][]  get (  GRB.IntAttr     attr,
                GRBQConstr[][]  qconstrs )
```

Query an int-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
int[][][]  get (  GRB.IntAttr       attr,
                  GRBQConstr[][][]  qconstrs )
```

Query an int-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
int[]  get (  GRB.IntAttr      attr,
              GRBGenConstr[]   genconstrs )
```

Query an int-valued general constraint attribute for an array of general constraints.

Arguments:

attr: The attribute being queried.

genconstrs: The general constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input general constraint.

```
int[]  get (  GRB.IntAttr      attr,
              GRBGenConstr[]   genconstrs,
              int              start,
              int              len )
```

Query an int-valued general constraint attribute for a sub-array of general constraints.

Arguments:

attr: The attribute being queried.

genconstrs: A one-dimensional array of general constraints whose attribute values are being queried.

start: The index of the first general constraint of interest in the list.

len: The number of general constraints.

Return value:

The current values of the requested attribute for each input general constraint.

```
int[][]  get (  GRB.IntAttr       attr,
                GRBGenConstr[][]  genconstrs )
```

Query an int-valued general constraint attribute for a two-dimensional array of general constraints.

Arguments:

attr: The attribute being queried.

genconstrs: A two-dimensional array of general constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input general constraint.

```
int[][][]  get (  GRB.IntAttr         attr,
                  GRBGenConstr[][][]  genconstrs )
```

Query an int-valued general constraint attribute for a three-dimensional array of general constraints.

Arguments:

attr: The attribute being queried.

genconstrs: A three-dimensional array of general constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input general constraint.

```
String  get (  GRB.StringAttr  attr )
```

Query the value of a string-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
String[]  get (  GRB.StringAttr  attr,
                 GRBVar[]        vars )
```

Query a String-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: The variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
String[]  get (  GRB.StringAttr  attr,
                 GRBVar[]        vars,
                 int             start,
                 int             len )
```

Query a String-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being queried.

vars: A one-dimensional array of variables whose attribute values are being queried.

start: The index of the first variable of interest in the list.

len: The number of variables.

Return value:

The current values of the requested attribute for each input variable.

```
String[][]  get (  GRB.StringAttr  attr,
                   GRBVar[][]      vars )
```

Query a String-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A two-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
String[][][]  get (  GRB.StringAttr  attr,
                     GRBVar[][][]    vars )
```

Query a String-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A three-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
String[]  get (  GRB.StringAttr  attr,
                 GRBConstr[]     constrs )
```

Query a String-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: The constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
String[]  get (  GRB.StringAttr  attr,
                 GRBConstr[]     constrs,
                 int             start,
                 int             len )
```

Query a String-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A one-dimensional array of constraints whose attribute values are being queried.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

Return value:

The current values of the requested attribute for each input constraint.

```
String[][]  get (  GRB.StringAttr  attr,
                   GRBConstr[][]   constrs )
```

Query a String-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A two-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
String[][][]  get (  GRB.StringAttr   attr,
                     GRBConstr[][][]  constrs )
```

Query a String-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A three-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
String[]  get (  GRB.StringAttr  attr,
                 GRBQConstr[]    qconstrs )
```

Query a String-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: The quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
String[]  get (  GRB.StringAttr  attr,
                 GRBQConstr[]    qconstrs,
                 int             start,
                 int             len )
```

Query a String-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being queried.

start: The index of the first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
String[][]  get (  GRB.StringAttr  attr,
                   GRBQConstr[][]  qconstrs )
```

Query a String-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
String[][][]  get (  GRB.StringAttr    attr,
                     GRBQConstr[][][]  qconstrs )
```

Query a String-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:
The current values of the requested attribute for each input quadratic constraint.

```
String[]  get (  GRB.StringAttr  attr,
                 GRBGenConstr[]  genconstrs )
```
Query a String-valued general constraint attribute for an array of general constraints.
Arguments:
attr: The attribute being queried.
genconstrs: The general constraints whose attribute values are being queried.
Return value:
The current values of the requested attribute for each input general constraint.

```
String[]  get (  GRB.StringAttr  attr,
                 GRBGenConstr[]  genconstrs,
                 int             start,
                 int             len )
```
Query a String-valued general constraint attribute for a sub-array of constraints.
Arguments:
attr: The attribute being queried.
genconstrs: A one-dimensional array of general constraints whose attribute values are being queried.
start: The index of the first general constraint of interest in the list.
len: The number of general constraints.
Return value:
The current values of the requested attribute for each input general constraint.

```
String[][]  get (  GRB.StringAttr    attr,
                   GRBGenConstr[][]  genconstrs )
```
Query a String-valued constraint attribute for a two-dimensional array of general constraints.
Arguments:
attr: The attribute being queried.
genconstrs: A two-dimensional array of general constraints whose attribute values are being queried.
Return value:
The current values of the requested attribute for each input general constraint.

```
String[][][]  get (  GRB.StringAttr      attr,
                     GRBGenConstr[][][]  genconstrs )
```
Query a String-valued constraint attribute for a three-dimensional array of general constraints.
Arguments:
attr: The attribute being queried.
genconstrs: A three-dimensional array of general constraints whose attribute values are being queried.
Return value:
The current values of the requested attribute for each input general constraint.

GRBModel.getCoeff()

Query the coefficient of variable `var` in linear constraint `constr` (note that the result can be zero).

```
double  getCoeff (  GRBConstr  constr,
                    GRBVar     var )
```

Arguments:

`constr`: The requested constraint.

`var`: The requested variable.

Return value:

The current value of the requested coefficient.

GRBModel.getCol()

Retrieve the list of constraints in which a variable participates, and the associated coefficients. The result is returned as a GRBColumn object.

```
GRBColumn  getCol (  GRBVar  var )
```

Arguments:

`var`: The variable of interest.

Return value:

A GRBColumn object that captures the set of constraints in which the variable participates.

GRBModel.getConcurrentEnv()

Create/retrieve a concurrent environment for a model.

This method provides fine-grained control over the concurrent optimizer. By creating your own concurrent environments and setting appropriate parameters on these environments (e.g., the Method parameter), you can control exactly which strategies the concurrent optimizer employs. For example, if you create two concurrent environments, and set `Method` to primal simplex for one and dual simplex for the other, subsequent concurrent optimizer runs will use the two simplex algorithms rather than the default choices.

Note that you must create contiguously numbered concurrent environments, starting with `num=0`. For example, if you want three concurrent environments, they must be numbered 0, 1, and 2.

Once you create concurrent environments, they will be used for every subsequent concurrent optimization on that model. Use discardConcurrentEnvs to revert back to default concurrent optimizer behavior.

```
GRBEnv  getConcurrentEnv (  int  num )
```

Arguments:

`num`: The concurrent environment number.

Return value:

The concurrent environment for the model.

GRBModel.getConstrByName()

Retrieve a linear constraint from its name. If multiple linear constraints have the same name, this method chooses one arbitrarily. Returns null if no constraint has that name.

```
GRBConstr  getConstrByName (  String  name )
```

Arguments:

name: The name of the desired linear constraint.

Return value:

The requested linear constraint.

GRBModel.getConstrs()

Retrieve an array of all linear constraints in the model.

```
GRBConstr[]  getConstrs ( )
```

Return value:

All linear constraints in the model.

GRBModel.getGenConstrXxx()

The following methods allow you to retrieve general constraints from your model.

GRBModel.getGenConstrMax

Retrieve the data associated with a general constraint of type MAX. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a **null** value for the **vars** argument. The routine returns the total number of operand variables in the specified general constraint in **len**. That allows you to make certain that the **vars** array is of sufficient size to hold the result of the second call.

See also addGenConstrMax for a description of the semantics of this general constraint type.

```
void  getGenConstrMax (  GRBGenConstr  genc,
                         GRBVar[]      resvar,
                         GRBVar[]      vars,
                         int[]         len,
                         double[]      constant )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be **null**.

resvar: Store the resultant variable of the constraint at **resvar[0]**.

vars: Array to store the operand variables of the constraint.

len: Store the number of operand variables of the constraint at `len[0]`.
constant: Store the additional constant operand of the constraint at `constant[0]`.

GRBModel.getGenConstrMin

Retrieve the data associated with a general constraint of type MIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `null` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `len`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also addGenConstrMin for a description of the semantics of this general constraint type.

```
void  getGenConstrMin (  GRBGenConstr  genc,
                         GRBVar[]      resvar,
                         GRBVar[]      vars,
                         int[]         len,
                         double[]      constant )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

resvar: Store the resultant variable of the constraint at `resvar[0]`.
vars: Array to store the operand variables of the constraint.
len: Store the number of operand variables of the constraint at `len[0]`.
constant: Store the additional constant operand of the constraint at `constant[0]`.

GRBModel.getGenConstrAbs

Retrieve the data associated with a general constraint of type ABS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrAbs for a description of the semantics of this general constraint type.

```
void  getGenConstrAbs (  GRBGenConstr  genc,
                         GRBVar[]      resvar,
                         GRBVar[]      argvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

resvar: Store the resultant variable of the constraint at `resvar[0]`.
argvar: Store the argument variable of the constraint at `resvar[0]`.

GRBModel.getGenConstrAnd

Retrieve the data associated with a general constraint of type AND. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `null` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `len`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also addGenConstrAnd for a description of the semantics of this general constraint type.

```
void  getGenConstrAnd (  GRBGenConstr  genc,
                         GRBVar[]      resvar,
                         GRBVar[]      vars,
                         int[]         len )
```

Arguments:

`genc`: The general constraint object.

Any of the following arguments can be `null`.

`resvar`: Store the resultant variable of the constraint at `resvar[0]`.

`vars`: Array to store the operand variables of the constraint.

`len`: Store the number of operand variables of the constraint at `len[0]`.

GRBModel.getGenConstrOr

Retrieve the data associated with a general constraint of type OR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `null` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `len`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also addGenConstrOr for a description of the semantics of this general constraint type.

```
void  getGenConstrOr (  GRBGenConstr  genc,
                        GRBVar[]      resvar,
                        GRBVar[]      vars,
                        int[]         len )
```

Arguments:

`genc`: The general constraint object.

Any of the following arguments can be `null`.

`resvar`: Store the resultant variable of the constraint at `resvar[0]`.

`vars`: Array to store the operand variables of the constraint.

`len`: Store the number of operand variables of the constraint at `len[0]`.

GRBModel.getGenConstrNorm

Retrieve the data associated with a general constraint of type NORM. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `null` value for the `vars` argument. The routine returns the total number of operand variables in the specified general constraint in `len`. That allows you to make certain that the `vars` array is of sufficient size to hold the result of the second call.

See also addGenConstrNorm for a description of the semantics of this general constraint type.

```
void  getGenConstrNorm (  GRBGenConstr  genc,
                          GRBVar[]      resvar,
                          GRBVar[]      vars,
                          int[]         len,
                          double[]      which )
```

Arguments:

`genc`: The general constraint object.

Any of the following arguments can be `null`.

`resvar`: Store the resultant variable of the constraint at `resvar[0]`.

`vars`: Array to store the operand variables of the constraint.

`len`: Store the number of operand variables of the constraint at `len[0]`.

`which`: Store the norm type (possible values are 0, 1, 2, or GRB.INFINITY).

GRBModel.getGenConstrIndicator

Retrieve the data associated with a general constraint of type INDICATOR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrIndicator for a description of the semantics of this general constraint type.

```
void  getGenConstrIndicator (  GRBGenConstr  genc,
                               GRBVar[]      binvar,
                               int[]         binval,
                               GRBLinExpr[]  expr,
                               char[]        sense,
                               double[]      rhs )
```

Arguments:

`genc`: The general constraint object.

Any of the following arguments can be `null`.

`binvar`: Store the binary indicator variable of the constraint at `binvar[0]`.

`binval`: Store the value that the indicator variable has to take in order to trigger the linear constraint at `binval[0]`.

`expr`: Create a GRBLinExpr object to store the left-hand side expression of the linear constraint that is triggered by the indicator at `expr[0]`.

sense: Store the sense for the linear constraint at `sense[0]`. Options are `GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`.
rhs: Store the right-hand side value for the linear constraint at `rhs[0]`.

GRBModel.getGenConstrPWL

Retrieve the data associated with a general constraint of type PWL. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `null` value for the `xpts` and `ypts` arguments. The routine returns the length for the `xpts` and `ypts` arrays in `npts`. That allows you to make certain that the `xpts` and `ypts` arrays are of sufficient size to hold the result of the second call.

See also addGenConstrPWL for a description of the semantics of this general constraint type.

```
void  getGenConstrPWL (  GRBGenConstr  genc,
                         GRBVar[]      xvar,
                         GRBVar[]      yvar,
                         int[]         npts,
                         double[]      xpts,
                         double[]      ypts )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.
yvar: Store the y variable.
npts: Store the number of points that define the piecewise-linear function.
xpts: The x values for the points that define the piecewise-linear function.
ypts: The y values for the points that define the piecewise-linear function.

GRBModel.getGenConstrPoly

Retrieve the data associated with a general constraint of type POLY. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `null` value for the `p` argument. The routine returns the length of the `p` array in `plen`. That allows you to make certain that the `p` array is of sufficient size to hold the result of the second call.

See also addGenConstrPoly for a description of the semantics of this general constraint type.

```
void  getGenConstrPoly (  GRBGenConstr  genc,
                          GRBVar[]      xvar,
                          GRBVar[]      yvar,
                          int[]         plen,
                          double[]      p )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be **null**.

xvar: Store the x variable.

yvar: Store the y variable.

plen: Store the array length for p. If x^d is the highest power term, then $d+1$ will be returned.

p: The coefficients for polynomial function.

GRBModel.getGenConstrExp

Retrieve the data associated with a general constraint of type EXP. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrExp for a description of the semantics of this general constraint type.

```
void  getGenConstrExp (  GRBGenConstr  genc,
                         GRBVar[]      xvar,
                         GRBVar[]      yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be **null**.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.getGenConstrExpA

Retrieve the data associated with a general constraint of type EXPA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrExpA for a description of the semantics of this general constraint type.

```
void  getGenConstrExpA (  GRBGenConstr  genc,
                          GRBVar[]      xvar,
                          GRBVar[]      yvar,
                          double[]      a )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be **null**.

xvar: Store the x variable.

yvar: Store the y variable.

a: Store the base of the function.

GRBModel.getGenConstrLog

Retrieve the data associated with a general constraint of type LOG. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLog for a description of the semantics of this general constraint type.

```
void  getGenConstrLog (  GRBGenConstr  genc,
                         GRBVar[]      xvar,
                         GRBVar[]      yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.getGenConstrLogA

Retrieve the data associated with a general constraint of type LOGA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLogA for a description of the semantics of this general constraint type.

```
void  getGenConstrLogA (  GRBGenConstr  genc,
                          GRBVar[]      xvar,
                          GRBVar[]      yvar,
                          double[]      a )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

a: Store the base of the function.

GRBModel.getGenConstrLogistic

Retrieve the data associated with a general constraint of type LOGISTIC. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLogistic for a description of the semantics of this general constraint type.

```
void  getGenConstrLogistic (  GRBGenConstr  genc,
                              GRBVar[]      xvar,
                              GRBVar[]      yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.getGenConstrPow

Retrieve the data associated with a general constraint of type POW. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrPow for a description of the semantics of this general constraint type.

```
void  getGenConstrPow (  GRBGenConstr  genc,
                         GRBVar[]      xvar,
                         GRBVar[]      yvar,
                         double[]      a )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

a: Store the exponent of the function.

GRBModel.getGenConstrSin

Retrieve the data associated with a general constraint of type SIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrSin for a description of the semantics of this general constraint type.

```
void  getGenConstrSin (  GRBGenConstr  genc,
                         GRBVar[]      xvar,
                         GRBVar[]      yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.getGenConstrCos

Retrieve the data associated with a general constraint of type COS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrCos for a description of the semantics of this general constraint type.

```
void  getGenConstrCos (  GRBGenConstr  genc,
                         GRBVar[]      xvar,
                         GRBVar[]      yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.getGenConstrTan

Retrieve the data associated with a general constraint of type TAN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrTan for a description of the semantics of this general constraint type.

```
void  getGenConstrTan (  GRBGenConstr  genc,
                         GRBVar[]      xvar,
                         GRBVar[]      yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.getGenConstrs()

Retrieve an array of all general constraints in the model.

```
GRBGenConstr[]  getGenConstrs ( )
```

Return value:

All general constraints in the model.

GRBModel.getJSONSolution()

After a call to optimize, this method returns the resulting solution and related model attributes as a JSON string. Please refer to the JSON solution format section for details.

```
String  getJSONSolution ( )
```

Return value:

A JSON string.

GRBModel.getMultiobjEnv()

Create/retrieve a multi-objective environment for the optimization pass with the given index. This environment enables fine-grained control over the multi-objective optimization process. Specifically, by changing parameters on this environment, you modify the behavior of the optimization that occurs during the corresponding pass of the multi-objective optimization.

Each multi-objective environment starts with a copy of the current model environment.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Please refer to the discussion on Combining Blended and Hierarchical Objectives for information on the optimization passes to solve multi-objective models.

Use discardMultiobjEnvs to discard multi-objective environments and return to standard behavior.

```
GRBEnv  getMultiobjEnv ( int index )
```

Arguments:

`index`: The optimization pass index, starting from 0.

Return value:

The multi-objective environment for that optimization pass when solving the model.

GRBModel.getObjective()

Retrieve the model objective(s).

```
GRBExpr  getObjective ( )
```

Retrieve the optimization objective.

Note that the constant and linear portions of the objective can also be retrieved using the ObjCon and Obj attributes.

Return value:

The model objective.

```
GRBLinExpr  getObjective ( int index )
```

Retrieve an alternative optimization objective. Alternative objectives will always be linear. You can also use this routine to retrieve the primary objective (using `index` = 0), but you will get an exception if the primary objective contains quadratic terms.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

Note that alternative objectives can also be retrieved using the ObjNCon and ObjN attributes.

Arguments:

`index`: The index for the requested alternative objective.

Return value:

The requested alternative objective.

GRBModel.getPWLObj()

Retrieve the piecewise-linear objective function for a variable. The return value gives the number of points that define the function, and the x and y arguments give the coordinates of the points, respectively. The x and y arguments must be large enough to hold the result. Call this method with `null` values for x and y if you just want the number of points.

Refer to this discussion for additional information on what the values in x and y mean.

```
int  getPWLObj ( GRBVar    var,
                 double[]  x,
                 double[]  y )
```

Arguments:

`var`: The variable whose objective function is being retrieved.

`x`: The x values for the points that define the piecewise-linear function. These will always be in non-decreasing order.

`y`: The y values for the points that define the piecewise-linear function.

Return value:

The number of points that define the piecewise-linear objective function.

GRBModel.getQCRow()

Retrieve the left-hand side expression from a quadratic constraint. The result is returned as a GRBQuadExpr object.

```
GRBQuadExpr  getQCRow ( GRBQConstr  qc )
```

Arguments:

`qc`: The quadratic constraint of interest.

Return value:

A GRBQuadExpr object that captures the left-hand side of the quadratic constraint.

GRBModel.getQConstrs()

Retrieve an array of all quadratic constraints in the model.

```
GRBQConstr[]  getQConstrs ( )
```

Return value:

All quadratic constraints in the model.

GRBModel.getRow()

Retrieve a list of variables that participate in a constraint, and the associated coefficients. The result is returned as a GRBLinExpr object.

```
GRBLinExpr  getRow ( GRBConstr  constr )
```

Arguments:

constr: The constraint of interest. A GRBConstr object, typically obtained from addConstr or getConstrs.

Return value:

A GRBLinExpr object that captures the set of variables that participate in the constraint.

GRBModel.getSOS()

Retrieve the list of variables that participate in an SOS constraint, and the associated coefficients. The return value is the length of this list. Note that the argument arrays must be long enough to accommodate the result. Call the method with `null` array arguments to determine the appropriate array lengths.

```
int  getSOS ( GRBSOS    sos,
              GRBVar[]  vars,
              double[]  weights,
              int[]     type )
```

Arguments:

sos: The SOS set of interest.

vars: A list of variables that participate in `sos`. Can be `null`.

weights: The SOS weights for each participating variable. Can be `null`.

type: The type of the SOS set (either `GRB.SOS_TYPE1` or `GRB.SOS_TYPE2`) is returned in `type[0]`.

Return value:

The number of entries placed in the output arrays. Note that you should consult the return value to determine the length of the result; the arrays sizes won't necessarily match the result size.

GRBModel.getSOSs()

Retrieve an array of all SOS constraints in the model.

```
GRBSOS[]  getSOSs ( )
```

Return value:

All SOS constraints in the model.

GRBModel.getTuneResult()

Use this method to retrieve the results of a previous tune call. Calling this method with argument `n` causes tuned parameter set `n` to be copied into the model. Parameter sets are stored in order of decreasing quality, with parameter set 0 being the best. The number of available sets is stored in attribute TuneResultCount.

Once you have retrieved a tuning result, you can call optimize to use these parameter settings to optimize the model, or write to write the changed parameters to a `.prm` file.

Please refer to the Parameter Tuning section in the Reference Manual for details on the tuning tool.

```
void  getTuneResult ( int i )
```

`i`: The index of the tuning result to retrieve. The best result is available as index 0. The number of stored results is available in attribute TuneResultCount.

GRBModel.getVarByName()

Retrieve a variable from its name. If multiple variables have the same name, this method chooses one arbitrarily. Returns null if no variable has that name.

```
GRBVar  getVarByName ( String name )
```

Arguments:

`name`: The name of the desired variable.

Return value:

The requested variable.

GRBModel.getVars()

Retrieve an array of all variables in the model.

```
GRBVar[]  getVars ( )
```

Return value:

All variables in the model.

GRBModel.optimize()

Optimize the model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion, this method will populate the solution related attributes of the model. See the Attributes section in the Reference Manual for more information on attributes.

Please consult this section in the Reference Manual for a discussion of some of the practical issues associated with solving a precisely defined mathematical model using finite-precision floating-point arithmetic.

Note that this method will process all pending model modifications.

```
void  optimize ( )
```

GRBModel.optimizeasync()

Optimize a model asynchronously. This routine returns immediately. Your program can perform other computations while optimization proceeds in the background. To check the state of the asynchronous optimization, query the Status attribute for the model. A value of IN_PROGRESS indicates that the optimization has not yet completed. When you are done with your foreground tasks, you must call sync to sync your foreground program with the asynchronous optimization task.

Note that the set of Gurobi calls that you are allowed to make while optimization is running in the background is severely limited. Specifically, you can only perform attribute queries, and only for a few attributes (listed below). Any other calls on the running model, *or on any other models that were built within the same Gurobi environment*, will fail with error code OPTIMIZATION_IN_PROGRESS.

Note that there are no such restrictions on models built in other environments. Thus, for example, you could create multiple environments, and then have a single foreground program launch multiple simultaneous asynchronous optimizations, each in its own environment.

As already noted, you are allowed to query the value of the Status attribute while an asynchronous optimization is in progress. The other attributes that can be queried are: ObjVal, ObjBound, IterCount, NodeCount, and BarIterCount. In each case, the returned value reflects progress in the optimization to that point. Any attempt to query the value of an attribute not on this list will return an OPTIMIZATION_IN_PROGRESS error.

```
void  optimizeasync ( )
```

GRBModel.optimizeBatch()

Submit a new batch request to the Cluster Manager. Returns the BatchID (a string), which uniquely identifies the job in the Cluster Manager and can be used to query the status of this request (from this program or from any other). Once the request has completed, the BatchID can also be used to retrieve the associated solution. To submit a batch request, you must tag at least one element of the model by setting one of the VTag, CTag or QCTag attributes. For more details on batch optimization, please refer to the Batch Optimization section.

Note that this routine will process all pending model modifications.

```
String  optimizeBatch ( )
```

Return value:

A unique string identifier for the batch request.

GRBModel.presolve()

Perform presolve on a model.

Please note that the presolved model computed by this function may be different from the presolved model computed when optimizing the model.

```
GRBModel  presolve ( )
```

Return value:

Presolved version of original model.

GRBModel.read()

This method is the general entry point for importing data from a file into a model. It can be used to read basis files for continuous models, start vectors for MIP models, or parameter settings. The type of data read is determined by the file suffix. File formats are described in the File Format section.

Note that reading a file does **not** process all pending model modifications. These modifications can be processed by calling GRBModel.update.

Note also that this is **not** the method to use if you want to read a new model from a file. For that, use the GRBModel constructor. One variant of the constructor takes the name of the file that contains the new model as its argument.

```
void  read ( String  filename )
```

Arguments:

filename: Name of the file to read. The suffix on the file must be either `.bas` (for an LP basis), `.mst` or `.sol` (for a MIP start), `.hnt` (for MIP hints), `.ord` (for a priority order), or `.prm` (for a parameter file). The suffix may optionally be followed by `.zip`, `.gz`, `.bz2`, or `.7z`.

GRBModel.remove()

Remove a variable, constraint, or SOS from a model.

```
void  remove ( GRBConstr  constr )
```

Remove a linear constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.update), optimize the model (using GRBModel.optimize), or write the model to disk (using GRBModel.write).

Arguments:

constr: The linear constraint to remove.

```
void  remove ( GRBGenConstr  genconstr )
```

Remove a general constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.update), optimize the model (using GRBModel.optimize), or write the model to disk (using GRBModel.write).

Arguments:

genconstr: The general constraint to remove.

```
void  remove ( GRBQConstr  qconstr )
```

Remove a quadratic constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.update), optimize the model (using GRBModel.optimize), or write the model to disk (using GRBModel.write).

Arguments:

`qconstr`: The quadratic constraint to remove.

```
void  remove ( GRBSOS  sos )
```

Remove an SOS constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.update), optimize the model (using GRBModel.optimize), or write the model to disk (using GRBModel.write).

Arguments:

`sos`: The SOS constraint to remove.

```
void  remove ( GRBVar  var )
```

Remove a variable from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.update), optimize the model (using GRBModel.optimize), or write the model to disk (using GRBModel.write).

Arguments:

`var`: The variable to remove.

GRBModel.reset()

Reset the model to an unsolved state, discarding any previously computed solution information.

```
void  reset ( )
```

```
void  reset ( int  clearall )
```

Arguments:

`clearall`: A value of 1 discards additional information that affects the solution process but not the actual model (currently MIP starts, variable hints, branching priorities, lazy flags, and partition information). Pass 0 to just discard the solution.

GRBModel.setCallback()

Set the callback object for a model. The `callback()` method on this object will be called periodically from the Gurobi solver. You will have the opportunity to obtain more detailed information about the state of the optimization from this callback. See the documentation for GRBCallback for additional information.

Note that a model can only have a single callback method, so this call will replace an existing callback.

```
void  setCallback ( GRBCallback  cb )
```

Arguments:

cb: New callback object. To disable a previously set callback, call this method with a **null** argument.

GRBModel.set()

Set the value(s) of a parameter or attribute. Use this method for parameters, for scalar model attributes, or for arrays of constraint or variable attributes.

```
void  set ( GRB.DoubleParam  param,
            double           newval )
```

Set the value of a double-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv.set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newval: The desired new value for the parameter.

```
void  set ( GRB.IntParam  param,
            int           newval )
```

Set the value of an int-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv.set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newval: The desired new value for the parameter.

```
void  set ( GRB.StringParam  param,
            String           newval )
```

Set the value of a string-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv.set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newval: The desired new value for the parameter.

```
void  set (  String  param,
             String  newval )
```

Set the value of any parameter using strings alone.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv.set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The name of the parameter being modified.

newval: The desired new value for the parameter.

```
void  set (  GRB.CharAttr  attr,
             GRBVar[]      vars,
             char[]        newvals )
```

Set a char-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: The variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.CharAttr  attr,
             GRBVar[]      vars,
             char[]        newvals,
             int           start,
             int           len )
```

Set a char-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being modified.

vars: A one-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

start: The index of the first variable of interest in the list.

len: The number of variables.

```
void  set (  GRB.CharAttr  attr,
             GRBVar[][]    vars,
             char[][]      newvals )
```

Set a char-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A two-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.CharAttr  attr,
             GRBVar[][][]  vars,
             char[][][]    newvals )
```

Set a char-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A three-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.CharAttr  attr,
             GRBConstr[]   constrs,
             char[]        newvals )
```

Set a char-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: The constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.CharAttr  attr,
             GRBConstr[]   constrs,
             char[]        newvals,
             int           start,
             int           len )
```

Set a char-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A one-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

```
void  set (  GRB.CharAttr   attr,
             GRBConstr[][]  constrs,
             char[][]       newvals )
```

Set a char-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A two-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.CharAttr     attr,
             GRBConstr[][][]  constrs,
             char[][][]       newvals )
```

Set a char-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A three-dimensional array of constraints whose attribute values are being modified.
newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.CharAttr  attr,
             GRBQConstr[]  qconstrs,
             char[]        newvals )
```

Set a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.
qconstrs: The quadratic constraints whose attribute values are being modified.
newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.CharAttr  attr,
             GRBQConstr[]  qconstrs,
             char[]        newvals,
             int           start,
             int           len )
```

Set a char-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being modified.
qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being modified.
newvals: The desired new values for the attribute for each input quadratic constraint.
start: The index of the first quadratic constraint of interest in the list.
len: The number of quadratic constraints.

```
void  set (  GRB.CharAttr    attr,
             GRBQConstr[][]  qconstrs,
             char[][]        newvals )
```

Set a char-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.
qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being modified.
newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.CharAttr      attr,
             GRBQConstr[][][]  qconstrs,
             char[][][]        newvals )
```

Set a char-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.DoubleAttr  attr,
             double          newval )
```

Set the value of a double-valued model attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value for the attribute.

```
void  set (  GRB.DoubleAttr  attr,
             GRBVar[]        vars,
             double[]        newvals )
```

Set a double-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: The variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.DoubleAttr  attr,
             GRBVar[]        vars,
             double[]        newvals,
             int             start,
             int             len )
```

Set a double-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being modified.

vars: A one-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

start: The index of the first variable of interest in the list.

len: The number of variables.

```
void  set (  GRB.DoubleAttr  attr,
             GRBVar[][]      vars,
             double[][]      newvals )
```

Set a double-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A two-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.DoubleAttr  attr,
             GRBVar[][][]    vars,
             double[][][]    newvals )
```
Set a double-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A three-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.DoubleAttr  attr,
             GRBConstr[]     constrs,
             double[]        newvals )
```
Set a double-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: The constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.DoubleAttr  attr,
             GRBConstr[]     constrs,
             double[]        newvals,
             int             start,
             int             len )
```
Set a double-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A one-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

start: The first constraint of interest in the list.

len: The number of constraints.

```
void  set (  GRB.DoubleAttr  attr,
             GRBConstr[][]   constrs,
             double[][]      newvals )
```
Set a double-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A two-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.DoubleAttr   attr,
             GRBConstr[][][]  constrs,
             double[][][]     newvals )
```
Set a double-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A three-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.DoubleAttr  attr,
             GRBQConstr[]    qconstrs,
             double[]        newvals )
```

Set a double-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: The quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.DoubleAttr  attr,
             GRBQConstr[]    qconstrs,
             double[]        newvals,
             int             start,
             int             len )
```

Set a double-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

start: The first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

```
void  set (  GRB.DoubleAttr  attr,
             GRBQConstr[][]  qconstrs,
             double[][]      newvals )
```

Set a double-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.DoubleAttr    attr,
             GRBQConstr[][][]  qconstrs,
             double[][][]      newvals )
```

Set a double-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.IntAttr  attr,
             int          newval )
```

Set the value of an int-valued model attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value for the attribute.

```
void  set (  GRB.IntAttr  attr,
             GRBVar[]     vars,
             int[]        newvals )
```

Set an int-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: The variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.IntAttr  attr,
             GRBVar[]     vars,
             int[]        newvals,
             int          start,
             int          len )
```

Set an int-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being modified.

vars: A one-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

start: The index of the first variable of interest in the list.

len: The number of variables.

```
void  set (  GRB.IntAttr  attr,
             GRBVar[][]   vars,
             int[][]      newvals )
```

Set an int-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A two-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set ( GRB.IntAttr   attr,
            GRBVar[][][]  vars,
            int[][][]     newvals )
```
Set an int-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A three-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set ( GRB.IntAttr  attr,
            GRBConstr[]  constrs,
            int[]        newvals )
```
Set an int-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: The constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set ( GRB.IntAttr  attr,
            GRBConstr[]  constrs,
            int[]        newvals,
            int          start,
            int          len )
```
Set an int-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A one-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

```
void  set ( GRB.IntAttr    attr,
            GRBConstr[][]  constrs,
            int[][]        newvals )
```
Set an int-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A two-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set ( GRB.IntAttr      attr,
            GRBConstr[][][]  constrs,
            int[][][]        newvals )
```
Set an int-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A three-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.StringAttr  attr,
             String          newval )
```

Set the value of a String-valued model attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value for the attribute.

```
void  set (  GRB.StringAttr  attr,
             GRBVar[]        vars,
             String[]        newvals )
```

Set a String-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: The variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.StringAttr  attr,
             GRBVar[]        vars,
             String[]        newvals,
             int             start,
             int             len )
```

Set a String-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being modified.

vars: A one-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

start: The index of the first variable of interest in the list.

len: The number of variables.

```
void  set (  GRB.StringAttr  attr,
             GRBVar[][]      vars,
             String[][]      newvals )
```

Set a String-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A two-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.StringAttr  attr,
             GRBVar[][][]    vars,
             String[][][]    newvals )
```
Set a String-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A three-dimensional array of variables whose attribute values are being modified.

newvals: The desired new values for the attribute for each input variable.

```
void  set (  GRB.StringAttr  attr,
             GRBConstr[]     constrs,
             String[]        newvals )
```
Set a String-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: The constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.StringAttr  attr,
             GRBConstr[]     constrs,
             String[]        newvals,
             int             start,
             int             len )
```
Set a String-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A one-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

```
void  set (  GRB.StringAttr  attr,
             GRBConstr[][]   constrs,
             String[][]      newvals )
```
Set a String-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A two-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.StringAttr   attr,
             GRBConstr[][][]  constrs,
             String[][][]     newvals )
```
Set a String-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A three-dimensional array of constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input constraint.

```
void  set (  GRB.StringAttr  attr,
             GRBQConstr[]    qconstrs,
             String[]        newvals )
```

Set a String-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: The quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.StringAttr  attr,
             GRBQConstr[]    qconstrs,
             String[]        newvals,
             int             start,
             int             len )
```

Set a String-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

start: The index of the first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

```
void  set (  GRB.StringAttr  attr,
             GRBQConstr[][]  qconstrs,
             String[][]      newvals )
```

Set a String-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.StringAttr    attr,
             GRBQConstr[][][]  qconstrs,
             String[][][]      newvals )
```

Set a String-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input quadratic constraint.

```
void  set (  GRB.StringAttr  attr,
             GRBGenConstr[]  genconstrs,
             String[]        newvals )
```

Set a String-valued general constraint attribute for an array of general constraints.

Arguments:

attr: The attribute being modified.

genconstrs: The general constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input general constraint.

```
void  set (  GRB.StringAttr  attr,
             GRBGenConstr[]  genconstrs,
             String[]        newvals,
             int             start,
             int             len )
```

Set a String-valued general constraint attribute for a sub-array of general constraints.

Arguments:

attr: The attribute being modified.

genconstrs: A one-dimensional array of general constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input general constraint.

start: The index of the first general constraint of interest in the list.

len: The number of general constraints.

```
void  set (  GRB.StringAttr    attr,
             GRBGenConstr[][]  genconstrs,
             String[][]        newvals )
```

Set a String-valued general constraint attribute for a two-dimensional array of general constraints.

Arguments:

attr: The attribute being modified.

genconstrs: A two-dimensional array of general constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input general constraint.

```
void  set (  GRB.StringAttr      attr,
             GRBGenConstr[][][]  genconstrs,
             String[][][]        newvals )
```

Set a String-valued general constraint attribute for a three-dimensional array of general constraints.

Arguments:

attr: The attribute being modified.

genconstrs: A three-dimensional array of general constraints whose attribute values are being modified.

newvals: The desired new values for the attribute for each input general constraint.

GRBModel.setLogCallback()

Sets a logging callback function to query all output posted by the model object. Can be set after a model was created.

```
void  setLogCallback (  java.util.function.Consumer<String>  logCallback )
```

Arguments:

logCallback: The logging callback function.

GRBModel.setObjective()

Set the model objective equal to a linear or quadratic expression (for multi-objective optimization, see setObjectiveN).

Note that you can also modify the linear portion of a model objective using the `Obj` variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `Obj` attribute can be used to modify individual linear terms.

```
void  setObjective (  GRBExpr  expr,
                      int      sense )
```

Set the model objective, and the objective sense (`GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization).

Arguments:

expr: New model objective.

sense: New optimization sense (`GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization).

```
void  setObjective (  GRBExpr  expr )
```

Set the model objective. The sense of the objective is determined by the value of the ModelSense attribute.

Arguments:

expr: New model objective.

GRBModel.setObjectiveN()

```
void  setObjectiveN (  GRBLinExpr  expr,
                       int         index,
                       int         priority,
                       double      weight,
                       double      abstol,
                       double      reltol,
                       String      name )
```

Set an alternative optimization objective equal to a linear expression.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

Note that you can also modify an alternative objective using the ObjN variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `ObjN` attribute can be used to modify individual terms.

Arguments:

`expr`: New alternative objective.

`index`: Index for new objective. If you use an index of 0, this routine will change the primary optimization objective.

`priority`: Priority for the alternative objective. This initializes the ObjNPriority attribute for this objective.

`weight`: Weight for the alternative objective. This initializes the ObjNWeight attribute for this objective.

`abstol`: Absolute tolerance for the alternative objective. This initializes the ObjNAbsTol attribute for this objective.

`reltol`: Relative tolerance for the alternative objective. This initializes the ObjNRelTol attribute for this objective.

`name`: Name of the alternative objective. This initializes the ObjNName attribute for this objective.

GRBModel.setPWLObj()

Set a piecewise-linear objective function for a variable.

The arguments to this method specify a list of points that define a piecewise-linear objective function for a single variable. Specifically, the x and y arguments give coordinates for the vertices of the function.

For additional details on piecewise-linear objective functions, refer to this discussion.

```
void  setPWLObj (  GRBVar    var,
                   double[]  x,
                   double[]  y )
```

Set the piecewise-linear objective function for a variable.

Arguments:

`var`: The variable whose objective function is being set.

x: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

y: The y values for the points that define the piecewise-linear function.

GRBModel.singleScenarioModel()

Capture a single scenario from a multi-scenario model. Use the ScenarioNumber parameter to indicate which scenario to capture.

The model on which this method is invoked must be a multi-scenario model, and the result will be a single-scenario model.

```
GRBModel  singleScenarioModel ( )
```

Return value:

Model for a single scenario.

GRBModel.sync()

Wait for a previous asynchronous optimization call to complete.

Calling optimizeasync returns control to the calling routine immediately. The caller can perform other computations while optimization proceeds, and can check on the progress of the optimization by querying various model attributes. The `sync` call forces the calling program to wait until the asynchronous optimization call completes. You *must* call `sync` before the corresponding model object is deleted.

The `sync` call throws an exception if the optimization itself ran into any problems. In other words, exceptions thrown by this method are those that `optimize` itself would have thrown, had the original method not been asynchronous.

Note that you need to call `sync` even if you know that the asynchronous optimization has already completed.

```
void  sync ( )
```

GRBModel.terminate()

Generate a request to terminate the current optimization. This method can be called at any time during an optimization (from a callback, from another thread, from an interrupt handler, etc.). Note that, in general, the request won't be acted upon immediately.

When the optimization stops, the Status attribute will be equal to `GRB_INTERRUPTED`.

```
void  terminate ( )
```

GRBModel.tune()

Perform an automated search for parameter settings that improve performance. Upon completion, this method stores the best parameter sets it found. The number of stored parameter sets can be determined by querying the value of the TuneResultCount attribute. The actual settings can be retrieved using getTuneResult.

Please refer to the Parameter Tuning section in the Reference Manual for details on the tuning tool.

```
void  tune ( )
```

GRBModel.update()

Process any pending model modifications.

```
void  update ( )
```

GRBModel.write()

This method is the general entry point for writing optimization data to a file. It can be used to write optimization models, solutions vectors, basis vectors, start vectors, or parameter settings. The type of data written is determined by the file suffix. File formats are described in the File Format section.

Note that writing a model to a file will process all pending model modifications. This is also true when writing other model information such as solutions, bases, etc.

Note also that when you write a Gurobi parameter file (PRM), both integer or double parameters not at their default value will be saved, but no string parameter will be saved into the file.

```
void  write (  String  filename )
```

Arguments:

`filename`: The name of the file to be written. The file type is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, or `.rlp` for writing the model itself, `.dua` or `.dlp` for writing the dualized model (only pure LP), `.ilp` for writing just the IIS associated with an infeasible model (see GRBModel.computeIIS for further information), `.sol` for writing the solution selected by the SolutionNumber parameter, `.mst` for writing a start vector, `.hnt` for writing a hint file, `.bas` for writing an LP basis, `.prm` for writing modified parameter settings, `.attr` for writing model attributes, or `.json` for writing solution information in JSON format. If your system has compression utilities installed (e.g., `7z` or `zip` for Windows, and `gzip`, `bzip2`, or `unzip` for Linux or macOS), then the files can be compressed, so additional suffixes of `.gz`, `.bz2`, or `.7z` are accepted.

5.3 GRBVar

Gurobi variable object. Variables are always associated with a particular model. You create a variable object by adding a variable to a model (using GRBModel.addVar), rather than by using a `GRBVar` constructor.

The methods on variable objects are used to get and set variable attributes. For example, solution information can be queried by calling get(`GRB.DoubleAttr.X`). Note, however, that it is generally more efficient to query attributes for a set of variables at once. This is done using the attribute query method on the `GRBModel` object (GRBModel.get).

GRBVar.get()

Query the value of a variable attribute.

```
char  get (  GRB.CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get (  GRB.DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get (  GRB.IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
String  get (  GRB.StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBVar.index()

```
int  index ( )
```

This method returns the current index, or order, of the variable in the underlying constraint matrix.

Return value:

-2: removed, -1: not in model, otherwise: index of the variable in the model

Note that the index of a variable may change after subsequent model modifications.

GRBVar.sameAs()

```
boolean  sameAs (  GRBVar  otherVar )
```

Check whether two variable objects refer to the same variable.

Arguments:

`otherVar`: The other variable.

Return value:

Boolean result indicates whether the two variable objects refer to the same model variable.

GRBVar.set()

Set the value of a variable attribute.

```
void  set (  GRB.CharAttr  attr,
             char          newval )
```

Set the value of a char-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newval`: The desired new value of the attribute.

```
void  set (  GRB.DoubleAttr  attr,
             double          newval )
```

Set the value of a double-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newval`: The desired new value of the attribute.

```
void  set (  GRB.IntAttr  attr,
             int          newval )
```

Set the value of an int-valued attribute.

Arguments:

`attr`: The attribute being modified.

newval: The desired new value of the attribute.

```
void  set (  GRB.StringAttr  attr,
             String          newval )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value of the attribute.

5.4 GRBConstr

Gurobi constraint object. Constraints are always associated with a particular model. You create a constraint object by adding a constraint to a model (using GRBModel.addConstr), rather than by using a `GRBConstr` constructor.

The methods on constraint objects are used to get and set constraint attributes. For example, constraint right-hand sides can be queried by calling get(`GRB.DoubleAttr.RHS`). Note, however, that it is generally more efficient to query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel.get).

GRBConstr.get()

Query the value of a constraint attribute.

```
char  get ( GRB.CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get ( GRB.DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get ( GRB.IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
String  get ( GRB.StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBConstr.index()

```
int  index ( )
```

This method returns the current index, or order, of the constraint in the underlying constraint matrix.

Note that the index of a constraint may change after subsequent model modifications.

Return value:

-2: removed, -1: not in model, otherwise: index of the constraint in the model

GRBConstr.sameAs()

```
boolean  sameAs (  GRBConstr  otherConstr )
```

Check whether two constraint objects refer to the same constraint.

Arguments:

`otherConstr`: The other constraint.

Return value:

Boolean result indicates whether the two constraint objects refer to the same model constraint.

GRBConstr.set()

Set the value of a constraint attribute.

```
void  set (  GRB.CharAttr  attr,
             char          newval )
```

Set the value of a char-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newval`: The desired new value of the attribute.

```
void  set (  GRB.DoubleAttr  attr,
             double          newval )
```

Set the value of a double-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newval`: The desired new value of the attribute.

```
void  set (  GRB.IntAttr  attr,
             int          newval )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.
newval: The desired new value of the attribute.

```
void  set (  GRB.StringAttr  attr,
             String          newval )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.
newval: The desired new value of the attribute.

5.5 GRBQConstr

Gurobi quadratic constraint object. Quadratic constraints are always associated with a particular model. You create a quadratic constraint object by adding a quadratic constraint to a model (using GRBModel.addQConstr), rather than by using a `GRBQConstr` constructor.

The methods on quadratic constraint objects are used to get and set constraint attributes. For example, quadratic constraint right-hand sides can be queried by calling get(`GRB.DoubleAttr.QCRHS`). Note, however, that it is generally more efficient to query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel.get).

GRBQConstr.get()

Query the value of a quadratic constraint attribute.

```
char  get ( GRB.CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get ( GRB.DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get ( GRB.IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
String  get ( GRB.StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBQConstr.set()

Set the value of a quadratic constraint attribute.

```
void  set (  GRB.CharAttr  attr,
             char          newval )
```
Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value of the attribute.

```
void  set (  GRB.DoubleAttr  attr,
             double          newval )
```
Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value of the attribute.

```
void  set (  GRB.IntAttr  attr,
             int          newval )
```
Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value of the attribute.

```
void  set (  GRB.StringAttr  attr,
             String          newval )
```
Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value of the attribute.

5.6 GRBSOS

Gurobi SOS constraint object. SOS constraints are always associated with a particular model. You create an SOS object by adding an SOS constraint to a model (using GRBModel.addSOS), rather than by using a `GRBSOS` constructor. Similarly, SOS constraints are removed using the GRBModel.remove method.

An SOS constraint can be of type 1 or 2 (`GRB.SOS_TYPE1` or `GRB.SOS_TYPE2`). A type 1 SOS constraint is a set of variables where at most one variable in the set may take a value other than zero. A type 2 SOS constraint is an ordered set of variables where at most two variables in the set may take non-zero values. If two take non-zero values, they must be contiguous in the ordered set.

SOS constraint objects have one attribute, `IISSOS`, which can be queried with the GRBSOS.get method.

GRBSOS.get()

Query the value of an SOS attribute.

```
int  get ( GRB.IntAttr  attr )
```

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBSOS.set()

Set the value of an SOS attribute.

```
void  set ( GRB.IntAttr  attr,
            int          newval )
```

Set the value of an int-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newval`: The desired new value of the attribute.

5.7 GRBGenConstr

Gurobi general constraint object. General constraints are always associated with a particular model. You create a general constraint object by adding a general constraint to a model (using one of the `GRBModel.addGenConstrXxx()` methods), rather than by using a `GRBGenConstr` constructor.

The methods on general constraint objects are used to get and set constraint attributes. For example, general constraint types can be queried by calling get(`GRB.IntAttr.GenConstrType`). Note, however, that it is generally more efficient to query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel.get).

GRBGenConstr.get()

Query the value of a general constraint attribute.

```
double  get (  GRB.DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get (  GRB.IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
String  get (  GRB.StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBGenConstr.set()

Set the value of a general constraint attribute.

```
void  set (  GRB.DoubleAttr  attr,
             double          newval )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value of the attribute.

```
void  set (  GRB.IntAttr  attr,
             int          newval )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value of the attribute.

```
void  set (  GRB.StringAttr  attr,
             String          newval )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newval: The desired new value of the attribute.

5.8 GRBExpr

Abstract base class for the GRBLinExpr and GRBQuadExpr classes. Expressions are used to build objectives and constraints. They are temporary objects that typically have short lifespans.

GRBExpr.getValue()

Compute the value of an expression for the current solution.

```
double  getValue ( )
```

Return value:

Value of the expression for the current solution.

5.9 GRBLinExpr

Gurobi linear expression object. A linear expression consists of a constant term, plus a list of coefficient-variable pairs that capture the linear terms. Linear expressions are used to build constraints. They are temporary objects that typically have short lifespans.

The `GRBLinExpr` class is a sub-class of the abstract base class GRBExpr.

You generally build linear expressions by starting with an empty expression (using the GRBLinExpr constructor), and then adding terms. Terms can be added individually, using addTerm, or in groups, using addTerms, or multAdd. Terms can also be removed from an expression, using remove.

To add a linear constraint to your model, you generally build one or two linear expression objects (`expr1` and `expr2`) and then pass them to GRBModel.addConstr. To give a few examples:

$$\text{model.addConstr}(expr1, GRB.LESS\_EQUAL, expr2)$$
$$\text{model.addConstr}(expr1, GRB.EQUAL, 1)$$

Once you add a constraint to your model, subsequent changes to the expression object you used to build the constraint will not change the constraint (you would use GRBModel.chgCoeff for that).

Individual terms in a linear expression can be queried using the getVar, getCoeff, and getConstant methods. You can query the number of terms in the expression using the size method.

Note that a linear expression may contain multiple terms that involve the same variable. These duplicate terms are merged when creating a constraint from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar).

GRBLinExpr()

Linear expression constructor. Create an empty linear expression, or copy an existing expression.

```
GRBLinExpr  GRBLinExpr ( )
```

Create an empty linear expression.

Return value:

An empty expression object.

```
GRBLinExpr  GRBLinExpr (  GRBLinExpr  le )
```

Copy an existing linear expression.

Arguments:

`le`: Existing expression to copy.

Return value:

A copy of the input expression object.

GRBLinExpr.add()

Add one linear expression into another. Upon completion, the invoking linear expression will be equal to the sum of itself and the argument expression.

```
void  add ( GRBLinExpr  le )
```

Arguments:

`le`: Linear expression to add.

GRBLinExpr.addConstant()

Add a constant into a linear expression.

```
void  addConstant ( double  c )
```

Arguments:

`c`: Constant to add to expression.

GRBLinExpr.addTerm()

Add a single term into a linear expression.

```
void  addTerm ( double  coeff,
                GRBVar  var )
```

Arguments:

`coeff`: Coefficient for new term.

`var`: Variable for new term.

GRBLinExpr.addTerms()

Add new terms into a linear expression.

```
void  addTerms ( double[]  coeffs,
                 GRBVar[]  vars )
```

Add a list of terms into a linear expression. Note that the lengths of the two argument arrays must be equal.

Arguments:

`coeffs`: Coefficients for new terms.

`vars`: Variables for new terms.

```
void  addTerms ( double[]  coeffs,
                 GRBVar[]  vars,
                 int       start,
                 int       len )
```

Add new terms into a linear expression. This signature allows you to use arrays to hold the coefficients and variables that describe the terms in an array without being forced to add a term for each entry in the array. The `start` and `len` arguments allow you to specify which terms to add.

Arguments:

coeffs: Coefficients for new terms.
vars: Variables for new terms.
start: The first term in the list to add.
len: The number of terms to add.

GRBLinExpr.clear()

Set a linear expression to 0.

```
void  clear ( )
```

GRBLinExpr.getConstant()

Retrieve the constant term from a linear expression.

```
double  getConstant ( )
```

Return value:
Constant from expression.

GRBLinExpr.getCoeff()

Retrieve the coefficient from a single term of the expression.

```
double  getCoeff (  int  i )
```

Arguments:
i: Index for coefficient of interest.
Return value:
Coefficient for the term at index **i** in the expression.

GRBLinExpr.getValue()

Compute the value of a linear expression for the current solution.

```
double  getValue ( )
```

Return value:
Value of the expression for the current solution.

GRBLinExpr.getVar()

Retrieve the variable object from a single term of the expression.

```
GRBVar  getVar (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

Variable for the term at index `i` in the expression.

GRBLinExpr.multAdd()

Add a constant multiple of one linear expression into another. Upon completion, the invoking linear expression is equal the sum of itself and the constant times the argument expression.

```
void  multAdd (  double      m,
                 GRBLinExpr  le )
```

Arguments:

`m`: Constant multiplier for added expression.

`le`: Linear expression to add.

GRBLinExpr.remove()

Remove a term from a linear expression.

```
void  remove (  int  i )
```

Remove the term stored at index `i` of the expression.

Arguments:

`i`: The index of the term to be removed.

```
boolean  remove (  GRBVar  var )
```

Remove all terms associated with variable `var` from the expression.

Arguments:

`var`: The variable whose term should be removed.

Return value:

Returns `true` if the variable appeared in the linear expression (and was removed).

GRBLinExpr.size()

Retrieve the number of terms in the linear expression (not including the constant).

```
int  size ( )
```

Return value:

Number of terms in the expression.

5.10 GRBQuadExpr

Gurobi quadratic expression object. A quadratic expression consists of a linear expression, plus a list of coefficient-variable-variable triples that capture the quadratic terms. Quadratic expressions are used to build quadratic objective functions and quadratic constraints. They are temporary objects that typically have short lifespans.

The `GRBQuadExpr` class is a sub-class of the abstract base class GRBExpr.

You generally build quadratic expressions by starting with an empty expression (using the GRBQuadExpr constructor), and then adding terms. Terms can be added individually, using addTerm, or in groups, using addTerms, or multAdd. Quadratic terms can be removed from a quadratic expression using remove.

To add a quadratic constraint to your model, you generally build one or two quadratic expression objects (`qexpr1` and `qexpr2`) and then use an overloaded comparison operator to build an argument for GRBModel.addQConstr. To give a few examples:

$$\text{model.addQConstr}(qexpr1, GRB.LESS\_EQUAL, qexpr2)$$
$$\text{model.addQConstr}(qexpr1, GRB.EQUAL, 1)$$

Once you add a constraint to your model, subsequent changes to the expression object you used to build the constraint will have no effect on that constraint.

Individual quadratic terms in a quadratic expression can be queried using the getVar1, getVar2, and getCoeff methods. You can query the number of quadratic terms in the expression using the size method. To query the constant and linear terms associated with a quadratic expression, first obtain the linear portion of the quadratic expression using getLinExpr, and then use the getConstant, getCoeff, and getVar methods on the resulting GRBLinExpr object.

Note that a quadratic expression may contain multiple terms that involve the same variable pair. These duplicate terms are merged when creating the model objective from an expression, but they may be visible when inspecting individual quadratic terms in the expression (e.g., when using getVar1 and getVar2).

GRBQuadExpr()

Quadratic expression constructor. Create an empty quadratic expression, or copy an existing expression.

`GRBQuadExpr` **`GRBQuadExpr`** `( )`

Create an empty quadratic expression.

Return value:

An empty expression object.

`GRBQuadExpr` **`GRBQuadExpr`** `( GRBLinExpr le )`

Initialize a quadratic expression from an existing linear expression.

Arguments:

`le`: Existing linear expression to copy.

Return value:
Quadratic expression object whose initial value is taken from the input linear expression.

```
GRBQuadExpr  GRBQuadExpr ( GRBQuadExpr  qe )
```

Copy an existing quadratic expression.
Arguments:
`qe`: Existing expression to copy.
Return value:
A copy of the input expression object.

GRBQuadExpr.add()

Add an expression into a quadratic expression. Upon completion, the invoking quadratic expression will be equal to the sum of itself and the argument expression.

```
void  add ( GRBLinExpr  le )
```

Add a linear expression.
Arguments:
`le`: Linear expression to add.

```
void  add ( GRBQuadExpr  qe )
```

Add a quadratic expression.
Arguments:
`qe`: Quadratic expression to add.

GRBQuadExpr.addConstant()

Add a constant into a quadratic expression.

```
void  addConstant ( double  c )
```

Arguments:
`c`: Constant to add to expression.

GRBQuadExpr.addTerm()

Add a single term into a quadratic expression.

```
void  addTerm ( double  coeff,
                GRBVar  var )
```

Add a single linear term (`coeff*var`) into a quadratic expression.

Arguments:

`coeff`: Coefficient for new term.

`var`: Variable for new term.

```
void  addTerm ( double  coeff,
                GRBVar  var1,
                GRBVar  var2 )
```

Add a single quadratic term (`coeff*var1*var2`) into a quadratic expression.

Arguments:

`coeff`: Coefficient for new quadratic term.

`var1`: First variable for new quadratic term.

`var2`: Second variable for new quadratic term.

GRBQuadExpr.addTerms()

Add new terms into a quadratic expression.

```
void  addTerms ( double[]  coeffs,
                 GRBVar[]  vars )
```

Add a list of linear terms into a quadratic expression. Note that the lengths of the two argument arrays must be equal.

Arguments:

`coeffs`: Coefficients for new terms.

`vars`: Variables for new terms.

```
void  addTerms ( double[]  coeffs,
                 GRBVar[]  vars,
                 int       start,
                 int       len )
```

Add new linear terms into a quadratic expression. This signature allows you to use arrays to hold the coefficients and variables that describe the linear terms in an array without being forced to add a term for each entry in the array. The `start` and `len` arguments allow you to specify which terms to add.

Arguments:

`coeffs`: Coefficients for new terms.

`vars`: Variables for new terms.

`start`: The first term in the list to add.

`len`: The number of terms to add.

```
void  addTerms ( double[]  coeffs,
                 GRBVar[]  vars1,
                 GRBVar[]  vars2 )
```

Add a list of quadratic terms into a quadratic expression. Note that the lengths of the three argument arrays must be equal.

Arguments:

`coeffs`: Coefficients for new quadratic terms.

`vars1`: First variables for new quadratic terms.

`vars2`: Second variables for new quadratic terms.

```
void  addTerms (  double[]  coeffs,
                  GRBVar[]  vars1,
                  GRBVar[]  vars2,
                  int       start,
                  int       len )
```

Add new quadratic terms into a quadratic expression. This signature allows you to use arrays to hold the coefficients and variables that describe the terms in an array without being forced to add a term for each entry in the array. The `start` and `len` arguments allow you to specify which terms to add.

Arguments:

`coeffs`: Coefficients for new quadratic terms.

`vars1`: First variables for new quadratic terms.

`vars2`: Second variables for new quadratic terms.

`start`: The first term in the list to add.

`len`: The number of terms to add.

GRBQuadExpr.clear()

Set a quadratic expression to 0.

```
void  clear ( )
```

GRBQuadExpr.getCoeff()

Retrieve the coefficient from a single quadratic term of the quadratic expression.

```
double  getCoeff (  int  i )
```

Arguments:

`i`: Index for coefficient of interest.

Return value:

Coefficient for the quadratic term at index `i` in the expression.

GRBQuadExpr.getLinExpr()

A quadratic expression is represented as a linear expression, plus a list of quadratic terms. This method retrieves the linear expression associated with the quadratic expression.

```
GRBLinExpr  getLinExpr ( )
```

Return value:

Linear expression associated with the quadratic expression.

GRBQuadExpr.getValue()

Compute the value of a quadratic expression for the current solution.

```
double  getValue ( )
```

Return value:

Value of the expression for the current solution.

GRBQuadExpr.getVar1()

Retrieve the first variable object associated with a single quadratic term from the expression.

```
GRBVar  getVar1 (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

First variable for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr.getVar2()

Retrieve the second variable object associated with a single quadratic term from the expression.

```
GRBVar  getVar2 (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

Second variable for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr.multAdd()

Add a constant multiple of one quadratic expression into another. Upon completion, the invoking quadratic expression is equal the sum of itself and the constant times the argument expression.

```
void  multAdd (  double      m,
                 GRBLinExpr  le )
```

Add a linear expression into a quadratic expression.

Arguments:

`m`: Constant multiplier for added expression.

`le`: Linear expression to add.

```
void  multAdd (  double       m,
                 GRBQuadExpr  qe )
```

Add a quadratic expression into a quadratic expression.

Arguments:

`m`: Constant multiplier for added expression.

`qe`: Quadratic expression to add.

GRBQuadExpr.remove()

Remove a term from a quadratic expression.

```
void  remove (  int  i )
```

Remove the quadratic term stored at index `i` of the expression.

Arguments:

`i`: The index of the quadratic term to be removed.

```
boolean  remove (  GRBVar  var )
```

Remove all quadratic terms associated with variable `var` from the expression.

Arguments:

`var`: The variable whose quadratic term should be removed.

Return value:

Returns `true` if the variable appeared in the quadratic expression (and was removed).

GRBQuadExpr.size()

Retrieve the number of quadratic terms in the quadratic expression. Use GRBQuadExpr.getLinExpr to retrieve constant or linear terms from the quadratic expression.

```
int  size ( )
```

Return value:

Number of quadratic terms in the expression.

5.11 GRBColumn

Gurobi column object. A column consists of a list of coefficient, constraint pairs. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients. They are temporary objects that typically have short lifespans.

You generally build columns by starting with an empty column (using the GRBColumn constructor), and then adding terms. Terms can be added individually, using addTerm, or in groups, using addTerms. Terms can also be removed from a column, using remove.

Individual terms in a column can be queried using the getConstr, and getCoeff methods. You can query the number of terms in the column using the size method.

GRBColumn()

Column constructor. Create an empty column, or copy an existing column.

```
GRBColumn  GRBColumn ( )
```

Create an empty column.

Return value:

An empty column object.

```
GRBColumn  GRBColumn (  GRBColumn  col )
```

Copy an existing column.

Arguments:

`col`: Existing column object.

Return value:

A copy of the input column object.

GRBColumn.addTerm()

Add a single term into a column.

```
void  addTerm (  double     coeff,
                 GRBConstr  constr )
```

Arguments:

`coeff`: Coefficient for new term.

`constr`: Constraint for new term.

GRBColumn.addTerms()

Add new terms into a column.

```
void  addTerms (  double[]     coeffs,
                  GRBConstr[]  constrs )
```

Add a list of terms into a column. Note that the lengths of the two argument arrays must be equal.

Arguments:

`coeffs`: Coefficients for added constraints.

`constrs`: Constraints to add to column.

```
void  addTerms (  double[]     coeffs,
                  GRBConstr[]  constrs,
                  int          start,
                  int          len )
```

Add new terms into a column. This signature allows you to use arrays to hold the coefficients and constraints that describe the terms in an array without being forced to add an term for each member in the array. The `start` and `len` arguments allow you to specify which terms to add.

Arguments:

`coeffs`: Coefficients for added constraints.

`constrs`: Constraints to add to column.

`start`: The first term in the list to add.

`len`: The number of terms to add.

GRBColumn.clear()

Remove all terms from a column.

```
void  clear ( )
```

GRBColumn.getCoeff()

Retrieve the coefficient from a single term in the column.

```
double  getCoeff (  int  i )
```

Arguments:

`i`: Index for coefficient of interest.

Return value:

Coefficient for the term at index `i` in the column.

GRBColumn.getConstr()

Retrieve the constraint object from a single term in the column.

```
GRBConstr  getConstr (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

Constraint for the term at index `i` in the column.

GRBColumn.remove()

Remove a single term from a column.

```
void  remove (  int  i )
```

Remove the term stored at index `i` of the column.

Arguments:

`i`: The index of the term to be removed.

```
boolean  remove (  GRBConstr  constr )
```

Remove the term associated with constraint `constr` from the column.

Arguments:

`constr`: The constraint whose term should be removed.

Return value:

Returns `true` if the constraint appeared in the column (and was removed).

GRBColumn.size()

Retrieve the number of terms in the column.

```
int  size ( )
```

Return value:

Number of terms in the column.

5.12 GRBCallback

Gurobi callback class. This is an abstract class. To implement a callback, you should create a subclass of this class and implement a `callback()` method. If you pass an object of this subclass to method GRBModel.setCallback before calling GRBModel.optimize or GRBModel.computeIIS, the `callback()` method of the class will be called periodically. Depending on where the callback is called from, you will be able to obtain various information about the progress of the optimization.

Note that this class contains one protected *int* member variable: `where`. You can query this variable from your `callback()` method to determine where the callback was called from.

Gurobi callbacks can be used both to monitor the progress of the optimization and to modify the behavior of the Gurobi Optimizer. A simple user callback function might call the GRBCallback.getIntInfo or GRBCallback.getDoubleInfo methods to produce a custom display, or perhaps to terminate optimization early (using GRBCallback.abort) or to proceed to the next phase of the computation (using GRBCallback.proceed). More sophisticated MIP callbacks might use GRBCallback.getNodeRel or GRBCallback.getSolution to retrieve values from the solution to the current node, and then use GRBCallback.addCut or GRBCallback.addLazy to add a constraint to cut off that solution, or GRBCallback.setSolution to import a heuristic solution built from that solution. For multi-objective problems, you might use GRBCallback.stopOneMultiObj to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

When solving a model using multiple threads, the user callback is only ever called from a single thread, so you don't need to worry about the thread-safety of your callback.

Note that changing parameters from within a callback is not supported, doing so may lead to undefined behavior.

You can look at the `Callback.java` example for details of how to use Gurobi callbacks.

GRBCallback()

Callback constructor.

```
GRBCallback  GRBCallback ( )
```

Return value:

A callback object.

GRBCallback.abort()

Abort optimization. When the optimization stops, the Status attribute will be equal to `GRB.Status.INTERRUPTED`.

```
void  abort ( )
```

GRBCallback.addCut()

Add a cutting plane to the MIP model from within a callback function. Note that this method can only be invoked when the `where` member variable is equal to `GRB.CB_MIPNODE` (see the Callback Codes section in the Reference Manual for more information).

Cutting planes can be added at any node of the branch-and-cut tree. However, they should be added sparingly, since they increase the size of the relaxation model that is solved at each node and can significantly degrade node processing speed.

Cutting planes are typically used to cut off the current relaxation solution. To retrieve the relaxation solution at the current node, you should first call getNodeRel.

You should consider setting parameter PreCrush to value 1 when adding your own cuts. This setting shuts off a few presolve reductions that can sometimes prevent your cut from being applied to the presolved model (which would result in your cut being silently ignored).

Note that cutting planes added through this method must truly be cutting planes -- they can cut off continuous solutions, but they may not cut off integer solutions that respect the original constraints of the model. Ignoring this restriction will lead to incorrect solutions.

```
void  addCut ( GRBLinExpr  lhsExpr,
               char        sense,
               double      rhs )
```

Arguments:

lhsExpr: Left-hand side expression for new cutting plane.

sense: Sense for new cutting plane (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhs: Right-hand side value for new cutting plane.

GRBCallback.addLazy()

Add a lazy constraint to the MIP model from within a callback function. Note that this method can only be invoked when the **where** member variable is equal to GRB.CB_MIPNODE or GRB.CB_MIPSOL (see the Callback Codes section in the Reference Manual for more information).

Lazy constraints are typically used when the full set of constraints for a MIP model is too large to represent explicitly. By only including the constraints that are actually violated by solutions found during the branch-and-cut search, it is sometimes possible to find a proven optimal solution while only adding a fraction of the full set of constraints.

You would typically add a lazy constraint by first querying the current node solution (by calling getSolution from a GRB.CB_MIPSOL callback, or getNodeRel from a GRB.CB_MIPNODE callback), and then calling addLazy() to add a constraint that cuts off the solution. Gurobi guarantees that you will have the opportunity to cut off any solutions that would otherwise be considered feasible.

MIP solutions may be generated outside of a MIP node. Thus, generating lazy constraints is optional when the **where** value in the callback function equals GRB.CB_MIPNODE. To avoid this, we recommend to always check when the **where** value equals GRB.CB_MIPSOL.

Your callback should be prepared to cut off solutions that violate any of your lazy constraints, including those that have already been added. Node solutions will usually respect previously added lazy constraints, but not always.

Note that you must set the LazyConstraints parameter if you want to use lazy constraints.

```
void  addLazy ( GRBLinExpr  lhsExpr,
                char        sense,
                double      rhs )
```

Arguments:

lhsExpr: Left-hand side expression for new lazy constraint.
sense: Sense for new lazy constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).
rhs: Right-hand side value for new lazy constraint.

GRBCallback.getDoubleInfo()

Request double-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the double-valued information that can be queried for different values of **where**, please refer to the Callback Codes section of the Reference Manual.

```
double  getDoubleInfo (  int  what )
```

Arguments:
what: Information requested. Please refer to the list of Callback Codes in the Reference Manual for possible values.
Return value:
Value of requested callback information.

GRBCallback.getIntInfo()

Request int-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the int-valued information that can be queried for different values of **where**, please refer to the Callback Codes section in the Reference Manual.

```
int  getIntInfo (  int  what )
```

Arguments:
what: Information requested. Please refer to the list of Callback Codes in the Reference Manual for possible values.
Return value:
Value of requested callback information.

GRBCallback.getNodeRel()

Retrieve node relaxation solution values at the current node. Only available when the **where** member variable is equal to GRB.CB_MIPNODE, and GRB.CB_MIPNODE_STATUS is equal to GRB.Status.OPTIMAL.

```
double  getNodeRel (  GRBVar  v )
```

Arguments:

v: The variable whose value is desired.

Return value:

The value of the specified variable in the node relaxation for the current node.

```
double[]  getNodeRel (  GRBVar[]  xvars )
```

Arguments:

xvars: The list of variables whose values are desired.

Return value:

The values of the specified variables in the node relaxation for the current node.

```
double[][]  getNodeRel (  GRBVar[][]  xvars )
```

Arguments:

xvars: The array of variables whose values are desired.

Return value:

The values of the specified variables in the node relaxation for the current node.

GRBCallback.getSolution()

Retrieve values from the current solution vector. Only available when the **where** member variable is equal to GRB.CB_MIPSOL or GRB.CB_MULTIOBJ.

```
double  getSolution (  GRBVar  v )
```

Arguments:

v: The variable whose value is desired.

Return value:

The value of the specified variable in the current solution vector.

```
double[]  getSolution (  GRBVar[]  xvars )
```

Arguments:

xvars: The list of variables whose values are desired.

Return value:

The values of the specified variables in the current solution.

```
double[][]  getSolution (  GRBVar[][]  xvars )
```

Arguments:

xvars: The array of variables whose values are desired.

Return value:

The values of the specified variables in the current solution.

GRBCallback.getStringInfo()

Request string-valued callback information. The available information depends on the value of the `where` member. For information on possible values of `where`, and the string-valued information that can be queried for different values of `where`, please refer to the Callback Codes section of the Reference Manual.

```
String  getStringInfo (  int  what )
```

Arguments:

`what`: Information requested. Please refer to the list of Callback Codes in the Reference Manual for possible values.

Return value:

Value of requested callback information.

GRBCallback.proceed()

Generate a request to proceed to the next phase of the computation. Note that the request is only accepted in a few phases of the algorithm, and it won't be acted upon immediately.

In the current Gurobi version, this callback allows you to proceed from the NoRel heuristic to the standard MIP search. You can determine the current algorithm phase using `MIP_PHASE`, `MIPNODE_PHASE`, or `MIPSOL_PHASE` queries from a callback.

```
void  proceed ( )
```

GRBCallback.setSolution()

Import solution values for a heuristic solution. Only available when the `where` member variable is equal to `GRB.CB_MIP`, `GRB.CB_MIPNODE`, or `GRB.CB_MIPSOL` (see the Callback Codes section for more information).

When you specify a heuristic solution from a callback, variables initially take undefined values. You should use this method to specify variable values. You can make multiple calls to `setSolution` from one callback invocation to specify values for multiple sets of variables. After the callback, if values have been specified for any variables, the Gurobi Optimizer will try to compute a feasible solution from the specified values, possibly filling in values for variables whose values were left undefined. You can also optionally call useSolution within your callback function to try to immediately compute a feasible solution from the specified values.

Note that this method is not supported in a Compute Server environment.

```
void  setSolution (  GRBVar  v,
                     double  val )
```

Arguments:

`v`: The variable whose values is being set.

`val`: The value of the variable in the new solution.

```
void  setSolution ( GRBVar[]  xvars,
                    double[]  sol )
```

Arguments:

xvars: The variables whose values are being set.

sol: The desired values of the specified variables in the new solution.

GRBCallback.stopOneMultiObj()

Interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process. Only available for multi-objective MIP models and when the **where** member variable is not equal to GRB.CB_MULTIOBJ (see the Callback Codes section for more information).

You would typically stop a multi-objective optimization step by querying the last finished number of multi-objectives steps, and using that number to stop the current step and move on to the next hierarchical objective (if any) as shown in the following example:

Example usage:

```
import gurobi.*;

public class Callback extends GRBCallback {
  private int objcnt;
  private long starttime;

  protected void callback() {
    try {
      if (where == GRB.CB_MULTIOBJ) {
        /* get current objective number */
        objcnt    = getIntInfo(GRB.CB_MULTIOBJ_OBJCNT);

        /* reset start time to current time */
        starttime = System.currentTimeMillis();

      } else if (System.currentTimeMillis() - starttime > BIG ||
                 /* takes too long or good enough */) {
        /* stop only this optimization step */
        stopOneMultiObj(objcnt);
      }
    }
  }
}
```

You should refer to the section on Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

```
void  stopOneMultiObj ( int  objcnt )
```

Arguments:

objnum: The number of the multi-objective optimization step to interrupt. For processes running locally, this argument can have the special value -1, meaning to stop the current step.

GRBCallback.useSolution()

Once you have imported solution values using setSolution, you can optionally call `useSolution` in a `GRB.CB_MIPNODE` callback to immediately use these values to try to compute a heuristic solution. Alternatively, you can call `useSolution` in a `GRB.CB_MIP` or `GRB.CB_MIPSOL` callback, which will store the solution until it can be processed internally.

```
double  useSolution ( )
```

Return value:

The objective value for the solution obtained from your solution values. It equals `GRB.INFINITY` if no improved solution is found or the method has been called from a callback other than `GRB.CB_MIPNODE` as, in these contexts, the solution is stored instead of being processed immediately.

5.13 GRBException

Gurobi exception object. This is a sub-class of the Java `Exception` class. A number of useful methods, including `getMessage()` and `printStackTrace()`, are inherited from the parent class. For a list of parent class methods in Java 1.6, visit this site.

GRBException()

Exception constructor.

```
GRBException  GRBException ( int  errcode )
```

Create a Gurobi exception.
Arguments:
`errcode`: Error code for exception.
Return value:
An exception object.

```
GRBException  GRBException ( String  errmsg )
```

Create a Gurobi exception.
Arguments:
`errmsg`: Error message for exception.
Return value:
An exception object.

```
GRBException  GRBException ( String  errmsg,
                             int     errcode )
```

Create a Gurobi exception.
Arguments:
`errmsg`: Error message for exception.
`errcode`: Error code for exception.
Return value:
An exception object.

GRBException.getErrorCode()

Retrieve the error code associated with a Gurobi exception.

```
int  getErrorCode ( )
```

Return value:
The error code associated with the exception.

5.14 GRBBatch

Gurobi batch object. Batch optimization is a feature available with the Gurobi Cluster Manager. It allows a client program to build an optimization model, submit it to a Compute Server cluster (through a Cluster Manager), and later check on the status of the model and retrieve its solution. For more information, please refer to the Batch Optimization section.

Commonly used methods on the batch object include update (refresh attributes from the Cluster Manager), abort (abort execution of a batch request), retry (retry optimization for an interrupted or failed batch request), discard (remove the batch request and all related information from the Cluster Manager), and getJSONSolution (query solution information for the batch request).

These methods are built on top of calls to the Cluster Manager REST API. They are meant to simplify such calls, but note that you always have the option of calling the REST API directly.

Batch objects have four attributes:

: BatchID: Unique ID for the batch request.

: BatchStatus: Current optimization status for the batch request. Status values are described in the Batch Status Code section.

: BatchErrorCode: Last error code.

: BatchErrorMessage: Last error message.

You can access their values by using get. Note that all Batch attributes are locally cached, and are only updated when you create a client-side batch object or when you explicitly update this cache, which can done by calling update.

While the Java garbage collector will eventually collect an unused `GRBBatch` object, the vast majority of the memory associated with a model is stored outside of the Java heap. As a result, the garbage collector can't see this memory usage, and thus it can't take this quantity into account when deciding whether collection is necessary. We recommend that you call GRBBatch.dispose when you are done with the batch.

GRBBatch()

Given a `BatchID`, as returned by optimizeBatch, and a Gurobi environment that can connect to the appropriate Cluster Manager (i.e., one where parameters CSManager, UserName, and ServerPassword have been set appropriately), this function returns a GRBBatch object. With it, you can query the current status of the associated batch request and, once the batch request has been processed, you can query its solution. Please refer to the Batch Optimization section for details and examples.

```
GRBBatch  GRBBatch (  GRBEnv  env,
                      String  batchID )
```

Arguments:

`env`: The environment in which the new batch object should be created.

`batchID`: ID of the batch request for which you want to access status and other information.

Return value:

New batch object.

Example usage:

```
// Create Batch-object
GRBBatch batch = new GRBBatch(env, batchid);
```

GRBBatch.abort()

`void` **abort** ( )

This method instructs the Cluster Manager to abort the processing of this batch request, changing its status to `ABORTED`. Please refer to the Batch Status Codes section for further details.

Example usage:

```
// Request to abort the batch
batch.abort();
```

GRBBatch.discard()

`void` **discard** ( )

This method instructs the Cluster Manager to remove all information related to the batch request in question, including the stored solution if available. Further queries for the associated batch request will fail with error code `GRB_ERROR_DATA_NOT_AVAILABLE`. Use this function with care, as the removed information can not be recovered later on.

Example usage:

```
// Request to erase input and output data related to this batch
batch.discard();
```

GRBBatch.dispose()

`void` **dispose** ( )

Free all resources associated with this Batch object. After this method is called, this Batch object must no longer be used.

Example usage:

```
// Dispose resources
batch.dispose();
```

GRBBatch.getJSONSolution()

`String getJSONSolution ( )` **Return value:**

The requested solution in JSON format.

This function retrieves the solution of a completed batch request from a Cluster Manager. The solution is returned as a JSON solution string. For this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details.

Example usage:

```
// print JSON solution into string
System.out.println("JSON solution:" + batch.getJSONSolution());
```

GRBBatch.get()

Query the value of an attribute.

`int get ( GRB.IntAttr attr )`

Query the value of an int-valued batch attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

`String get ( GRB.StringAttr attr )`

Query the value of a string-valued batch attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBBatch.retry()

`void retry ( )`

This method instructs the Cluster Manager to retry optimization of a failed or aborted batch request, changing its status to `SUBMITTED`. Please refer to the Batch Status Codes section for further details.

Example usage:

```
// Retry the batch job
batch.retry();
```

GRBBatch.update()

```
void  update ( )
```

All Batch attribute values are cached locally, so queries return the value received during the last communication with the Cluster Manager. This method refreshes the values of all attributes with the values currently available in the Cluster Manager (which involves network communication).

Example usage:

```
// Update local attributes
batch.update();
```

GRBBatch.writeJSONSolution()

```
void  writeJSONSolution ( String filename )
```

This method returns the stored solution of a completed batch request from a Cluster Manager. The solution is returned in a gzip-compressed JSON file. The file name you provide must end with a `.json.gz` extension. The JSON format is described in the JSON solution format section. Note that for this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details.

Arguments:

`filename`: Name of file where the solution should be stored (in JSON format).

Example usage:

```
// save solution into a file
batch.writeJSONSolution("batch-sol.json.gz");
```

5.15 GRB

Class for Java enums and constants. The enums are used to get or set Gurobi attributes or parameters.

Constants

The following list contains the set of constants needed by the Gurobi Java interface. You would refer to them using a `GRB.` prefix (e.g., `GRB.Status.OPTIMAL`).

```
// Model-status-codes

public static final int LOADED          = 1;
public static final int OPTIMAL         = 2;
public static final int INFEASIBLE      = 3;
public static final int INF_OR_UNBD     = 4;
public static final int UNBOUNDED       = 5;
public static final int CUTOFF          = 6;
public static final int ITERATION_LIMIT = 7;
public static final int NODE_LIMIT      = 8;
public static final int TIME_LIMIT      = 9;
public static final int SOLUTION_LIMIT  = 10;
public static final int INTERRUPTED     = 11;
public static final int NUMERIC         = 12;
public static final int SUBOPTIMAL      = 13;
public static final int INPROGRESS      = 14;
public static final int USER_OBJ_LIMIT  = 15;
public static final int WORK_LIMIT      = 16;
public static final int MEM_LIMIT       = 17;

public class Status {
  public static final int LOADED          = 1;
  public static final int OPTIMAL         = 2;
  public static final int INFEASIBLE      = 3;
  public static final int INF_OR_UNBD     = 4;
  public static final int UNBOUNDED       = 5;
  public static final int CUTOFF          = 6;
  public static final int ITERATION_LIMIT = 7;
  public static final int NODE_LIMIT      = 8;
  public static final int TIME_LIMIT      = 9;
  public static final int SOLUTION_LIMIT  = 10;
  public static final int INTERRUPTED     = 11;
  public static final int NUMERIC         = 12;
  public static final int SUBOPTIMAL      = 13;
  public static final int INPROGRESS      = 14;
  public static final int USER_OBJ_LIMIT  = 15;
  public static final int WORK_LIMIT      = 16;
  public static final int MEM_LIMIT       = 17;
}

// BatchStatus codes

public static final int CREATED   = 1;
public static final int SUBMITTED = 2;
public static final int ABORTED   = 3;
public static final int FAILED    = 4;
```

```
public static final int COMPLETED = 5;

public class BatchStatus {
  public static final int CREATED   = 1;
  public static final int SUBMITTED = 2;
  public static final int ABORTED   = 3;
  public static final int FAILED    = 4;
  public static final int COMPLETED = 5;
}

// Constraint senses

public static final char LESS_EQUAL    = '<';
public static final char GREATER_EQUAL = '>';
public static final char EQUAL         = '=';

// Variable types

public static final char CONTINUOUS   = 'C';
public static final char BINARY       = 'B';
public static final char INTEGER      = 'I';
public static final char SEMICONT     = 'S';
public static final char SEMIINT      = 'N';

// Objective sense

public static final int MINIMIZE = 1;
public static final int MAXIMIZE = -1;

// SOS types

public static final int SOS_TYPE1      = 1;
public static final int SOS_TYPE2      = 2;

// General constraint types

public static final int GENCONSTR_MAX       = 0;
public static final int GENCONSTR_MIN       = 1;
public static final int GENCONSTR_ABS       = 2;
public static final int GENCONSTR_AND       = 3;
public static final int GENCONSTR_OR        = 4;
public static final int GENCONSTR_NORM      = 5;
public static final int GENCONSTR_INDICATOR = 6;
public static final int GENCONSTR_PWL       = 7;
public static final int GENCONSTR_POLY      = 8;
public static final int GENCONSTR_EXP       = 9;
public static final int GENCONSTR_EXPA      = 10;
public static final int GENCONSTR_LOG       = 11;
public static final int GENCONSTR_LOGA      = 12;
public static final int GENCONSTR_POW       = 13;
public static final int GENCONSTR_SIN       = 14;
public static final int GENCONSTR_COS       = 15;
public static final int GENCONSTR_TAN       = 16;
public static final int GENCONSTR_LOGISTIC  = 17;

// Basis status info
```

```
public static final int BASIC           = 0;
public static final int NONBASIC_LOWER  = -1;
public static final int NONBASIC_UPPER  = -2;
public static final int SUPERBASIC      = -3;

// Numeric constants

public static final double INFINITY    = 1e100;
public static final double UNDEFINED   = 1e101;
public static final int    MAXINT      = 2000000000;

// Limits

public static final int MAX_STRLEN     = 512;
public static final int MAX_NAMELEN    = 255;
public static final int MAX_TAGLEN     = 10240;
public static final int MAX_CONCURRENT = 64;

// Other constants

public static final int DEFAULT_CS_PORT = 61000;

// Version numbers

public static final int VERSION_MAJOR     = 11;
public static final int VERSION_MINOR     = 0;
public static final int VERSION_TECHNICAL = 0;

// Callback constants

public static final int CB_POLLING                =     0;
public static final int CB_PRESOLVE               =     1;
public static final int CB_SIMPLEX                =     2;
public static final int CB_MIP                    =     3;
public static final int CB_MIPSOL                 =     4;
public static final int CB_MIPNODE                =     5;
public static final int CB_MESSAGE                =     6;
public static final int CB_BARRIER                =     7;
public static final int CB_MULTIOBJ               =     8;
public static final int CB_IIS                    =     9;
public static final int CB_PRE_COLDEL             =  1000;
public static final int CB_PRE_ROWDEL             =  1001;
public static final int CB_PRE_SENCHG             =  1002;
public static final int CB_PRE_BNDCHG             =  1003;
public static final int CB_PRE_COECHG             =  1004;
public static final int CB_SPX_ITRCNT             =  2000;
public static final int CB_SPX_OBJVAL             =  2001;
public static final int CB_SPX_PRIMINF            =  2002;
public static final int CB_SPX_DUALINF            =  2003;
public static final int CB_SPX_ISPERT             =  2004;
public static final int CB_MIP_OBJBST             =  3000;
public static final int CB_MIP_OBJBND             =  3001;
public static final int CB_MIP_NODCNT             =  3002;
public static final int CB_MIP_SOLCNT             =  3003;
public static final int CB_MIP_CUTCNT             =  3004;
```

```
public static final int CB_MIP_NODLFT             =  3005;
public static final int CB_MIP_ITRCNT             =  3006;
public static final int CB_MIP_OPENSCENARIOS      =  3007;
public static final int CB_MIP_PHASE              =  3008;
public static final int CB_MIPSOL_SOL             =  4001;
public static final int CB_MIPSOL_OBJ             =  4002;
public static final int CB_MIPSOL_OBJBST          =  4003;
public static final int CB_MIPSOL_OBJBND          =  4004;
public static final int CB_MIPSOL_NODCNT          =  4005;
public static final int CB_MIPSOL_SOLCNT          =  4006;
public static final int CB_MIPSOL_OPENSCENARIOS   =  4007;
public static final int CB_MIPSOL_PHASE           =  4008;
public static final int CB_MIPNODE_STATUS         =  5001;
public static final int CB_MIPNODE_REL            =  5002;
public static final int CB_MIPNODE_OBJBST         =  5003;
public static final int CB_MIPNODE_OBJBND         =  5004;
public static final int CB_MIPNODE_NODCNT         =  5005;
public static final int CB_MIPNODE_SOLCNT         =  5006;
public static final int CB_MIPNODE_BRVAR          =  5007;
public static final int CB_MIPNODE_OPENSCENARIOS  =  5008;
public static final int CB_MIPNODE_PHASE          =  5009;
public static final int CB_MSG_STRING             =  6001;
public static final int CB_RUNTIME                =  6002;
public static final int CB_WORK                   =  6003;
public static final int CB_BARRIER_ITRCNT         =  7001;
public static final int CB_BARRIER_PRIMOBJ        =  7002;
public static final int CB_BARRIER_DUALOBJ        =  7003;
public static final int CB_BARRIER_PRIMINF        =  7004;
public static final int CB_BARRIER_DUALINF        =  7005;
public static final int CB_BARRIER_COMPL          =  7006;
public static final int CB_MULTIOBJ_OBJCNT        =  8001;
public static final int CB_MULTIOBJ_SOLCNT        =  8002;
public static final int CB_MULTIOBJ_SOL           =  8003;
public static final int CB_IIS_CONSTRMIN          =  9001;
public static final int CB_IIS_CONSTRMAX          =  9002;
public static final int CB_IIS_CONSTRGUESS        =  9003;
public static final int CB_IIS_BOUNDMIN           =  9004;
public static final int CB_IIS_BOUNDMAX           =  9005;
public static final int CB_IIS_BOUNDGUESS         =  9006;

public class Callback {
  public static final int POLLING                 =     0;
  public static final int PRESOLVE                =     1;
  public static final int SIMPLEX                 =     2;
  public static final int MIP                     =     3;
  public static final int MIPSOL                  =     4;
  public static final int MIPNODE                 =     5;
  public static final int MESSAGE                 =     6;
  public static final int BARRIER                 =     7;
  public static final int MULTIOBJ                =     8;
  public static final int IIS                     =     9;
  public static final int PRE_COLDEL              =  1000;
  public static final int PRE_ROWDEL              =  1001;
  public static final int PRE_SENCHG              =  1002;
  public static final int PRE_BNDCHG              =  1003;
  public static final int PRE_COECHG              =  1004;
```

```
  public static final int SPX_ITRCNT             =  2000;
  public static final int SPX_OBJVAL             =  2001;
  public static final int SPX_PRIMINF            =  2002;
  public static final int SPX_DUALINF            =  2003;
  public static final int SPX_ISPERT             =  2004;
  public static final int MIP_OBJBST             =  3000;
  public static final int MIP_OBJBND             =  3001;
  public static final int MIP_NODCNT             =  3002;
  public static final int MIP_SOLCNT             =  3003;
  public static final int MIP_CUTCNT             =  3004;
  public static final int MIP_NODLFT             =  3005;
  public static final int MIP_ITRCNT             =  3006;
  public static final int MIP_OPENSCENARIOS      =  3007;
  public static final int MIP_PHASE              =  3008;
  public static final int MIPSOL_SOL             =  4001;
  public static final int MIPSOL_OBJ             =  4002;
  public static final int MIPSOL_OBJBST          =  4003;
  public static final int MIPSOL_OBJBND          =  4004;
  public static final int MIPSOL_NODCNT          =  4005;
  public static final int MIPSOL_SOLCNT          =  4006;
  public static final int MIPSOL_OPENSCENARIOS   =  4007;
  public static final int MIPSOL_PHASE           =  4008;
  public static final int MIPNODE_STATUS         =  5001;
  public static final int MIPNODE_REL            =  5002;
  public static final int MIPNODE_OBJBST         =  5003;
  public static final int MIPNODE_OBJBND         =  5004;
  public static final int MIPNODE_NODCNT         =  5005;
  public static final int MIPNODE_SOLCNT         =  5006;
  public static final int MIPNODE_BRVAR          =  5007;
  public static final int MIPNODE_OPENSCENARIOS  =  5008;
  public static final int MIPNODE_PHASE          =  5009;
  public static final int MSG_STRING             =  6001;
  public static final int RUNTIME                =  6002;
  public static final int WORK                   =  6003;
  public static final int BARRIER_ITRCNT         =  7001;
  public static final int BARRIER_PRIMOBJ        =  7002;
  public static final int BARRIER_DUALOBJ        =  7003;
  public static final int BARRIER_PRIMINF        =  7004;
  public static final int BARRIER_DUALINF        =  7005;
  public static final int BARRIER_COMPL          =  7006;
  public static final int MULTIOBJ_OBJCNT        =  8001;
  public static final int MULTIOBJ_SOLCNT        =  8002;
  public static final int MULTIOBJ_SOL           =  8003;
  public static final int IIS_CONSTRMIN          =  9001;
  public static final int IIS_CONSTRMAX          =  9002;
  public static final int IIS_CONSTRGUESS        =  9003;
  public static final int IIS_BOUNDMIN           =  9004;
  public static final int IIS_BOUNDMAX           =  9005;
  public static final int IIS_BOUNDGUESS         =  9006;
}

// Errors

public static final int ERROR_OUT_OF_MEMORY           = 10001;
public static final int ERROR_NULL_ARGUMENT           = 10002;
public static final int ERROR_INVALID_ARGUMENT        = 10003;
```

```
public static final int ERROR_UNKNOWN_ATTRIBUTE        = 10004;
public static final int ERROR_DATA_NOT_AVAILABLE       = 10005;
public static final int ERROR_INDEX_OUT_OF_RANGE       = 10006;
public static final int ERROR_UNKNOWN_PARAMETER        = 10007;
public static final int ERROR_VALUE_OUT_OF_RANGE       = 10008;
public static final int ERROR_NO_LICENSE               = 10009;
public static final int ERROR_SIZE_LIMIT_EXCEEDED      = 10010;
public static final int ERROR_CALLBACK                 = 10011;
public static final int ERROR_FILE_READ                = 10012;
public static final int ERROR_FILE_WRITE               = 10013;
public static final int ERROR_NUMERIC                  = 10014;
public static final int ERROR_IIS_NOT_INFEASIBLE       = 10015;
public static final int ERROR_NOT_FOR_MIP              = 10016;
public static final int ERROR_OPTIMIZATION_IN_PROGRESS = 10017;
public static final int ERROR_DUPLICATES               = 10018;
public static final int ERROR_NODEFILE                 = 10019;
public static final int ERROR_Q_NOT_PSD                = 10020;
public static final int ERROR_QCP_EQUALITY_CONSTRAINT  = 10021;
public static final int ERROR_NETWORK                  = 10022;
public static final int ERROR_JOB_REJECTED             = 10023;
public static final int ERROR_NOT_SUPPORTED            = 10024;
public static final int ERROR_EXCEED_2B_NONZEROS       = 10025;
public static final int ERROR_INVALID_PIECEWISE_OBJ    = 10026;
public static final int ERROR_UPDATEMODE_CHANGE        = 10027;
public static final int ERROR_CLOUD                    = 10028;
public static final int ERROR_MODEL_MODIFICATION       = 10029;
public static final int ERROR_CSWORKER                 = 10030;
public static final int ERROR_TUNE_MODEL_TYPES         = 10031;
public static final int ERROR_SECURITY                 = 10032;
public static final int ERROR_NOT_IN_MODEL             = 20001;
public static final int ERROR_FAILED_TO_CREATE_MODEL   = 20002;
public static final int ERROR_INTERNAL                 = 20003;

public class Error {
  public static final int OUT_OF_MEMORY            = 10001;
  public static final int NULL_ARGUMENT            = 10002;
  public static final int INVALID_ARGUMENT         = 10003;
  public static final int UNKNOWN_ATTRIBUTE        = 10004;
  public static final int DATA_NOT_AVAILABLE       = 10005;
  public static final int INDEX_OUT_OF_RANGE       = 10006;
  public static final int UNKNOWN_PARAMETER        = 10007;
  public static final int VALUE_OUT_OF_RANGE       = 10008;
  public static final int NO_LICENSE               = 10009;
  public static final int SIZE_LIMIT_EXCEEDED      = 10010;
  public static final int CALLBACK                 = 10011;
  public static final int FILE_READ                = 10012;
  public static final int FILE_WRITE               = 10013;
  public static final int NUMERIC                  = 10014;
  public static final int IIS_NOT_INFEASIBLE       = 10015;
  public static final int NOT_FOR_MIP              = 10016;
  public static final int OPTIMIZATION_IN_PROGRESS = 10017;
  public static final int DUPLICATES               = 10018;
  public static final int NODEFILE                 = 10019;
  public static final int ERROR_Q_NOT_PSD          = 10020;
  public static final int QCP_EQUALITY_CONSTRAINT  = 10021;
  public static final int NETWORK                  = 10022;
```

```
  public static final int JOB_REJECTED             = 10023;
  public static final int NOT_SUPPORTED            = 10024;
  public static final int EXCEED_2B_NONZEROS       = 10025;
  public static final int INVALID_PIECEWISE_OBJ    = 10026;
  public static final int UPDATEMODE_CHANGE        = 10027;
  public static final int CLOUD                    = 10028;
  public static final int MODEL_MODIFICATION       = 10029;
  public static final int CSWORKER                 = 10030;
  public static final int TUNE_MODEL_TYPES         = 10031;
  public static final int ERROR_SECURITY           = 10032;
  public static final int NOT_IN_MODEL             = 20001;
  public static final int FAILED_TO_CREATE_MODEL   = 20002;
  public static final int INTERNAL                 = 20003;
}

// Cuts parameter values

public static final int CUTS_AUTO           = -1;
public static final int CUTS_OFF            = 0;
public static final int CUTS_CONSERVATIVE   = 1;
public static final int CUTS_AGGRESSIVE     = 2;
public static final int CUTS_VERYAGGRESSIVE = 3;

// Presolve parameter values

public static final int PRESOLVE_AUTO         = -1;
public static final int PRESOLVE_OFF          =  0;
public static final int PRESOLVE_CONSERVATIVE =  1;
```

GRB.CharAttr

This enum is used to get or set char-valued attributes (through GRBModel.get or GRBModel.set). Please refer to the Attributes section of the Reference Manual to see a list of all char attributes and their functions.

GRB.DoubleAttr

This enum is used to get or set double-valued attributes (through GRBModel.get or GRBModel.set). Please refer to the Attributes section of the Reference Manual to see a list of all double attributes and their functions.

GRB.DoubleParam

This enum is used to get or set double-valued parameters (through GRBModel.get, GRBModel.set, GRBEnv.get, or GRBEnv.set). Please refer to the Parameters section of the Reference Manual to see a list of all double parameters and their functions.

GRB.IntAttr

This enum is used to get or set int-valued attributes (through GRBModel.get or GRBModel.set). Please refer to the Attributes section of the Reference Manual to see a list of all int attributes and their functions.

GRB.IntParam

This enum is used to get or set int-valued parameters (through GRBModel.get, GRBModel.set, GRBEnv.get, GRBEnv.set). Please refer to the Parameters section of the Reference Manual to see a list of all int parameters and their functions.

GRB.StringAttr

This enum is used to get or set string-valued attributes (through GRBModel.get or GRBModel.set). Please refer to the Attributes section of the Reference Manual to see a list of all string attributes and their functions.

GRB.StringParam

This enum is used to get or set string-valued parameters (through GRBModel.get, GRBModel.set, GRBEnv.get, or GRBEnv.set). Please refer to the Parameters section of the Reference Manual to see a list of all string parameters and their functions.

.NET API Overview

This section documents the Gurobi .NET interface. This manual begins with a quick overview of the classes exposed in the interface and the most important methods on those classes. It then continues with a comprehensive presentation of all of the available classes and methods.

If you are new to the Gurobi Optimizer, we suggest that you start with the Getting Started Knowledge Base article or the Example Tour. These documents provide concrete examples of how to use the classes and methods described here.

Environments

The first step in using the Gurobi .NET interface is to create an environment object. Environments are represented using the GRBEnv class. An environment acts as the container for all data associated with a set of optimization runs. You will generally only need one environment object in your program.

For more advanced use cases, you can use an empty environment to create an uninitialized environment and then, programmatically, set all required options for your specific requirements. For further details see the Environment section.

Models

You can create one or more optimization models within an environment. Each model is represented as an object of class GRBModel. A model consists of a set of decision variables (objects of class GRBVar), a linear or quadratic objective function on those variables (specified using GRBModel.SetObjective), and a set of constraints on these variables (objects of class GRBConstr, GRBQConstr, GRBSOS, or GRBGenConstr). Each variable has an associated lower bound, upper bound, and type (continuous, binary, etc.). Each linear or quadratic constraint has an associated sense (less-than-or-equal, greater-than-or-equal, or equal), and right-hand side value. Refer to this section for more information on variables, constraints, and objectives.

Linear constraints are specified by building linear expressions (objects of class GRBLinExpr), and then specifying relationships between these expressions (for example, requiring that one expression be equal to another). Quadratic constraints are built in a similar fashion, but using quadratic expressions (objects of class GRBQuadExpr) instead.

An optimization model may be specified all at once, by loading the model from a file (using the appropriate GRBModel constructor), or built incrementally, by first constructing an empty object of class GRBModel and then subsequently calling GRBModel.AddVar or GRBModel.AddVars to add additional variables, and GRBModel.AddConstr, GRBModel.AddQConstr, GRBModel.AddSOS, or any of the GRBModel.AddGenConstrXxx methods to add additional constraints. Models are dynamic entities; you can always add or remove variables or constraints.

We often refer to the *class* of an optimization model. At the highest level, a model can be continuous or discrete, depending on whether the modeling elements present in the model require discrete decisions to be made. Among continuous models...

- A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*.

- If the objective is quadratic, the model is a *Quadratic Program (QP)*.

- If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We sometimes refer to a few special cases of QCP: QCPs with convex constraints, QCPs with non-convex constraints, *bilinear programs*, and *Second-Order Cone Programs (SOCP)*.

- If any of the constraints are non-linear (chosen from among the available general constraints), the model is a *Non-Linear Program (NLP)*.

A model that contains any integer variables, semi-continuous variables, semi-integer variables, Special Ordered Set (SOS) constraints, or general constraints, is discrete, and is referred to as a *Mixed Integer Program (MIP)*. The special cases of MIP, which are the discrete versions of the continuous models types we've already described, are...

- *Mixed Integer Linear Programs (MILP)*

- *Mixed Integer Quadratic Programs (MIQP)*

- *Mixed Integer Quadratically-Constrained Programs (MIQCP)*

- *Mixed Integer Second-Order Cone Programs (MISOCP)*

- *Mixed Integer Non-Linear Programs (MINLP)*

The Gurobi Optimizer handles all of these model classes. Note that the boundaries between them aren't as clear as one might like, because we are often able to transform a model from one class to a simpler class.

Solving a Model

Once you have built a model, you can call GRBModel.Optimize to compute a solution. By default, Optimize will use the concurrent optimizer to solve LP models, the barrier algorithm to solve QP models with convex objectives and QCP models with convex constraints, and the branch-and-cut algorithm otherwise. The solution is stored in a set of *attributes* of the model. These attributes can be queried using a set of attribute query methods on the GRBModel, GRBVar, GRBConstr, GRBQConstr, GRBSOS, and GRBGenConstr classes.

The Gurobi algorithms keep careful track of the state of the model, so calls to GRBModel.Optimize will only perform further optimization if relevant data has changed since the model was last optimized. If you would like to discard previously computed solution information and restart the optimization from scratch without changing the model, you can call GRBModel.Reset.

After a MIP model has been solved, you can call GRBModel.FixedModel to compute the associated *fixed* model. This model is identical to the original, except that the integer variables are fixed to their values in the MIP solution. If your model contains SOS constraints, some continuous variables that appear in these constraints may be fixed as well. In some applications, it can be useful to compute information on this fixed model (e.g., dual variables, sensitivity information, etc.), although you should be careful in how you interpret this information.

Multiple Solutions, Objectives, and Scenarios

By default, the Gurobi Optimizer assumes that your goal is to find one proven optimal solution to a single model with a single objective function. Gurobi provides the following features that allow you to relax these assumptions:

- Solution Pool: Allows you to find more solutions.
- Multiple Scenarios: Allows you to find solutions to multiple, related models.
- Multiple Objectives: Allows you to specify multiple objective functions and control the trade-off between them.

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call GRBModel.ComputeIIS to compute an Irreducible Inconsistent Subsystem (IIS). This method can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. This method populates a set of IIS attributes.

To attempt to repair an infeasibility, call GRBModel.FeasRelax to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Querying and Modifying Attributes

Most of the information associated with a Gurobi model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. To give a simple example, solving an optimization model causes the `X` variable attribute to be populated. Attributes such as `X` that are computed by the Gurobi Optimizer cannot be modified directly by the user, while others, such as the variable lower bound (the `LB` attribute) can.

Attributes can be accessed in two ways in the .NET interface. The easiest is through .NET properties. To query or modify the `LB` attribute on variable `v`, you would use `v.LB` or `v.LB = 0`, respectively. Attributes can also be queried using GRBVar.Get, GRBConstr.Get, GRBQConstr.Get, GRBSOS.Get, GRBGenConstr.Get, or GRBModel.Get, and modified using GRBVar.Set, GRBConstr.Set, GRBQConstr.Set, GRBGenConstr.Set, or GRBModel.Set. Attributes are grouped into a set of enums by type (GRB.CharAttr, GRB.DoubleAttr, GRB.IntAttr, GRB.StringAttr). The `Get()` and `Set()` methods are overloaded, so the type of the attribute determines the type of the returned value. Thus, `constr.Get(GRB.DoubleAttr.RHS)` returns a double, while `constr.Get(GRB.CharAttr.Sense)` returns a char.

If you wish to retrieve attribute values for a set of variables or constraints, it is usually more efficient to use the array methods on the associated GRBModel object. Method GRBModel.Get includes signatures that allow you to query or modify attribute values for one-, two-, and three-dimensional arrays of variables or constraints.

The full list of attributes can be found in the Attributes section.

Additional Model Modification Information

Most modifications to an existing model are done through the attribute interface (e.g., changes to variable bounds, constraint right-hand sides, etc.). The main exceptions are modifications to the constraint matrix and to the objective function.

The constraint matrix can be modified in a few ways. The first is to call the ChgCoeff method on a GRBModel object to change individual matrix coefficients. This method can be used to modify the value of an existing non-zero, to set an existing non-zero to zero, or to create a new non-zero. The constraint matrix is also modified when you remove a variable or constraint from the model (through the GRBModel.Remove method). The non-zero values associated with the deleted constraint or variable are removed along with the constraint or variable itself.

The model objective function can also be modified in a few ways. The easiest is to build an expression that captures the objective function (a GRBLinExpr or GRBQuadExpr object), and then pass that expression to method GRBModel.SetObjective. If you wish to modify the objective, you can simply call `setObjective` again with a new `GRBLinExpr` or `GRBQuadExpr` object.

For linear objective functions, an alternative to `SetObjective` is to use the `Obj` variable attribute to modify individual linear objective coefficients.

If your variables have piecewise-linear objectives, you can specify them using the GRBModel.SetPWLObj method. Call this method once for each relevant variable. The Gurobi simplex solver includes algorithmic support for convex piecewise-linear objective functions, so for continuous models you should see a substantial performance benefit from using this feature. To clear a previously specified piecewise-linear objective function, simply set the `Obj` attribute on the corresponding variable to 0.

Lazy Updates

One important item to note about model modification in the Gurobi optimizer is that it is performed in a *lazy* fashion, meaning that modifications don't affect the model immediately. Rather, they are queued and applied later. If your program simply creates a model and solves it, you will probably never notice this behavior. However, if you ask for information about the model before your modifications have been applied, the details of the lazy update approach may be relevant to you.

As we just noted, model modifications (bound changes, right-hand side changes, objective changes, etc.) are placed in a queue. These queued modifications can be applied to the model in three different ways. The first is by an explicit call to GRBModel.Update. The second is by a call to GRBModel.Optimize. The third is by a call to GRBModel.Write to write out the model. The first case gives you fine-grained control over when modifications are applied. The second and third make the assumption that you want all pending modifications to be applied before you optimize your model or write it to disk.

Why does the Gurobi interface behave in this manner? There are a few reasons. The first is that this approach makes it much easier to perform multiple modifications to a model, since the model remains unchanged between modifications. The second is that processing model modifications can be expensive, particularly in a Compute Server environment, where modifications require communication between machines. Thus, it is useful to have visibility into exactly when these modifications are applied. In general, if your program needs to make multiple modifications to the model, you should aim to make them in phases, where you make a set of modifications, then update, then make more modifications, then update again, etc. Updating after each individual

modification can be extremely expensive.

If you forget to call update, your program won't crash. Your query will simply return the value of the requested data from the point of the last update. If the object you tried to query didn't exist then, you'll get a `NOT_IN_MODEL` exception instead.

The semantics of lazy updates have changed since earlier Gurobi versions. While the vast majority of programs are unaffected by this change, you can use the UpdateMode parameter to revert to the earlier behavior if you run into an issue.

Managing Parameters

The Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization. Parameters can be of type *int*, *double*, or *string*.

The simplest way to set parameters is through the `Model.Parameters` class and its associated .NET properties. To set the MIPGap parameter to 0.0 for model `m`, for example, you would do `m.Parameters.MIPGap = 0`.

Parameters can also be set on the Gurobi environment object, using GRBEnv.Set. Note that each model gets its own copy of the environment when it is created, so parameter changes to the original environment have no effect on existing models.

You can read a set of parameter settings from a file using GRBEnv.ReadParams, or write the set of changed parameters using GRBEnv.WriteParams.

We also include an automated parameter tuning tool that explores many different sets of parameter changes in order to find a set that improves performance. You can call GRBModel.Tune to invoke the tuning tool on a model. Refer to the parameter tuning tool section for more information.

The full list of Gurobi parameters can be found in the Parameters section.

Memory Management

Users typically do not need to concern themselves with memory management in .NET, since it is handled automatically by the garbage collector. The Gurobi .NET interface utilizes the same garbage collection mechanism as other .NET programs, but there are a few specifics of our memory management that users should be aware of.

In general, Gurobi objects live in the same .NET heap as other .NET objects. When they are no longer referenced, they become candidates for garbage collection, and are returned to the pool of free space at the next invocation of the garbage collector. Two important exceptions are the GRBEnv and GRBModel objects. A `GRBModel` object has a small amount of memory associated with it in the .NET heap, but the majority of the space associated with a model lives in the heap of the Gurobi native code DLL. The .NET heap manager is unaware of the memory associated with the model in the native code library, so it does not consider this memory usage when deciding whether to invoke the garbage collector. When the garbage collector eventually collects the .NET `GRBModel` object, the memory associated with the model in the Gurobi native code library will be freed, but this collection may come later than you might want. Similar considerations apply to the `GRBEnv` object.

If you are writing a .NET program that makes use of multiple Gurobi models or environments, we recommend that you call GRBModel.Dispose when you are done using the associated `GRBModel` object, and GRBEnv.Dispose when you are done using the associated `GRBEnv` object and after you have called GRBModel.Dispose on all of the models created using that `GRBEnv` object.

Native Code

As noted earlier, the Gurobi .NET interface is a thin layer that sits on top of our native code DLL. Thus, an application that uses the Gurobi .NET library will load the Gurobi DLL at runtime. In order for this happen, you need to make sure that two things are true. First, you need to make sure that the native code library is available in the `PATH` environment library. This environment variable is set up as part of the installation of the Gurobi Optimizer, but it may not be configured appropriately on a machine where the full Gurobi Optimizer has not been installed. Second, you need to be sure that the selected .NET Platform Target (as selected in Visual Studio) is compatible with the Gurobi DLL that is available through your `PATH`. In particular, you need to use the 64-bit Gurobi native library when you've selected the `x64` Platform Target. If you use the default `Any CPU` target, then your .NET application will look for the 64-bit Gurobi DLL on a 64-bit machine.

Monitoring Progress - Logging and Callbacks

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging behavior. If you would like to direct output to a file as well as to the screen, specify the log file name in the GRBEnv constructor. You can modify the LogFile parameter if you wish to redirect the log to a different file after creating the environment object. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter. A detailed description of the Gurobi log file can be found in the Logging section.

More detailed progress monitoring can be done through the GRBCallback class. The GRBModel.SetCallback method allows you to receive a periodic callback from the Gurobi Optimizer. You do this by sub-classing the GRBCallback abstract class, and writing your own `Callback()` method on this class. You can call GRBCallback.GetDoubleInfo, GRBCallback.GetIntInfo, GRBCallback.GetStringInfo, or GRBCallback.GetSolution from within the callback to obtain additional information about the state of the optimization.

Modifying Solver Behavior - Callbacks

Callbacks can also be used to modify the behavior of the Gurobi optimizer. The simplest control callback is GRBCallback.Abort, which asks the optimizer to terminate at the earliest convenient point. Method GRBCallback.SetSolution allows you to inject a feasible solution (or partial solution) during the solution of a MIP model. Methods GRBCallback.AddCut and GRBCallback.AddLazy allow you to add *cutting planes* and *lazy constraints* during a MIP optimization, respectively. Method GRBCallback.StopOneMultiObj allows you to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

Batch Optimization

Gurobi Compute Server enables programs to offload optimization computations onto dedicated servers. The Gurobi Cluster Manager adds a number of additional capabilities on top of this. One important one, *batch optimization*, allows you to build an optimization model with your client program, submit it to a Compute Server cluster (through the Cluster Manager), and later check on the status of the model and retrieve its solution. You can use a Batch object to make it easier to work with batches. For details on batches, please refer to the Batch Optimization section.

Error Handling

All of the methods in the Gurobi .NET library can throw an exception of type GRBException. When an exception occurs, additional information on the error can be obtained by retrieving the error code (using property GRBException.ErrorCode), or by retrieving the exception message (using property `GRBException.Message` from the parent class). The list of possible error return codes can be found in the Error Codes section.

6.1 GRBEnv

Gurobi environment object. Gurobi models are always associated with an environment. You must create an environment before can you create and populate a model. You will generally only need a single environment object in your program.

Objects of this class have unmanaged resources associated with them. The class implements the `IDisposable` interface.

The methods on environment objects are mainly used to manage Gurobi parameters (e.g., Get, GetParamInfo, Set).

While the .NET garbage collector will eventually collect an unused `GRBEnv` object, an environment will hold onto resources (Gurobi licenses, file descriptors, etc.) until that collection occurs. If your program creates multiple `GRBEnv` objects, we recommend that you call GRBEnv.Dispose when you are done using one (or use the .NET `using` statement).

GRBEnv()

Constructor for `GRBEnv` object. You have the option of constructing either a local environment, which solves Gurobi models on the local machine, a client environment for a Gurobi Compute Server, which will solve Gurobi models on a server machine, or an Instant Cloud environment, which will launch a Gurobi Cloud server and solve models on that server. Choose the appropriate signature for the type of environment you wish to launch.

```
GRBEnv  GRBEnv ( )
```

Create a Gurobi environment (with logging disabled). This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Return value:

An environment object (with no associated log file).

```
GRBEnv  GRBEnv (  string  logFileName )
```

Create a Gurobi environment (with logging enabled). This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded

program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`logFileName`: The desired log file name.

Return value:

An environment object.

```
GRBEnv  GRBEnv (  bool  empty )
```

Create an empty Gurobi environment. Use Start to start the environment.

If the environment is not empty, This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`empty`: Indicates whether the environment should be empty. You should use `empty=True` if you want to set parameters before actually starting the environment. This can be useful if you want to connect to a Compute Server, a Token Server, the Gurobi Instant Cloud, a Cluster Manager or use a WLS license. See the Environment Section for more details.

Return value:

An environment object.

GRBEnv.Dispose()

Release the resources associated with a `GRBEnv` object. While the .NET garbage collector will eventually reclaim these resources, we recommend that you call the `Dispose` method when you are done using an environment if your program creates more than one.

The `Dispose` method on a `GRBEnv` should be called only after you have called `Dispose` on all of the models that were created within that environment. You should not attempt to use a `GRBEnv` object after calling `Dispose`.

```
void  Dispose ( )
```

GRBEnv.ErrorMsg

```
string  ErrorMsg
```

(Property) The error message for the most recent exception associated with this environment.

GRBEnv.Get()

Query the value of a parameter.

```
double  Get ( GRB.DoubleParam  param )
```

Query the value of a double-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

```
int  Get ( GRB.IntParam  param )
```

Query the value of an int-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

```
string  Get ( GRB.StringParam  param )
```

Query the value of a string-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

GRBEnv.GetParamInfo()

Obtain information about a parameter.

```
void  GetParamInfo ( GRB.DoubleParam  param,
                     double[]         info )
```

Obtain detailed information about a double parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

info: The returned information. The result will contain four entries: the current value of the parameter, the minimum allowed value, the maximum allowed value, and the default value.

```
void  GetParamInfo ( GRB.IntParam  param,
                     int[]         info )
```

Obtain detailed information about an integer parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

info: The returned information. The result will contain four entries: the current value of the parameter, the minimum allowed value, the maximum allowed value, and the default value.

```
void  GetParamInfo ( GRB.StringParam  param,
                     string[]         info )
```

Obtain detailed information about a string parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

info: The returned information. The result will contain two entries: the current value of the parameter and the default value.

GRBEnv.Message()

Write a message to the console and the log file.

```
void  Message ( string  message )
```

Arguments:

message: Print a message to the console and to the log file. Note that this call has no effect unless the `OutputFlag` parameter is set.

GRBEnv.ReadParams()

Read new parameter settings from a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  ReadParams ( string  paramfile )
```

Arguments:

paramfile: Name of the file containing parameter settings. Parameters should be listed one per line, with the parameter name first and the desired value second. For example:

```
# Gurobi parameter file
Threads 1
MIPGap 0
```

Blank lines and lines that begin with the hash symbol are ignored.

GRBEnv.Release()

Release the license associated with this environment. You will no longer be able to call Optimize on models created with this environment after the license has been released.

```
void  Release ( )
```

GRBEnv.ResetParams()

Reset all parameters to their default values.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  ResetParams ( )
```

GRBEnv.Set()

Set the value of a parameter.

Important notes:

Note that a model gets its own copy of the environment when it is created. Changes to the original environment have no effect on the copy. Use GRBModel.Set to change a parameter on an existing model.

```
void  Set ( GRB.DoubleParam  param,
            double           newvalue )
```

Set the value of a double-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  Set ( GRB.IntParam  param,
            int           newvalue )
```

Set the value of an int-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  Set ( GRB.StringParam  param,
            string           newvalue )
```

Set the value of a string-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  Set ( string  param,
            string  newvalue )
```

Set the value of any parameter using strings alone.

Arguments:

param: The name of the parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

GRBEnv.Start()

Start an empty environment. If the environment has already been started, this method will do nothing. If the call fails, the environment will have the same state as it had before the call to this method.

This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter). After that, it will apply all parameter changes specified by the user prior to this call. Note that this might overwrite parameters set in the license file, or in the `gurobi.env` file, if present.

After all these changes are performed, the code will actually activate the environment, and make it ready to work with models.

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  Start ( )
```

GRBEnv.WriteParams()

Write all non-default parameter settings to a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  WriteParams ( string  paramfile )
```

Arguments:

paramfile: Name of the file to which non-default parameter settings should be written. The previous contents are overwritten.

6.2 GRBModel

Gurobi model object. Commonly used methods include AddVar (adds a new decision variable to the model), AddConstr (adds a new constraint to the model), Optimize (optimizes the current model), and Get (retrieves the value of an attribute).

Objects of this class have unmanaged resources associated with them. The class implements the `IDisposable` interface.

While the .NET garbage collector will eventually collect an unused `GRBModel` object, the vast majority of the memory associated with a model is stored outside of the .NET heap. As a result, the garbage collector can't see this memory usage, and thus it can't take this quantity into account when deciding whether collection is necessary. We recommend that you call GRBModel.Dispose when you are done using a model (or use the .NET `using` statement).

GRBModel()

Constructor for `GRBModel`. The simplest version creates an empty model. You can then call AddVar and AddConstr to populate the model with variables and constraints. The more complex constructors can read a model from a file, or make a copy of an existing model.

```
GRBModel  GRBModel ( GRBEnv  env )
```

Model constructor.

Arguments:

`env`: Environment for new model.

Return value:

New model object. Model initially contains no variables or constraints.

```
GRBModel  GRBModel ( GRBEnv  env,
                     string  filename )
```

Read a model from a file. Note that the type of the file is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, `.rlp`, `.dua`, `.dlp`, `.ilp`, or `.opb`. The files can be compressed, so additional suffixes of `.zip`, `.gz`, `.bz2`, or `.7z` are accepted.

Arguments:

`env`: Environment for new model.

`modelname`: Name of the file containing the model.

Return value:

New model object.

```
GRBModel  GRBModel ( GRBModel  model )
```

Create a copy of an existing model. Note that due to the lazy update approach in Gurobi, you have to call Update before copying it.

Arguments:

`model`: Model to copy.

Return value:

New model object. Model is a clone of the original.

```
GRBModel  GRBModel ( GRBModel  model,
                     GRBEnv    targetenv )
```

Copy an existing model to a different environment. Multiple threads can not work simultaneously within the same environment. Copies of models must therefore reside in different environments for multiple threads to operate on them simultaneously.

Note that this method itself is not thread safe, so you should either call it from the main thread or protect access to it with a lock.

Note that pending updates will not be applied to the model, so you should call update before copying if you would like those to be included in the copy.

For Compute Server users, note that you can copy a model from a client to a Compute Server environment, but it is not possible to copy models from a Compute Server environment to another (client or Compute Server) environment.

Arguments:

`model`: Model to copy.

`targetenv`: Environment to copy model into.

Return value:

New model object. Model is a clone of the original.

GRBModel.AddConstr()

Add a single linear constraint to a model. Multiple signatures are available.

```
GRBConstr  AddConstr ( GRBLinExpr  lhsExpr,
                       char        sense,
                       GRBLinExpr  rhsExpr,
                       string      name )
```

Add a single linear constraint to a model.

Arguments:

`lhsExpr`: Left-hand side expression for new linear constraint.

`sense`: Sense for new linear constraint (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

`rhsExpr`: Right-hand side expression for new linear constraint.

`name`: Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  AddConstr ( GRBTempConstr  tempConstr,
                       string         name )
```

Add a single linear constraint to a model.

Arguments:

`tempConstr`: Temporary constraint object, created by an overloaded comparison operator. See GRBTempConstr for more information.

`name`: Name for new constraint.

Return value:

New constraint object.

GRBModel.AddConstrs()

Add new linear constraints to a model.

We recommend that you build your model one constraint at a time (using AddConstr), since it introduces no significant overhead and we find that it produces simpler code. Feel free to use these methods if you disagree, though.

```
GRBConstr[]  AddConstrs (  int  count )
```

Add `count` new linear constraints to a model. The new constraints are all of the form `0 <= 0`.

Arguments:

`count`: Number of constraints to add.

Return value:

Array of new constraint objects.

```
GRBConstr[]  AddConstrs (  GRBLinExpr[]  lhsExprs,
                           char[]        senses,
                           double[]      rhsVals,
                           string[]      names )
```

Add new linear constraints to a model. The number of added constraints is determined by the length of the input arrays (which must be consistent across all arguments).

Arguments:

`lhsExprs`: Left-hand side expressions for the new linear constraints.

`senses`: Senses for new linear constraints (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

`rhsVals`: Right-hand side values for the new linear constraints.

`names`: Names for new constraints.

Return value:

Array of new constraint objects.

```
GRBConstr[]  AddConstrs (  GRBLinExpr[]  lhsExprs,
                           char[]        senses,
                           GRBLinExpr[]  rhsExprs,
                           int           start,
                           int           len,
                           string[]      names )
```

Add new linear constraints to a model. This signature allows you to use arrays to hold the various constraint attributes (left-hand side, sense, etc.), without forcing you to add one constraint for each entry in the array. The `start` and `len` arguments allow you to specify which constraints to add.

Arguments:

`lhsExprs`: Left-hand side expressions for the new linear constraints.

`senses`: Senses for new linear constraints (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

`rhs`: Right-hand side expressions for the new linear constraints.

`start`: The first constraint in the list to add.

len: The number of variables to add.

names: Names for new constraints.

Return value:

Array of new constraint objects.

GRBModel.AddGenConstrXxx()

Each of the functions described below adds a new *general constraint* to a model.

Mathematical programming has traditionally defined a set of fundamental constraint types: variable bound constraints, linear constraints, quadratic constraints, integrality constraints, and SOS constraints. These are typically treated directly by the underlying solver (although not always), and are fundamental to the overall algorithm.

Gurobi accepts a number of additional constraint types, which we collectively refer to as *general (function) constraints*. These are typically *not* treated directly by the solver. Rather, they are transformed by presolve into constraints (and variables) chosen from among the fundamental types listed above. In some cases, the resulting constraint or constraints are mathematically equivalent to the original; in others, they are approximations. If such constraints appear in your model, but if you prefer to reformulate them yourself using fundamental constraint types instead, you can certainly do so. However, note that Gurobi can sometimes exploit information contained in the other constraints in the model to build a more efficient formulation than what you might create.

The additional constraint types that fall under this *general constraint* umbrella are:

- AddGenConstrMax: $y = \max(x_1, x_2, ..., c)$
- AddGenConstrMin: $y = \min(x_1, x_2, ..., c)$
- AddGenConstrAbs: $y = |x|$
- AddGenConstrAnd: $y = x_1 \wedge x_2 \wedge x_3 ...$
- AddGenConstrOr: $y = x_1 \vee x_2 \vee x_3 ...$
- AddGenConstrNorm: $y = norm(x_1, x_2, x_3 ...)$
- AddGenConstrIndicator: $y = 1 \rightarrow a'x \leq b$ (an indicator constraint)
- AddGenConstrPWL: $y = pwl(x)$ (a piecewise-linear function, specified using breakpoints)
- AddGenConstrPoly: $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$
- AddGenConstrExp: $y = e^x$
- AddGenConstrExpA: $y = a^x$
- AddGenConstrLog: $y = \log_e(x)$
- AddGenConstrLogA: $y = \log_a(x)$
- AddGenConstrLogistic: $y = \frac{1}{1+e^{-x}}$
- AddGenConstrPow: $y = x^a$

- AddGenConstrSin: $y = \sin(x)$
- AddGenConstrCos: $y = \cos(x)$
- AddGenConstrTan: $y = \tan(x)$

Please refer to this section for additional details on general constraints.

GRBModel.AddGenConstrMax()

Add a new general constraint of type `GRB.GENCONSTR_MAX` to a model.

A MAX constraint $r = \max\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the maximum of the operand variables $x_1, \ldots, x_n$ and the constant c.

```
GRBGenConstr  AddGenConstrMax (  GRBVar    resvar,
                                 GRBVar[]  vars,
                                 double    constant,
                                 string    name )
```

Arguments:

`resvar`: The resultant variable of the new constraint.

`vars`: Array of variables that are the operands of the new constraint.

`constant`: The additional constant operand of the new constraint.

`name`: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.AddGenConstrMin()

Add a new general constraint of type `GRB.GENCONSTR_MIN` to a model.

A MIN constraint $r = \min\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the minimum of the operand variables $x_1, \ldots, x_n$ and the constant c.

```
GRBGenConstr  AddGenConstrMin (  GRBVar    resvar,
                                 GRBVar[]  vars,
                                 double    constant,
                                 string    name )
```

Arguments:

`resvar`: The resultant variable of the new constraint.

`vars`: Array of variables that are the operands of the new constraint.

`constant`: The additional constant operand of the new constraint.

`name`: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.AddGenConstrAbs()

Add a new general constraint of type `GRB.GENCONSTR_ABS` to a model.

An ABS constraint $r = \text{abs}\{x\}$ states that the resultant variable r should be equal to the absolute value of the argument variable x.

```
GRBGenConstr  AddGenConstrAbs (  GRBVar  resvar,
                                 GRBVar  argvar,
                                 string  name )
```

Arguments:

resvar: The resultant variable of the new constraint.

argvar: The argument variable of the new constraint.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.AddGenConstrAnd()

Add a new general constraint of type `GRB.GENCONSTR_AND` to a model.

An AND constraint $r = \text{and}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if all of the operand variables $x_1, \ldots, x_n$ are equal to 1. If any of the operand variables is 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

```
GRBGenConstr  AddGenConstrAnd (  GRBVar    resvar,
                                 GRBVar[]  vars,
                                 string    name )
```

Arguments:

resvar: The resultant variable of the new constraint.

vars: Array of variables that are the operands of the new constraint.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.AddGenConstrOr()

Add a new general constraint of type `GRB.GENCONSTR_OR` to a model.

An OR constraint $r = \text{or}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if any of the operand variables $x_1, \ldots, x_n$ is equal to 1. If all operand variables are 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

```
GRBGenConstr  AddGenConstrOr (  GRBVar    resvar,
                                GRBVar[]  vars,
                                string    name )
```

Arguments:

resvar: The resultant variable of the new constraint.

vars: Array of variables that are the operands of the new constraint.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.AddGenConstrNorm()

Add a new general constraint of type `GRB.GENCONSTR_NORM` to a model.

A NORM constraint $r = \text{norm}\{x_1, \ldots, x_n\}$ states that the resultant variable r should be equal to the vector norm of the argument vector $x_1, \ldots, x_n$.

```
GRBGenConstr  AddGenConstrNorm (  GRBVar    resvar,
                                  GRBVar[]  vars,
                                  double    which,
                                  string    name )
```

Arguments:

resvar: The resultant variable of the new constraint.

vars: Array of variables that are the operands of the new constraint. Note that this array may not contain duplicates.

which: Which norm to use. Options are 0, 1, 2, and GRB.INFINITY.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.AddGenConstrIndicator()

Add a new general constraint of type `GRB.GENCONSTR_INDICATOR` to a model.

An INDICATOR constraint $z = f \rightarrow a^T x \leq b$ states that if the binary indicator variable z is equal to f, where $f \in \{0, 1\}$, then the linear constraint $a^T x \leq b$ should hold. On the other hand, if $z = 1 - f$, the linear constraint may be violated. The sense of the linear constraint can also be specified to be $=$ or $\geq$.

Note that the indicator variable z of a constraint will be forced to be binary, independent of how it was created.

Multiple signatures are available.

```
GRBGenConstr  AddGenConstrIndicator (  GRBVar      binvar,
                                       int         binval,
                                       GRBLinExpr  expr,
                                       char        sense,
                                       double      rhs,
                                       string      name )
```

Arguments:

binvar: The binary indicator variable.

binval: The value for the binary indicator variable that would force the linear constraint to be satisfied (0 or 1).

expr: Left-hand side expression for the linear constraint triggered by the indicator.

sense: Sense for the linear constraint. Options are `GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`.

rhs: Right-hand side value for the linear constraint.

name: Name for the new general constraint.

Return value:

New general constraint.

```
GRBGenConstr  AddGenConstrIndicator (  GRBVar         binvar,
                                       int            binval,
                                       GRBTempConstr  constr,
                                       string         name )
```

Arguments:

binvar: The binary indicator variable.

binval: The value for the binary indicator variable that would force the linear constraint to be satisfied (0 or 1).

constr: Temporary constraint object defining the linear constraint that is triggered by the indicator. The temporary constraint object is created using an overloaded comparison operator. See GRBTempConstr for more information.

name: Name for the new general constraint.

Return value:

New general constraint.

GRBModel.AddGenConstrPWL()

Add a new general constraint of type `GRB.GENCONSTR_PWL` to a model.

A piecewise-linear (PWL) constraint states that the relationship $y = f(x)$ must hold between variables x and y, where f is a piecewise-linear function. The breakpoints for f are provided as arguments. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined.

```
GRBGenConstr  AddGenConstrPWL (  GRBVar    xvar,
                                 GRBVar    yvar,
                                 double[]  xpts,
                                 double[]  ypts,
                                 string    name )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

xpts: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

ypts: The y values for the points that define the piecewise-linear function.

name: Name for the new general constraint.

Return value:
New general constraint.

GRBModel.AddGenConstrPoly()

A polynomial function constraint states that the relationship $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$ should hold between variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrPoly (  GRBVar    xvar,
                                  GRBVar    yvar,
                                  double[]  p,
                                  string    name,
                                  string    options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

p: The coefficients for the polynomial function (starting with the coefficient for the highest power).

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:
New general constraint.

GRBModel.AddGenConstrExp()

Add a new general constraint of type GRB.GENCONSTR_EXP to a model.

A natural exponential function constraint states that the relationship $y = exp(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrExp (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 string  name,
                                 string  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddGenConstrExpA()

Add a new general constraint of type `GRB.GENCONSTR_EXPA` to a model.

An exponential function constraint states that the relationship $y = a^x$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrExpA (  GRBVar  xvar,
                                  GRBVar  yvar,
                                  double  a,
                                  string  name,
                                  string  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

a: The base of the function, $a > 0$.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddGenConstrLog()

Add a new general constraint of type `GRB.GENCONSTR_LOG` to a model.

A natural logarithmic function constraint states that the relationship $y = log(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrLog (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 string  name,
                                 string  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddGenConstrLogA()

Add a new general constraint of type `GRB.GENCONSTR_LOGA` to a model.

A logarithmic function constraint states that the relationship $y = log_a(x)$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrLogA (  GRBVar  xvar,
                                  GRBVar  yvar,
                                  double  a,
                                  string  name,
                                  string  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

a: The base of the function, $a > 0$.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the

attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddGenConstrLogistic()

Add a new general constraint of type `GRB.GENCONSTR_LOGISTIC` to a model.

A logistic function constraint states that the relationship $y = \frac{1}{1+e^{-x}}$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrLogistic (  GRBVar  xvar,
                                      GRBVar  yvar,
                                      string  name,
                                      string  options )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

`name`: Name for the new general constraint.

`options`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddGenConstrPow()

Add a new general constraint of type `GRB.GENCONSTR_POW` to a model.

A power function constraint states that the relationship $y = x^a$ should hold for variables x and y, where a is the (constant) exponent.

If the exponent a is negative, the lower bound on x must be strictly positive. If the exponent isn't an integer, the lower bound on x must be non-negative.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrPow (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 double  a,
                                 string  name,
                                 string  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

a: The exponent of the function.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddGenConstrSin()

Add a new general constraint of type `GRB.GENCONSTR_SIN` to a model.

A sine function constraint states that the relationship $y = sin(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrSin (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 string  name,
                                 string  options )
```

Arguments:

xvar: The x variable.

yvar: The y variable.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddGenConstrCos()

Add a new general constraint of type `GRB.GENCONSTR_COS` to a model.

A cosine function constraint states that the relationship $y = cos(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrCos (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 string  name,
                                 string  options )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

`name`: Name for the new general constraint.

`options`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddGenConstrTan()

Add a new general constraint of type `GRB.GENCONSTR_TAN` to a model.

A tangent function constraint states that the relationship $y = tan(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

```
GRBGenConstr  AddGenConstrTan (  GRBVar  xvar,
                                 GRBVar  yvar,
                                 string  name,
                                 string  options )
```

Arguments:

`xvar`: The x variable.

`yvar`: The y variable.

name: Name for the new general constraint.

options: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

GRBModel.AddQConstr()

Add a quadratic constraint to a model. Multiple signatures are available.

Important note: Gurobi can handle both convex and non-convex quadratic constraints. The differences between them can be both important and subtle. Refer to this discussion for additional information.

```
GRBQConstr  AddQConstr (  GRBQuadExpr  lhsExpr,
                          char         sense,
                          GRBQuadExpr  rhsExpr,
                          string       name )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new quadratic constraint.

sense: Sense for new quadratic constraint (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

rhsExpr: Right-hand side expression for new quadratic constraint.

name: Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  AddQConstr (  GRBTempConstr  tempConstr,
                          string         name )
```

Add a quadratic constraint to a model.

Arguments:

tempConstr: Temporary constraint object, created by an overloaded comparison operator. See GRBTempConstr for more information.

name: Name for new constraint.

Return value:

New quadratic constraint object.

GRBModel.AddRange()

Add a single range constraint to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution.

Note that range constraints are stored internally as equality constraints. We add an extra variable to the model to capture the range information. Thus, the Sense attribute on a range constraint will always be `GRB.EQUAL`. In particular introducing a range constraint

$$L \leq a^T x \leq U$$

is equivalent to adding a slack variable s and the following constraints

$$\begin{aligned} a^T x - s &= L \\ 0 \leq s &\leq U - L. \end{aligned}$$

```
GRBConstr  AddRange (  GRBLinExpr  expr,
                       double      lower,
                       double      upper,
                       string      name )
```

Arguments:

`expr`: Linear expression for new range constraint.

`lower`: Lower bound for linear expression.

`upper`: Upper bound for linear expression.

`name`: Name for new constraint.

Return value:

New constraint object.

GRBModel.AddRanges()

Add new range constraints to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution.

```
GRBConstr[]  AddRanges (  GRBLinExpr[]  exprs,
                          double[]      lower,
                          double[]      upper,
                          string[]      names )
```

Arguments:

`exprs`: Linear expressions for the new range constraints.

`lower`: Lower bounds for linear expressions.

`upper`: Upper bounds for linear expressions.

`name`: Names for new range constraints.

`count`: Number of range constraints to add.

Return value:

Array of new constraint objects.

GRBModel.AddSOS()

Add an SOS constraint to the model. Please refer to this section for details on SOS constraints.

```
GRBSOS  AddSOS (  GRBVar[]  vars,
                  double[]  weights,
                  int       type )
```

Arguments:

vars: Array of variables that participate in the SOS constraint.

weights: Weights for the variables in the SOS constraint.

type: SOS type (can be GRB.SOS_TYPE1 or GRB.SOS_TYPE2).

Return value:

New SOS constraint.

GRBModel.AddVar()

Add a single decision variable to a model.

```
GRBVar  AddVar (  double  lb,
                  double  ub,
                  double  obj,
                  char    type,
                  string  name )
```

Add a variable to a model; non-zero entries will be added later.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB.CONTINUOUS, GRB.BINARY, GRB.INTEGER, GRB.SEMICONT, or GRB.SEMIINT).

name: Name for new variable.

Return value:

New variable object.

```
GRBVar  AddVar (  double       lb,
                  double       ub,
                  double       obj,
                  char         type,
                  GRBConstr[]  constrs,
                  double[]     coeffs,
                  string       name )
```

Add a variable to a model, and the associated non-zero coefficients.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB.CONTINUOUS, GRB.BINARY, GRB.INTEGER, GRB.SEMICONT, or GRB.SEMIINT).

constrs: Array of constraints in which the variable participates.

coeffs: Array of coefficients for each constraint in which the variable participates. The lengths of the **constrs** and **coeffs** arrays must be identical.

name: Name for new variable.

Return value:

New variable object.

```
GRBVar  AddVar (  double     lb,
                  double     ub,
                  double     obj,
                  char       type,
                  GRBColumn  col,
                  string     name )
```

Add a variable to a model. This signature allows you to specify the set of constraints to which the new variable belongs using a GRBColumn object.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB.CONTINUOUS, GRB.BINARY, GRB.INTEGER, GRB.SEMICONT, or GRB.SEMIINT).

col: GRBColumn object for specifying a set of constraints to which new variable belongs.

name: Name for new variable.

Return value:

New variable object.

GRBModel.AddVars()

Add new decision variables to a model.

```
GRBVar[]  AddVars (  int   count,
                     char  type )
```

Add **count** new decision variables to a model. All associated attributes take their default values, except the variable **type**, which is specified as an argument.

Arguments:

count: Number of variables to add.

type: Variable type for new variables (GRB.CONTINUOUS, GRB.BINARY, GRB.INTEGER, GRB.SEMICONT, or GRB.SEMIINT).

Return value:

Array of new variable objects.

```
GRBVar[]  AddVars (  double[]  lb,
                     double[]  ub,
                     double[]  obj,
                     char[]    type,
                     string[]  names )
```

Add new decision variables to a model. The number of added variables is determined by the length of the input arrays (which must be consistent across all arguments).

Arguments:

lb: Lower bounds for new variables. Can be `null`, in which case the variables get lower bounds of 0.0.

ub: Upper bounds for new variables. Can be `null`, in which case the variables get infinite upper bounds.

obj: Objective coefficients for new variables. Can be `null`, in which case the variables get objective coefficients of 0.0.

type: Variable types for new variables (`GRB.CONTINUOUS`, `GRB.BINARY`, `GRB.INTEGER`, `GRB.SEMICONT`, or `GRB.SEMIINT`). Can be `null`, in which case the variables are assumed to be continuous.

names: Names for new variables. Can be `null`, in which case all variables are given default names.

Return value:

Array of new variable objects.

```
GRBVar[]  AddVars (  double[]  lb,
                     double[]  ub,
                     double[]  obj,
                     char[]    type,
                     string[]  names,
                     int       start,
                     int       len )
```

Add new decision variables to a model. This signature allows you to use arrays to hold the various variable attributes (lower bound, upper bound, etc.), without forcing you to add a variable for each entry in the array. The `start` and `len` arguments allow you to specify which variables to add.

Arguments:

lb: Lower bounds for new variables. Can be `null`, in which case the variables get lower bounds of 0.0.

ub: Upper bounds for new variables. Can be `null`, in which case the variables get infinite upper bounds.

obj: Objective coefficients for new variables. Can be `null`, in which case the variables get objective coefficients of 0.0.

type: Variable types for new variables (`GRB.CONTINUOUS`, `GRB.BINARY`, `GRB.INTEGER`, `GRB.SEMICONT`, or `GRB.SEMIINT`). Can be `null`, in which case the variables are assumed to be continuous.

names: Names for new variables. Can be `null`, in which case all variables are given default names.

start: The first variable in the list to add.

len: The number of variables to add.

Return value:

Array of new variable objects.

```
GRBVar[]  AddVars (  double[]     lb,
                     double[]     ub,
                     double[]     obj,
                     char[]       type,
                     string[]     names,
                     GRBColumn[]  col )
```

Add new decision variables to a model. This signature allows you to specify the list of constraints to which each new variable belongs using an array of GRBColumn objects.

Arguments:

lb: Lower bounds for new variables. Can be `null`, in which case the variables get lower bounds of 0.0.

ub: Upper bounds for new variables. Can be `null`, in which case the variables get infinite upper bounds.

obj: Objective coefficients for new variables. Can be `null`, in which case the variables get objective coefficients of 0.0.

type: Variable types for new variables (`GRB.CONTINUOUS`, `GRB.BINARY`, `GRB.INTEGER`, `GRB.SEMICONT`, or `GRB.SEMIINT`). Can be `null`, in which case the variables are assumed to be continuous.

names: Names for new variables. Can be `null`, in which case all variables are given default names.

cols: GRBColumn objects for specifying a set of constraints to which each new column belongs.

Return value:

Array of new variable objects.

GRBModel.ChgCoeff()

Change one coefficient in the model. The desired change is captured using a GRBVar object, a GRBConstr object, and a desired coefficient for the specified variable in the specified constraint. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.Update), optimize the model (using GRBModel.Optimize), or write the model to disk (using GRBModel.Write).

```
void  ChgCoeff (  GRBConstr  constr,
                  GRBVar     var,
                  double     newvalue )
```

Arguments:

constr: Constraint for coefficient to be changed.

var: Variable for coefficient to be changed.

newvalue: Desired new value for coefficient.

GRBModel.ChgCoeffs()

Change a list of coefficients in the model. Each desired change is captured using a `GRBVar` object, a `GRBConstr` object, and a desired coefficient for the specified variable in the specified constraint. The entries in the input arrays each correspond to a single desired coefficient change. The lengths of the input arrays must all be the same. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.Update), optimize the model (using GRBModel.Optimize), or write the model to disk (using GRBModel.Write).

```
void  ChgCoeffs (  GRBConstr[]  constrs,
                   GRBVar[]     vars,
                   double[]     vals )
```

Arguments:

constrs: Constraints for coefficients to be changed.

vars: Variables for coefficients to be changed.

vals: Desired new values for coefficients.

GRBModel.ComputeIIS()

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the smallest one; there may exist others with fewer constraints or bounds.

IIS results are returned in a number of attributes: IISConstr, IISLB, IISUB, IISSOS, IISQConstr, and IISGenConstr. Each indicates whether the corresponding model element is a member of the computed IIS.

Note that for models with general function constraints, piecewise-linear approximation of the constraints may cause unreliable IIS results.

The IIS log provides information about the progress of the algorithm, including a guess at the eventual IIS size.

If an IIS computation is interrupted before completion, Gurobi will return the smallest infeasible subsystem found to that point.

The IISConstrForce, IISLBForce, IISUBForce, IISSOSForce, IISQConstrForce, and IISGenConstrForce attributes allow you mark model elements to either include or exclude from the computed IIS. Setting the attribute to 1 forces the corresponding element into the IIS, setting it to 0 forces it out of the IIS, and setting it to -1 allows the algorithm to decide.

To give an example of when these attributes might be useful, consider the case where an initial model is known to be feasible, but it becomes infeasible after adding constraints or tightening bounds. If you are only interested in knowing which of the changes caused the infeasibility, you can

force the unmodified bounds and constraints into the IIS. That allows the IIS algorithm to focus exclusively on the new constraints, which will often be substantially faster.

Note that setting any of the `Force` attributes to 0 may make the resulting subsystem feasible, which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

This method populates the IISConstr, IISQConstr, and IISGenConstr constraint attributes, the IISSOS, SOS attribute, and the IISLB and IISUB variable attributes. You can also obtain information about the results of the IIS computation by writing a `.ilp` format file (see GRBModel.Write). This file contains only the IIS from the original model.

Use the IISMethod parameter to adjust the behavior of the IIS algorithm.

Note that this method can be used to compute IISs for both continuous and MIP models.

```
void  ComputeIIS ( )
```

GRBModel.DiscardConcurrentEnvs()

Discard concurrent environments for a model.

The concurrent environments created by GetConcurrentEnv will be used by every subsequent call to the concurrent optimizer until the concurrent environments are discarded.

```
void  DiscardConcurrentEnvs ( )
```

GRBModel.DiscardMultiobjEnvs()

Discard all multi-objective environments associated with the model, thus restoring multi objective optimization to its default behavior.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Use GetMultiobjEnv to create a multi-objective environment.

```
void  DiscardMultiobjEnvs ( )
```

GRBModel.Dispose()

Release the resources associated with a `GRBModel` object. While the .NET garbage collector will eventually reclaim these resources, we recommend that you call the `Dispose` method when you are done using a model.

You should not attempt to use a `GRBModel` object after calling `Dispose` on it.

```
void  Dispose ( )
```

GRBModel.FeasRelax()

Modifies the `GRBModel` object to create a feasibility relaxation. Note that you need to call Optimize on the result to compute the actual relaxed solution.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This method provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost per unit violation in the lower bounds, upper bounds, and linear constraints, respectively.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the coefficients on the squares of the lower bound, upper bound, and linear constraint violations, respectively.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the weighted count of bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost of violating a lower bound, upper bound, and linear constraint, respectively.

To give an example, if a constraint with `rhspen` value `p` is violated by 2.0, it would contribute `2*p` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2*2*p` for `relaxobjtype=1`, it would contribute `p` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=false`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=true`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `FeasRelax` must solve an optimization problem to find the minimum possible relaxation when `minrelax=true`, which can be quite expensive.

There are two signatures for this method. The more complex one takes a list of variables and constraints, as well as penalties associated with relaxing the corresponding lower bounds, upper bounds, and constraints. If a variable or constraint is not included in one of these lists, the associated bounds or constraints may not be violated. The simpler signature takes a pair of boolean arguments, `vrelax` and `crelax`, that indicate whether variable bounds and/or constraints can be violated. If `vrelax`/`crelax` is `true`, then every bound/constraint is allowed to be violated, respectively, and the associated cost is 1.0.

Note that this is a destructive method: it modifies the model on which it is invoked. If you don't want to modify your original model, use the GRBModel constructor to create a copy before invoking this method.

```
double  FeasRelax (  int          relaxobjtype,
                     boolean      minrelax,
                     GRBVar[]     vars,
                     double[]     lbpen,
                     double[]     ubpen,
                     GRBConstr[]  constrs,
                     double[]     rhspen )
```

Create a feasibility relaxation model.

Arguments:

`relaxobjtype`: The cost function used when finding the minimum cost relaxation.

`minrelax`: The type of feasibility relaxation to perform.

`vars`: Variables whose bounds are allowed to be violated.

`lbpen`: Penalty for violating a variable lower bound. One entry for each variable in argument `vars`.

ubpen: Penalty for violating a variable upper bound. One entry for each variable in argument `vars`.

constrs: Linear constraints that are allowed to be violated.

rhspen: Penalty for violating a linear constraint. One entry for each constraint in argument `constrs`.

Arguments:

Return value:

Zero if `minrelax` is false. If `minrelax` is true, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

```
double  FeasRelax (  int      relaxobjtype,
                     boolean  minrelax,
                     boolean  vrelax,
                     boolean  crelax )
```

Simplified method for creating a feasibility relaxation model.

Arguments:

relaxobjtype: The cost function used when finding the minimum cost relaxation.

minrelax: The type of feasibility relaxation to perform.

vrelax: Indicates whether variable bounds can be relaxed (with a cost of 1.0 for any violations.

crelax: Indicates whether linear constraints can be relaxed (with a cost of 1.0 for any violations.

Return value:

Zero if `minrelax` is false. If `minrelax` is true, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

GRBModel.FixedModel()

Create the fixed model associated with a MIP model. The MIP model must have a solution loaded (e.g., after a call to the Optimize method). In the fixed model, each integer variable is fixed to the value that variable takes in the MIP solution. In addition, continuous variables may be fixed to satisfy SOS or general constraints. The result is a model without any integrality constraints, SOS constraints, or general constraints.

Note that, while the fixed problem is always a continuous model, it may contain a non-convex quadratic objective or non-convex quadratic constraints. As a result, it may still be solved using the MIP algorithm.

```
GRBModel  FixedModel ( )
```

Return value:

Fixed model associated with calling object.

GRBModel.Get()

Query the value(s) of a parameter or attribute. Use this method for parameters, for scalar model attributes, and for arrays of constraint or variable attributes.

```
double  Get (  GRB.DoubleParam  param )
```

Query the value of a double-valued parameter.

Arguments:

param: The parameter being queried.

Return value:

The current value of the requested parameter.

```
int  Get (  GRB.IntParam  param )
```

Query the value of an int-valued parameter.

Arguments:

param: The parameter being queried.

Return value:

The current value of the requested parameter.

```
string  Get (  GRB.StringParam  param )
```

Query the value of a string-valued parameter.

Arguments:

param: The parameter being queried.

Return value:

The current value of the requested parameter.

```
char[]  Get (  GRB.CharAttr  attr,
               GRBVar[]      vars )
```

Query a char-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: The variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
char[]  Get (  GRB.CharAttr  attr,
               GRBVar[]      vars,
               int           start,
               int           len )
```

Query a char-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being queried.

vars: A one-dimensional array of variables whose attribute values are being queried.

start: The index of the first variable of interest in the list.

len: The number of variables.

Return value:

The current values of the requested attribute for each input variable.

```
char[,]  Get (  GRB.CharAttr  attr,
                GRBVar[,]     vars )
```
Query a char-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A two-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
char[„]  Get (  GRB.CharAttr  attr,
                GRBVar[„]     vars )
```
Query a char-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A three-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
char[]  Get (  GRB.CharAttr  attr,
               GRBConstr[]   constrs )
```
Query a char-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: The constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
char[]  Get (  GRB.CharAttr  attr,
               GRBConstr[]   constrs,
               int           start,
               int           len )
```
Query a char-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A one-dimensional array of constraints whose attribute values are being queried.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

Return value:

The current values of the requested attribute for each input constraint.

```
char[,]  Get (  GRB.CharAttr  attr,
                GRBConstr[,]  constrs )
```
Query a char-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A two-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
char[,]  Get (  GRB.CharAttr  attr,
                GRBConstr[,]  constrs )
```

Query a char-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A three-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
char[]  Get (  GRB.CharAttr  attr,
               GRBQConstr[]  qconstrs )
```

Query a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: The quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
char[]  Get (  GRB.CharAttr  attr,
               GRBQConstr[]  qconstrs,
               int           start,
               int           len )
```

Query a char-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being queried.

start: The index of the first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
char[,]  Get (  GRB.CharAttr   attr,
                GRBQConstr[,]  qconstrs )
```

Query a char-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
char[,]  Get (  GRB.CharAttr   attr,
                GRBQConstr[,]  qconstrs )
```

Query a char-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
double  Get (  GRB.DoubleAttr  attr )
```

Query the value of a double-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
double[]  Get (  GRB.DoubleAttr  attr,
                 GRBVar[]        vars )
```

Query a double-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: The variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
double[]  Get (  GRB.DoubleAttr  attr,
                 GRBVar[]        vars,
                 int             start,
                 int             len )
```

Query a double-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being queried.

vars: A one-dimensional array of variables whose attribute values are being queried.

start: The index of the first variable of interest in the list.

len: The number of variables.

Return value:

The current values of the requested attribute for each input variable.

```
double[,]  Get (  GRB.DoubleAttr  attr,
                  GRBVar[,]       vars )
```

Query a double-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A two-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
double[„]  Get (  GRB.DoubleAttr  attr,
                  GRBVar[„]       vars )
```

Query a double-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A three-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
double[]  Get (  GRB.DoubleAttr  attr,
                 GRBConstr[]     constrs )
```

Query a double-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: The constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
double[]  Get (  GRB.DoubleAttr  attr,
                 GRBConstr[]     constrs,
                 int             start,
                 int             len )
```

Query a double-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A one-dimensional array of constraints whose attribute values are being queried.

start: The first constraint of interest in the list.

len: The number of constraints.

Return value:

The current values of the requested attribute for each input constraint.

```
double[,]  Get (  GRB.DoubleAttr  attr,
                  GRBConstr[,]    constrs )
```

Query a double-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A two-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
double[,]  Get ( GRB.DoubleAttr  attr,
                 GRBConstr[,]    constrs )
```

Query a double-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A three-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
double[]  Get ( GRB.DoubleAttr  attr,
                GRBQConstr[]    qconstrs )
```

Query a double-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: The quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
double[]  Get ( GRB.DoubleAttr  attr,
                GRBQConstr[]    qconstrs,
                int             start,
                int             len )
```

Query a double-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being queried.

start: The first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
double[,]  Get ( GRB.DoubleAttr  attr,
                 GRBQConstr[,]   qconstrs )
```

Query a double-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
double[,]  Get ( GRB.DoubleAttr  attr,
                 GRBQConstr[,]   qconstrs )
```

Query a double-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
int  Get ( GRB.IntAttr  attr )
```

Query the value of an int-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
int[]  Get ( GRB.IntAttr  attr,
             GRBVar[]     vars )
```

Query an int-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: The variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
int[]  Get ( GRB.IntAttr  attr,
             GRBVar[]     vars,
             int          start,
             int          len )
```

Query an int-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being queried.

vars: A one-dimensional array of variables whose attribute values are being queried.

start: The index of the first variable of interest in the list.

len: The number of variables.

Return value:

The current values of the requested attribute for each input variable.

```
int[,]  Get (  GRB.IntAttr  attr,
               GRBVar[,]    vars )
```

Query an int-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A two-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
int[„]  Get (  GRB.IntAttr  attr,
               GRBVar[„]    vars )
```

Query an int-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A three-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
int[]  Get (  GRB.IntAttr  attr,
              GRBConstr[]  constrs )
```

Query an int-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: The constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
int[]  Get (  GRB.IntAttr  attr,
              GRBConstr[]  constrs,
              int          start,
              int          len )
```

Query an int-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A one-dimensional array of constraints whose attribute values are being queried.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

Return value:

The current values of the requested attribute for each input constraint.

```
int[,]  Get (  GRB.IntAttr    attr,
               GRBConstr[,]   constrs )
```

Query an int-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A two-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
int[,,]  Get (  GRB.IntAttr    attr,
                GRBConstr[,,]  constrs )
```

Query an int-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A three-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
string  Get (  GRB.StringAttr  attr )
```

Query the value of a string-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
string[]  Get (  GRB.StringAttr  attr,
                 GRBVar[]        vars )
```

Query a string-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: The variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
string[]  Get (  GRB.StringAttr  attr,
                 GRBVar[]        vars,
                 int             start,
                 int             len )
```

Query a string-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being queried.

vars: A one-dimensional array of variables whose attribute values are being queried.

start: The index of the first variable of interest in the list.

len: The number of variables.

Return value:

The current values of the requested attribute for each input variable.

```
string[,]  Get (  GRB.StringAttr  attr,
                  GRBVar[,]       vars )
```

Query a string-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A two-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
string[,,]  Get (  GRB.StringAttr  attr,
                   GRBVar[,,]      vars )
```

Query a string-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being queried.

vars: A three-dimensional array of variables whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input variable.

```
string[]  Get (  GRB.StringAttr  attr,
                 GRBConstr[]     constrs )
```

Query a string-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: The constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
string[]  Get (  GRB.StringAttr  attr,
                 GRBConstr[]     constrs,
                 int             start,
                 int             len )
```

Query a string-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A one-dimensional array of constraints whose attribute values are being queried.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

Return value:

The current values of the requested attribute for each input constraint.

```
string[,]  Get (  GRB.StringAttr  attr,
                  GRBConstr[,]    constrs )
```

Query a string-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A two-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
string[,,]  Get (  GRB.StringAttr  attr,
                   GRBConstr[,,]   constrs )
```

Query a string-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being queried.

constrs: A three-dimensional array of constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input constraint.

```
string[]  Get (  GRB.StringAttr  attr,
                 GRBQConstr[]    qconstrs )
```

Query a string-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: The quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
string[]  Get (  GRB.StringAttr  attr,
                 GRBQConstr[]    qconstrs,
                 int             start,
                 int             len )
```

Query a string-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being queried.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being queried.

start: The index of the first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
string[,]  Get (  GRB.StringAttr  attr,
                  GRBQConstr[,]   qconstrs )
```

Query a string-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

`attr`: The attribute being queried.

`qconstrs`: A two-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

```
string[,]  Get ( GRB.StringAttr  attr,
                 GRBQConstr[,]   qconstrs )
```

Query a string-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

`attr`: The attribute being queried.

`qconstrs`: A three-dimensional array of quadratic constraints whose attribute values are being queried.

Return value:

The current values of the requested attribute for each input quadratic constraint.

GRBModel.GetCoeff()

Query the coefficient of variable `var` in linear constraint `constr` (note that the result can be zero).

```
double  GetCoeff ( GRBConstr  constr,
                   GRBVar     var )
```

Arguments:

`constr`: The requested constraint.

`var`: The requested variable.

Return value:

The current value of the requested coefficient.

GRBModel.GetCol()

Retrieve the list of constraints in which a variable participates, and the associated coefficients. The result is returned as a GRBColumn object.

```
GRBColumn  GetCol ( GRBVar  var )
```

Arguments:

`var`: The variable of interest.

Return value:

A GRBColumn object that captures the set of constraints in which the variable participates.

GRBModel.GetConcurrentEnv()

Create/retrieve a concurrent environment for a model.

This method provides fine-grained control over the concurrent optimizer. By creating your own concurrent environments and setting appropriate parameters on these environments (e.g., the Method parameter), you can control exactly which strategies the concurrent optimizer employs. For example, if you create two concurrent environments, and set `Method` to primal simplex for one and dual simplex for the other, subsequent concurrent optimizer runs will use the two simplex algorithms rather than the default choices.

Note that you must create contiguously numbered concurrent environments, starting with `num=0`. For example, if you want three concurrent environments, they must be numbered 0, 1, and 2.

Once you create concurrent environments, they will be used for every subsequent concurrent optimization on that model. Use DiscardConcurrentEnvs to revert back to default concurrent optimizer behavior.

```
GRBEnv  GetConcurrentEnv (  int  num )
```

Arguments:

num: The concurrent environment number.

Return value:

The concurrent environment for the model.

GRBModel.GetConstrByName()

Retrieve a linear constraint from its name. If multiple linear constraints have the same name, this method chooses one arbitrarily.

```
GRBConstr  GetConstrByName (  string  name )
```

Arguments:

name: The name of the desired linear constraint.

Return value:

The requested linear constraint.

GRBModel.GetConstrs()

Retrieve an array of all linear constraints in the model.

```
GRBConstr[]  GetConstrs ( )
```

Return value:

All linear constraints in the model.

GRBModel.GetGenConstrXxx()

The following methods allow you to retrieve general constraints from your model.

GRBModel.GetGenConstrMax

Retrieve the data associated with a general constraint of type MAX. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrMax for a description of the semantics of this general constraint type.

```
void  GetGenConstrMax (  GRBGenConstr  genc,
                         out GRBVar    resvar,
                         out GRBVar[]  vars,
                         out double    constant )
```

Arguments:

genc: The general constraint object.

resvar: Stores the resultant variable of the constraint.

vars: Stores the array of operand variables of the constraint.

constant: Stores the additional constant operand of the constraint.

GRBModel.GetGenConstrMin

Retrieve the data associated with a general constraint of type MIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrMin for a description of the semantics of this general constraint type.

```
void  GetGenConstrMin (  GRBGenConstr  genc,
                         out GRBVar    resvar,
                         out GRBVar[]  vars,
                         out double    constant )
```

Arguments:

genc: The general constraint object.

resvar: Stores the resultant variable of the constraint.

vars: Stores the array of operand variables of the constraint.

constant: Stores the additional constant operand of the constraint.

GRBModel.GetGenConstrAbs

Retrieve the data associated with a general constraint of type ABS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrAbs for a description of the semantics of this general constraint type.

```
void  GetGenConstrAbs (  GRBGenConstr  genc,
                         out GRBVar    resvar,
                         out GRBVar    argvar )
```

Arguments:

genc: The general constraint object.
resvar: Stores the resultant variable of the constraint.
argvar: Stores the argument variable of the constraint.

GRBModel.GetGenConstrAnd

Retrieve the data associated with a general constraint of type AND. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrAnd for a description of the semantics of this general constraint type.

```
void  GetGenConstrAnd (  GRBGenConstr  genc,
                         out GRBVar    resvar,
                         out GRBVar[]  vars )
```

Arguments:

genc: The general constraint object.
resvar: Stores the resultant variable of the constraint.
vars: Stores the array of operand variables of the constraint.

GRBModel.GetGenConstrOr

Retrieve the data associated with a general constraint of type OR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrOr for a description of the semantics of this general constraint type.

```
void  GetGenConstrOr (  GRBGenConstr  genc,
                        out GRBVar    resvar,
                        out GRBVar[]  vars )
```

Arguments:

genc: The general constraint object.
resvar: Stores the resultant variable of the constraint.
vars: Stores the array of operand variables of the constraint.

GRBModel.GetGenConstrNorm

Retrieve the data associated with a general constraint of type NORM. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrNorm for a description of the semantics of this general constraint type.

```
void  GetGenConstrNorm (  GRBGenConstr  genc,
                          out GRBVar    resvar,
                          out GRBVar[]  vars,
                          out double    which )
```

Arguments:

genc: The general constraint object.

resvar: Stores the resultant variable of the constraint.

vars: Stores the array of operand variables of the constraint.

whichP: Stores the norm type (possible values are 0, 1, 2, or GRB.INFINITY).

GRBModel.GetGenConstrIndicator

Retrieve the data associated with a general constraint of type INDICATOR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrIndicator for a description of the semantics of this general constraint type.

```
void  GetGenConstrIndicator (  GRBGenConstr    genc,
                               out GRBVar      binvar,
                               out int         binval,
                               out GRBLinExpr  expr,
                               out char        sense,
                               out double      rhs )
```

Arguments:

genc: The general constraint object.

binvar: Stores the binary indicator variable of the constraint.

binval: Stores the value that the indicator variable has to take in order to trigger the linear constraint.

expr: Stores the left-hand side expression of the linear constraint that is triggered by the indicator.

sense: Stores the sense for the linear constraint. Options are `GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`.

rhs: Stores the right-hand side value for the linear constraint.

GRBModel.GetGenConstrPWL

Retrieve the data associated with a general constraint of type PWL. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `null` value for the `xpts` and `ypts` arguments. The routine returns the length for the `xpts` and `ypts` arrays in `npts`. That allows you to make certain that the `xpts` and `ypts` arrays are of sufficient size to hold the result of the second call.

See also AddGenConstrPWL for a description of the semantics of this general constraint type.

```
void  GetGenConstrPWL (  GRBGenConstr  genc,
                         out GRBVar    xvar,
                         out GRBVar    yvar,
                         out int       npts,
                         double[]      xpts,
                         double[]      ypts )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

npts: Store the number of points that define the piecewise-linear function.

xpts: The x values for the points that define the piecewise-linear function.

ypts: The y values for the points that define the piecewise-linear function.

GRBModel.GetGenConstrPoly

Retrieve the data associated with a general constraint of type POLY. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

Typical usage is to call this routine twice. In the first call, you specify the requested general constraint, with a `null` value for the `p` argument. The routine returns the length of the `p` array in `plen`. That allows you to make certain that the `p` array is of sufficient size to hold the result of the second call.

See also AddGenConstrPoly for a description of the semantics of this general constraint type.

```
void  GetGenConstrPoly (  GRBGenConstr  genc,
                          out GRBVar    xvar,
                          out GRBVar    yvar,
                          out int       plen,
                          double[]      p )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

plen: Store the array length for p. If x^d is the highest power term, then $d + 1$ will be returned.

p: The coefficients for polynomial function.

GRBModel.GetGenConstrExp

Retrieve the data associated with a general constraint of type EXP. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrExp for a description of the semantics of this general constraint type.

```
void  GetGenConstrExp (  GRBGenConstr  genc,
                         out GRBVar    xvar,
                         out GRBVar    yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.GetGenConstrExpA

Retrieve the data associated with a general constraint of type EXPA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrExpA for a description of the semantics of this general constraint type.

```
void  GetGenConstrExpA (  GRBGenConstr  genc,
                          out GRBVar    xvar,
                          out GRBVar    yvar,
                          out double    a )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

a: Store the base of the function.

GRBModel.GetGenConstrLog

Retrieve the data associated with a general constraint of type LOG. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrLog for a description of the semantics of this general constraint type.

```
void  GetGenConstrLog (  GRBGenConstr  genc,
                         out GRBVar    xvar,
                         out GRBVar    yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.GetGenConstrLogA

Retrieve the data associated with a general constraint of type LOGA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrLogA for a description of the semantics of this general constraint type.

```
void  GetGenConstrLogA (  GRBGenConstr  genc,
                          out GRBVar    xvar,
                          out GRBVar    yvar,
                          out double    a )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

a: Store the base of the function.

GRBModel.GetGenConstrLogistic

Retrieve the data associated with a general constraint of type LOGISTIC. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrLogistic for a description of the semantics of this general constraint type.

```
void  GetGenConstrLogistic (  GRBGenConstr  genc,
                              out GRBVar    xvar,
                              out GRBVar    yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.GetGenConstrPow

Retrieve the data associated with a general constraint of type POW. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrPow for a description of the semantics of this general constraint type.

```
void  GetGenConstrPow (  GRBGenConstr  genc,
                         out GRBVar    xvar,
                         out GRBVar    yvar,
                         out double    a )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

a: Store the power of the function.

GRBModel.GetGenConstrSin

Retrieve the data associated with a general constraint of type SIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrSin for a description of the semantics of this general constraint type.

```
void  GetGenConstrSin (  GRBGenConstr  genc,
                         out GRBVar    xvar,
                         out GRBVar    yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.GetGenConstrCos

Retrieve the data associated with a general constraint of type COS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrCos for a description of the semantics of this general constraint type.

```
void  GetGenConstrCos (  GRBGenConstr  genc,
                         out GRBVar    xvar,
                         out GRBVar    yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.GetGenConstrTan

Retrieve the data associated with a general constraint of type TAN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also AddGenConstrTan for a description of the semantics of this general constraint type.

```
void  GetGenConstrTan (  GRBGenConstr  genc,
                         out GRBVar    xvar,
                         out GRBVar    yvar )
```

Arguments:

genc: The general constraint object.

Any of the following arguments can be `null`.

xvar: Store the x variable.

yvar: Store the y variable.

GRBModel.GetGenConstrs()

Retrieve an array of all general constraints in the model.

```
GRBGenConstr[]  GetGenConstrs ( )
```

Return value:

All general constraints in the model.

GRBModel.GetJSONSolution()

After a call to Optimize, this method returns the resulting solution and related model attributes as a JSON string. Please refer to the JSON solution format section for details.

```
string  GetJSONSolution ( )
```

Return value:

A JSON string.

GRBModel.GetMultiobjEnv()

Create/retrieve a multi-objective environment for the optimization pass with the given index. This environment enables fine-grained control over the multi-objective optimization process. Specifically, by changing parameters on this environment, you modify the behavior of the optimization that occurs during the corresponding pass of the multi-objective optimization.

Each multi-objective environment starts with a copy of the current model environment.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Please refer to the discussion on Combining Blended and Hierarchical Objectives for information on the optimization passes to solve multi-objective models.

Use DiscardMultiobjEnvs to discard multi-objective environments and return to standard behavior.

```
GRBEnv  GetMultiobjEnv (  int  index )
```

Arguments:

index: The optimization pass index, starting from 0.

Return value:

The multi-objective environment for that optimization pass when solving the model.

GRBModel.GetObjective()

Retrieve the model objective(s).

```
GRBExpr  GetObjective ( )
```

Retrieve the optimization objective.

Note that the constant and linear portions of the objective can also be retrieved using the ObjCon and Obj attributes.

Return value:

The model objective.

```
GRBLinExpr  GetObjective (  int  index )
```

Retrieve an alternative optimization objective. Alternative objectives will always be linear. You can also use this routine to retrieve the primary objective (using `index` = 0), but you will get an exception if the primary objective contains quadratic terms.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

Note that alternative objectives can also be retrieved using the ObjNCon and ObjN attributes.

Arguments:

`index`: The index for the requested alternative objective.

Return value:

The requested alternative objective.

GRBModel.GetPWLObj()

Retrieve the piecewise-linear objective function for a variable. The return value gives the number of points that define the function, and the x and y arguments give the coordinates of the points, respectively. The x and y arguments must be large enough to hold the result. Call this method with `null` values for x and y if you just want the number of points.

Refer to this discussion for additional information on what the values in x and y mean.

```
int  GetPWLObj (  GRBVar    var,
                  double[]  x,
                  double[]  y )
```

Arguments:

`var`: The variable whose objective function is being retrieved.

`x`: The x values for the points that define the piecewise-linear function. These will always be in non-decreasing order.

`y`: The y values for the points that define the piecewise-linear function.

Return value:

The number of points that define the piecewise-linear objective function.

GRBModel.GetQConstr()

Retrieve the left-hand side expression from a quadratic constraint. The result is returned as a GRBQuadExpr object.

```
GRBQuadExpr  GetQConstr (  GRBQConstr  qconstr )
```

Arguments:

`qconstr`: The quadratic constraint of interest.

Return value:

A GRBQuadExpr object that captures the left-hand side of the quadratic constraint.

GRBModel.GetQConstrs()

Retrieve an array of all quadratic constraints in the model.

```
GRBQConstr[]  GetQConstrs ( )
```

Return value:

All quadratic constraints in the model.

GRBModel.GetQCRow()

Retrieve the left-hand side expression for a quadratic constraint. The result is returned as a GRBQuadExpr object.

```
GRBQuadExpr  GetQCRow (  GRBQConstr  qc )
```

Arguments:

`qc`: The quadratic constraint of interest.

Return value:

A GRBQuadExpr object that captures the left-hand side of the quadratic constraint.

GRBModel.GetRow()

Retrieve a list of variables that participate in a constraint, and the associated coefficients. The result is returned as a GRBLinExpr object.

```
GRBLinExpr  GetRow (  GRBConstr  constr )
```

Arguments:

`constr`: The constraint of interest. A GRBConstr object, typically obtained from AddConstr or GetConstrs.

Return value:

A GRBLinExpr object that captures the set of variables that participate in the constraint.

GRBModel.GetSOS()

Retrieve the list of variables that participate in an SOS constraint, and the associated coefficients. The return value is the length of this list. Note that the argument arrays must be long enough to accommodate the result. Call the method with `null` array arguments to determine the appropriate array lengths.

```
int  GetSOS (  GRBSOS    sos,
               GRBVar[]  vars,
               double[]  weights,
               int[]     type )
```

Arguments:

sos: The SOS set of interest.

vars: A list of variables that participate in **sos**. Can be **null**.

weights: The SOS weights for each participating variable. Can be **null**.

type: The type of the SOS set (either GRB.SOS_TYPE1 or GRB.SOS_TYPE2) is returned in type[0].

Return value:

The number of entries placed in the output arrays. Note that you should consult the return value to determine the length of the result; the arrays sizes won't necessarily match the result size.

GRBModel.GetSOSs()

Retrieve an array of all SOS constraints in the model.

```
GRBSOS[]  GetSOSs ( )
```

Return value:

All SOS constraints in the model.

GRBModel.GetTuneResult()

Use this method to retrieve the results of a previous Tune call. Calling this method with argument **n** causes tuned parameter set **n** to be copied into the model. Parameter sets are stored in order of decreasing quality, with parameter set 0 being the best. The number of available sets is stored in attribute TuneResultCount.

Once you have retrieved a tuning result, you can call optimize to use these parameter settings to optimize the model, or write to write the changed parameters to a .prm file.

Please refer to the parameter tuning section for details on the tuning tool.

```
void  GetTuneResult (  int  n )
```

n: The index of the tuning result to retrieve. The best result is available as index 0. The number of stored results is available in attribute TuneResultCount.

GRBModel.GetVarByName()

Retrieve a variable from its name. If multiple variable have the same name, this method chooses one arbitrarily.

```
GRBVar  GetVarByName (  string  name )
```

Arguments:

name: The name of the desired variable.

Return value:

The requested variable.

GRBModel.GetVars()

Retrieve an array of all variables in the model.

```
GRBVar[]  GetVars ( )
```

Return value:

All variables in the model.

GRBModel.Optimize()

Optimize the model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion, this method will populate the solution related attributes of the model. See the Attributes section for more information on attributes.

Please consult this section for a discussion of some of the practical issues associated with solving a precisely defined mathematical model using finite-precision floating-point arithmetic.

Note that this method will process all pending model modifications.

```
void  Optimize ( )
```

GRBModel.OptimizeAsync()

Optimize a model asynchronously. This routine returns immediately. Your program can perform other computations while optimization proceeds in the background. To check the state of the asynchronous optimization, query the Status attribute for the model. A value of `IN_PROGRESS` indicates that the optimization has not yet completed. When you are done with your foreground tasks, you must call Sync to sync your foreground program with the asynchronous optimization task.

Note that the set of Gurobi calls that you are allowed to make while optimization is running in the background is severely limited. Specifically, you can only perform attribute queries, and only for a few attributes (listed below). Any other calls on the running model, *or on any other models that were built within the same Gurobi environment*, will fail with error code `OPTIMIZATION_IN_PROGRESS`.

Note that there are no such restrictions on models built in other environments. Thus, for example, you could create multiple environments, and then have a single foreground program launch multiple simultaneous asynchronous optimizations, each in its own environment.

As already noted, you are allowed to query the value of the `Status` attribute while an asynchronous optimization is in progress. The other attributes that can be queried are: ObjVal, ObjBound, IterCount, NodeCount, and BarIterCount. In each case, the returned value reflects progress in the optimization to that point. Any attempt to query the value of an attribute not on this list will return an `OPTIMIZATION_IN_PROGRESS` error.

```
void  OptimizeAsync ( )
```

GRBModel.OptimizeBatch()

Submit a new batch request to the Cluster Manager. Returns the `BatchID` (a string), which uniquely identifies the job in the Cluster Manager and can be used to query the status of this request (from this program or from any other). Once the request has completed, the `BatchID` can also be used to retrieve the associated solution. To submit a batch request, you must tag at least

one element of the model by setting one of the VTag, CTag or QCTag attributes. For more details on batch optimization, please refer to the Batch Optimization section.

Note that this routine will process all pending model modifications.

```
string  OptimizeBatch ( )
```

Example usage:

```
// Submit batch request
batchID = model.OptimizeBatch();
```

GRBModel.Presolve()

Perform presolve on a model.

Please note that the presolved model computed by this function may be different from the presolved model computed when optimizing the model.

```
GRBModel  Presolve ( )
```

Return value:

Presolved version of original model.

GRBModel.Read()

This method is the general entry point for importing data from a file into a model. It can be used to read basis files for continuous models, start vectors for MIP models, or parameter settings. The type of data read is determined by the file suffix. File formats are described in the File Format section.

Note that reading a file does **not** process all pending model modifications. These modifications can be processed by calling GRBModel.Update.

Note also that this is **not** the method to use if you want to read a new model from a file. For that, use the GRBModel constructor. One variant of the constructor takes the name of the file that contains the new model as its argument.

```
void  Read ( string  filename )
```

Arguments:

`filename`: Name of the file to read. The suffix on the file must be either `.bas` (for an LP basis), `.mst` or `.sol` (for a MIP start), `.hnt` (for MIP hints), `.ord` (for a priority order), or `.prm` (for a parameter file). The suffix may optionally be followed by `.zip`, `.gz`, `.bz2`, or `.7z`.

GRBModel.Remove()

Remove a variable, constraint, or SOS from a model.

```
void  Remove ( GRBConstr  constr )
```

Remove a constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.Update), optimize the model (using GRBModel.Optimize), or write the model to disk (using GRBModel.Write).

Arguments:

`constr`: The constraint to remove.

```
void  Remove ( GRBGenConstr  genconstr )
```

Remove a general constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.Update), optimize the model (using GRBModel.Optimize), or write the model to disk (using GRBModel.Write).

Arguments:

genconstr: The general constraint to remove.

```
void  Remove ( GRBQConstr  qconstr )
```

Remove a quadratic constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.Update), optimize the model (using GRBModel.Optimize), or write the model to disk (using GRBModel.Write).

Arguments:

qconstr: The constraint to remove.

```
void  Remove ( GRBSOS  sos )
```

Remove an SOS constraint from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.Update), optimize the model (using GRBModel.Optimize), or write the model to disk (using GRBModel.Write).

Arguments:

sos: The SOS constraint to remove.

```
void  Remove ( GRBVar  var )
```

Remove a variable from the model. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using GRBModel.Update), optimize the model (using GRBModel.Optimize), or write the model to disk (using GRBModel.Write).

Arguments:

var: The variable to remove.

GRBModel.Reset()

Reset the model to an unsolved state, discarding any previously computed solution information.

```
void  Reset ( )
void  Reset ( int  clearall )
```

Arguments:

clearall: A value of 1 discards additional information that affects the solution process but not the actual model (currently MIP starts, variable hints, branching priorities, lazy flags, and partition information). Pass 0 to just discard the solution.

GRBModel.SetCallback()

Set the callback object for a model. The `Callback()` method on this object will be called periodically from the Gurobi solver. You will have the opportunity to obtain more detailed information about the state of the optimization from this callback. See the documentation for GRBCallback for additional information.

Note that a model can only have a single callback method, so this call will replace an existing callback. To disable a previously set callback, call this method with a `null` argument.

```
void  SetCallback ( GRBCallback cb )
```

GRBModel.Set()

Set the value(s) of a parameter or attribute. Use this method for parameters, for scalar model attributes, or for arrays of constraint or variable attributes.

```
void  Set ( GRB.DoubleParam  param,
            double           newvalue )
```

Set the value of a double-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv.Set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newvalue: The desired new value for the parameter.

```
void  Set ( GRB.IntParam  param,
            int           newvalue )
```

Set the value of an int-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv.Set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newvalue: The desired new value for the parameter.

```
void  Set ( GRB.StringParam  param,
            string           newvalue )
```

Set the value of a string-valued parameter.

The difference between setting a parameter on a model and setting it on an environment (i.e., through GRBEnv.Set) is that the former modifies the parameter for a single model, while the latter modifies the parameter for every model that is subsequently built using that environment (and leaves the parameter unchanged for models that were previously built using that environment).

Arguments:

param: The parameter being modified.

newvalue: The desired new value for the parameter.

```
void  Set (  GRB.CharAttr  attr,
             GRBVar[]      vars,
             char[]        newvalues )
```
Set a char-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: The variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.CharAttr  attr,
             GRBVar[]      vars,
             char[]        newvalues,
             int           start,
             int           len )
```
Set a char-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being modified.

vars: A one-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

start: The index of the first variable of interest in the list.

len: The number of variables.

```
void  Set (  GRB.CharAttr  attr,
             GRBVar[,]     vars,
             char[,]       newvalues )
```
Set a char-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A two-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.CharAttr  attr,
             GRBVar[,,]    vars,
             char[,,]      newvalues )
```
Set a char-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A three-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.CharAttr  attr,
             GRBConstr[]   constrs,
             char[]        newvalues )
```
Set a char-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: The constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set (  GRB.CharAttr  attr,
             GRBConstr[]   constrs,
             char[]        newvalues,
             int           start,
             int           len )
```

Set a char-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A one-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

```
void  Set (  GRB.CharAttr  attr,
             GRBConstr[,]  constrs,
             char[,]       newvalues )
```

Set a char-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A two-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set (  GRB.CharAttr  attr,
             GRBConstr[,,]  constrs,
             char[,,]       newvalues )
```

Set a char-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A three-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set (  GRB.CharAttr  attr,
             GRBQConstr[]  qconstrs,
             char[]        newvalues )
```

Set a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: The quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

```
void  Set (  GRB.CharAttr  attr,
             GRBQConstr[]  qconstrs,
             char[]        newvalues,
             int           start,
             int           len )
```

Set a char-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

start: The index of the first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

```
void  Set (  GRB.CharAttr   attr,
             GRBQConstr[,]  qconstrs,
             char[,]        newvalues )
```

Set a char-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

```
void  Set (  GRB.CharAttr   attr,
             GRBQConstr[,,]  qconstrs,
             char[,,]        newvalues )
```

Set a char-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

```
void  Set (  GRB.DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued model attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value for the attribute.

```
void  Set (  GRB.DoubleAttr  attr,
             GRBVar[]        vars,
             double[]        newvalues )
```

Set a double-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: The variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.DoubleAttr  attr,
             GRBVar[]        vars,
             double[]        newvalues,
             int             start,
             int             len )
```

Set a double-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being modified.

vars: A one-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

start: The index of the first variable of interest in the list.

len: The number of variables.

```
void  Set (  GRB.DoubleAttr  attr,
             GRBVar[,]       vars,
             double[,]       newvalues )
```

Set a double-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A two-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.DoubleAttr  attr,
             GRBVar[,,]      vars,
             double[,,]      newvalues )
```

Set a double-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A three-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set ( GRB.DoubleAttr  attr,
            GRBConstr[]     constrs,
            double[]        newvalues )
```

Set a double-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: The constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set ( GRB.DoubleAttr  attr,
            GRBConstr[]     constrs,
            double[]        newvalues,
            int             start,
            int             len )
```

Set a double-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A one-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

start: The first constraint of interest in the list.

len: The number of constraints.

```
void  Set ( GRB.DoubleAttr  attr,
            GRBConstr[,]    constrs,
            double[,]       newvalues )
```

Set a double-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A two-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set ( GRB.DoubleAttr  attr,
            GRBConstr[,,]   constrs,
            double[,,]      newvalues )
```

Set a double-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A three-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set ( GRB.DoubleAttr  attr,
            GRBQConstr[]    qconstrs,
            double[]        newvalues )
```

Set a double-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: The quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

```
void  Set (  GRB.DoubleAttr  attr,
             GRBQConstr[]    qconstrs,
             double[]        newvalues,
             int             start,
             int             len )
```

Set a double-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

start: The first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

```
void  Set (  GRB.DoubleAttr  attr,
             GRBQConstr[,]   qconstrs,
             double[,]       newvalues )
```

Set a double-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

```
void  Set (  GRB.DoubleAttr  attr,
             GRBQConstr[,,]  qconstrs,
             double[,,]      newvalues )
```

Set a double-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

```
void  Set (  GRB.IntAttr  attr,
             int          newvalue )
```

Set the value of an int-valued model attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value for the attribute.

```
void  Set (  GRB.IntAttr  attr,
             GRBVar[]     vars,
             int[]        newvalues )
```

Set an int-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: The variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.IntAttr  attr,
             GRBVar[]     vars,
             int[]        newvalues,
             int          start,
             int          len )
```

Set an int-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being modified.

vars: A one-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

start: The index of the first variable of interest in the list.

len: The number of variables.

```
void  Set (  GRB.IntAttr  attr,
             GRBVar[,]    vars,
             int[,]       newvalues )
```

Set an int-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A two-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.IntAttr  attr,
             GRBVar[,,]   vars,
             int[,,]      newvalues )
```

Set an int-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being modified.

vars: A three-dimensional array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

```
void  Set ( GRB.IntAttr  attr,
            GRBConstr[]  constrs,
            int[]        newvalues )
```
Set an int-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: The constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set ( GRB.IntAttr  attr,
            GRBConstr[]  constrs,
            int[]        newvalues,
            int          start,
            int          len )
```
Set an int-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A one-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

```
void  Set ( GRB.IntAttr   attr,
            GRBConstr[,]  constrs,
            int[,]        newvalues )
```
Set an int-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A two-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set ( GRB.IntAttr   attr,
            GRBConstr[,,]  constrs,
            int[,,]        newvalues )
```
Set an int-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A three-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set ( GRB.StringAttr  attr,
            string          newvalue )
```
Set the value of a string-valued model attribute.

Arguments:

attr: The attribute being modified.
newvalue: The desired new value for the attribute.

```
void  Set (  GRB.StringAttr  attr,
             GRBVar[]        vars,
             string[]        newvalues )
```

Set a string-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.
vars: The variables whose attribute values are being modified.
newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.StringAttr  attr,
             GRBVar[]        vars,
             string[]        newvalues,
             int             start,
             int             len )
```

Set a string-valued variable attribute for a sub-array of variables.

Arguments:

attr: The attribute being modified.
vars: A one-dimensional array of variables whose attribute values are being modified.
newvalues: The desired new values for the attribute for each input variable.
start: The index of the first variable of interest in the list.
len: The number of variables.

```
void  Set (  GRB.StringAttr  attr,
             GRBVar[,]       vars,
             string[,]       newvalues )
```

Set a string-valued variable attribute for a two-dimensional array of variables.

Arguments:

attr: The attribute being modified.
vars: A two-dimensional array of variables whose attribute values are being modified.
newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.StringAttr  attr,
             GRBVar[,,]      vars,
             string[,,]      newvalues )
```

Set a string-valued variable attribute for a three-dimensional array of variables.

Arguments:

attr: The attribute being modified.
vars: A three-dimensional array of variables whose attribute values are being modified.
newvalues: The desired new values for the attribute for each input variable.

```
void  Set (  GRB.StringAttr  attr,
             GRBConstr[]     constrs,
             string[]        newvalues )
```
Set a string-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: The constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set (  GRB.StringAttr  attr,
             GRBConstr[]     constrs,
             string[]        newvalues,
             int             start,
             int             len )
```
Set a string-valued constraint attribute for a sub-array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A one-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

start: The index of the first constraint of interest in the list.

len: The number of constraints.

```
void  Set (  GRB.StringAttr  attr,
             GRBConstr[,]    constrs,
             string[,]       newvalues )
```
Set a string-valued constraint attribute for a two-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A two-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set (  GRB.StringAttr  attr,
             GRBConstr[,,]   constrs,
             string[,,]      newvalues )
```
Set a string-valued constraint attribute for a three-dimensional array of constraints.

Arguments:

attr: The attribute being modified.

constrs: A three-dimensional array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

```
void  Set (  GRB.StringAttr  attr,
             GRBQConstr[]    qconstrs,
             string[]        newvalues )
```
Set a string-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: The quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

```
void  Set (  GRB.StringAttr  attr,
             GRBQConstr[]    qconstrs,
             string[]        newvalues,
             int             start,
             int             len )
```

Set a string-valued quadratic constraint attribute for a sub-array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A one-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

start: The index of the first quadratic constraint of interest in the list.

len: The number of quadratic constraints.

```
void  Set (  GRB.StringAttr  attr,
             GRBQConstr[,]   qconstrs,
             string[,]       newvalues )
```

Set a string-valued quadratic constraint attribute for a two-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A two-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

```
void  Set (  GRB.StringAttr  attr,
             GRBQConstr[,,]  qconstrs,
             string[,,]      newvalues )
```

Set a string-valued quadratic constraint attribute for a three-dimensional array of quadratic constraints.

Arguments:

attr: The attribute being modified.

qconstrs: A three-dimensional array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

GRBModel.SetObjective()

Set the model objective equal to a linear or quadratic expression (for multi-objective optimization, see SetObjectiveN).

Note that you can also modify the linear portion of a model objective using the Obj variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `Obj` attribute can be used to modify individual linear terms.

```
void  SetObjective ( GRBExpr  expr,
                     int      sense )
```

Set the model objective, and the objective sense (`GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization).

Arguments:

`expr`: New model objective.

`sense`: New optimization sense (`GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization).

```
void  SetObjective ( GRBExpr  expr )
```

Set the model objective. The sense of the objective is determined by the value of the ModelSense attribute.

Arguments:

`expr`: New model objective.

GRBModel.SetObjectiveN()

```
void  SetObjectiveN ( GRBLinExpr  expr,
                      int         index,
                      int         priority,
                      double      weight,
                      double      abstol,
                      double      reltol,
                      string      name )
```

Set an alternative optimization objective equal to a linear expression.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

Note that you can also modify an alternative objective using the ObjN variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `ObjN` attribute can be used to modify individual terms.

Arguments:

`expr`: New alternative objective.

`index`: Index for new objective. If you use an index of 0, this routine will change the primary optimization objective.

`priority`: Priority for the alternative objective. This initializes the ObjNPriority attribute for this objective.

weight: Weight for the alternative objective. This initializes the ObjNWeight attribute for this objective.

abstol: Absolute tolerance for the alternative objective. This initializes the ObjNAbsTol attribute for this objective.

reltol: Relative tolerance for the alternative objective. This initializes the ObjNRelTol attribute for this objective.

name: Name of the alternative objective. This initializes the ObjNName attribute for this objective.

GRBModel.SetPWLObj()

Set a piecewise-linear objective function for a variable.

The arguments to this method specify a list of points that define a piecewise-linear objective function for a single variable. Specifically, the x and y arguments give coordinates for the vertices of the function.

For additional details on piecewise-linear objective functions, refer to this discussion.

```
void  SetPWLObj ( GRBvar    var,
                  double[]  x,
                  double[]  y )
```

Set the piecewise-linear objective function for a variable.

Arguments:

var: The variable whose objective function is being set.

x: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

y: The y values for the points that define the piecewise-linear function.

GRBModel.SingleScenarioModel()

Capture a single scenario from a multi-scenario model. Use the ScenarioNumber parameter to indicate which scenario to capture.

The model on which this method is invoked must be a multi-scenario model, and the result will be a single-scenario model.

```
GRBModel  SingleScenarioModel ( )
```

Return value:

Model for a single scenario.

GRBModel.Sync()

Wait for a previous asynchronous optimization call to complete.

Calling OptimizeAsync returns control to the calling routine immediately. The caller can perform other computations while optimization proceeds, and can check on the progress of the optimization by querying various model attributes. The `sync` call forces the calling program to wait until the asynchronous optimization call completes. You *must* call `sync` before the corresponding model object is deleted.

The `sync` call throws an exception if the optimization itself ran into any problems. In other words, exceptions thrown by this method are those that `optimize` itself would have thrown, had the original method not been asynchronous.

Note that you need to call `sync` even if you know that the asynchronous optimization has already completed.

```
void  Sync ( )
```

GRBModel.Terminate()

Generate a request to terminate the current optimization. This method can be called at any time during an optimization (from a callback, from another thread, from an interrupt handler, etc.). Note that, in general, the request won't be acted upon immediately.

When the optimization stops, the Status attribute will be equal to `GRB_INTERRUPTED`.

```
void  Terminate ( )
```

GRBModel.Tune()

Perform an automated search for parameter settings that improve performance. Upon completion, this method stores the best parameter sets it found. The number of stored parameter sets can be determined by querying the value of the TuneResultCount attribute. The actual settings can be retrieved using GetTuneResult

Please refer to the parameter tuning section for details on the tuning tool.

```
void  Tune ( )
```

GRBModel.Update()

Process any pending model modifications.

```
void  Update ( )
```

GRBModel.Write()

This method is the general entry point for writing optimization data to a file. It can be used to write optimization models, solutions vectors, basis vectors, start vectors, or parameter settings. The type of data written is determined by the file suffix. File formats are described in the File Format section.

Note that writing a model to a file will process all pending model modifications. This is also true when writing other model information such as solutions, bases, etc.

Note also that when you write a Gurobi parameter file (PRM), both integer or double parameters not at their default value will be saved, but no string parameter will be saved into the file.

```
void  Write (  string  filename )
```

Arguments:

`filename`: The name of the file to be written. The file type is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, or `.rlp` for writing the model itself, `.dua` or `.dlp` for writing the dualized model (only pure LP), `.ilp` for writing just the IIS associated with an infeasible model (see GRBModel.ComputeIIS for further information), `.sol` for writing the solution selected by the SolutionNumber parameter, `.mst` for writing a start

vector, `.hnt` for writing a hint file, `.bas` for writing an LP basis, `.prm` for writing modified parameter settings, `.attr` for writing model attributes, or `.json` for writing solution information in JSON format. If your system has compression utilities installed (e.g., `7z` or `zip` for Windows, and `gzip`, `bzip2`, or `unzip` for Linux or macOS), then the files can be compressed, so additional suffixes of `.gz`, `.bz2`, or `.7z` are accepted.

6.3 GRBVar

Gurobi variable object. Variables are always associated with a particular model. You create a variable object by adding a variable to a model (using GRBModel.AddVar), rather than by using a `GRBVar` constructor.

The methods on variable objects are used to get and set variable attributes. For example, solution information can be queried by calling Get(`GRB.DoubleAttr.X`). Note, however, that it is generally more efficient to query attributes for a set of variables at once. This is done using the attribute query method on the `GRBModel` object (GRBModel.Get).

GRBVar.Get()

Query the value of a variable attribute.

```
char  Get ( GRB.CharAttr  attr )
```

Query the value of a char-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
double  Get ( GRB.DoubleAttr  attr )
```

Query the value of a double-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
int  Get ( GRB.IntAttr  attr )
```

Query the value of an int-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
string  Get ( GRB.StringAttr  attr )
```

Query the value of a string-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBVar.Index

```
int  Index
```

This property returns the current index, or order, of the variable in the underlying constraint matrix.

Return value:

-2: removed, -1: not in model, otherwise: index of the variable in the model

Note that the index of a variable may change after subsequent model modifications.

GRBVar.SameAs()

```
bool  SameAs (  GRBVar  var2 )
```

Check whether two variable objects refer to the same variable.

Arguments:

var2: The other variable.

Return value:

Boolean result indicates whether the two variable objects refer to the same model variable.

GRBVar.Set()

Set the value of a variable attribute.

```
void  Set (  GRB.CharAttr  attr,
             char          newvalue )
```

Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set (  GRB.DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set (  GRB.IntAttr  attr,
             int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set (  GRB.StringAttr  attr,
             string          newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

6.4 GRBConstr

Gurobi constraint object. Constraints are always associated with a particular model. You create a constraint object by adding a constraint to a model (using GRBModel.AddConstr), rather than by using a `GRBConstr` constructor.

The methods on constraint objects are used to get and set constraint attributes. For example, constraint right-hand sides can be queried by calling Get(`GRB.DoubleAttr.RHS`). Note, however, that it is generally more efficient to query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel.Get).

GRBConstr.Get()

Query the value of a constraint attribute.

```
char  Get ( GRB.CharAttr  attr )
```

Query the value of a char-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
double  Get ( GRB.DoubleAttr  attr )
```

Query the value of a double-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
int  Get ( GRB.IntAttr  attr )
```

Query the value of an int-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
string  Get ( GRB.StringAttr  attr )
```

Query the value of a string-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBConstr.Index

```
int  Index
```

This property returns the current index, or order, of the constraint in the underlying constraint matrix.

Note that the index of a constraint may change after subsequent model modifications.

Return value:

-2: removed, -1: not in model, otherwise: index of the constraint in the model

GRBConstr.SameAs()

```
bool  SameAs (  GRBConstr  constr2 )
```

Check whether two constraint objects refer to the same constraint.

Arguments:

constr2: The other constraint.

Return value:

Boolean result indicates whether the two constraint objects refer to the same model constraint.

GRBConstr.Set()

Set the value of a constraint attribute.

```
void  Set (  GRB.CharAttr  attr,
             char          newvalue )
```

Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set (  GRB.DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set (  GRB.IntAttr  attr,
             int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set (  GRB.StringAttr  attr,
             string          newvalue )
```

Set the value of a string-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

6.5 GRBQConstr

Gurobi quadratic constraint object. Quadratic constraints are always associated with a particular model. You create a quadratic constraint object by adding a quadratic constraint to a model (using GRBModel.AddQConstr), rather than by using a `GRBQConstr` constructor.

The methods on quadratic constraint objects are used to get and set quadratic constraint attributes. For example, quadratic constraint right-hand sides can be queried by calling Get(`GRB.DoubleAttr.QCRHS`). Note, however, that it is generally more efficient to query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel.Get).

GRBQConstr.Get()

Query the value of a quadratic constraint attribute.

```
char  Get ( GRB.CharAttr  attr )
```

Query the value of a char-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
double  Get ( GRB.DoubleAttr  attr )
```

Query the value of a double-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
int  Get ( GRB.IntAttr  attr )
```

Query the value of an int-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
string  Get ( GRB.StringAttr  attr )
```

Query the value of a string-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBQConstr.Set()

Set the value of a quadratic constraint attribute.

```
void  Set ( GRB.CharAttr  attr,
            char          newvalue )
```
Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set ( GRB.DoubleAttr  attr,
            double          newvalue )
```
Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set ( GRB.IntAttr  attr,
            int          newvalue )
```
Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set ( GRB.StringAttr  attr,
            string          newvalue )
```
Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

6.6 GRBSOS

Gurobi SOS constraint object. SOS constraints are always associated with a particular model. You create an SOS object by adding an SOS constraint to a model (using GRBModel.AddSOS), rather than by using a `GRBSOS` constructor. Similarly, SOS constraints are removed using the GRBModel.Remove method.

An SOS constraint can be of type 1 or 2 (`GRB.SOS_TYPE1` or `GRB.SOS_TYPE2`). A type 1 SOS constraint is a set of variables where at most one variable in the set may take a value other than zero. A type 2 SOS constraint is an ordered set of variables where at most two variables in the set may take non-zero values. If two take non-zero values, they must be contiguous in the ordered set.

SOS constraint objects have one attribute, `IISSOS`, which can be queried with the GRBSOS.Get method.

GRBSOS.Get()

Query the value of an SOS attribute.

```
int  Get ( GRB.IntAttr  attr )
```

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBSOS.Set()

Set the value of an SOS attribute.

```
void  Set ( GRB.IntAttr  attr,
            int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

6.7 GRBGenConstr

Gurobi general constraint object. General constraints are always associated with a particular model. You create a general constraint object by adding a general constraint to a model (using one of the GRBModel.AddGenConstrXxx methods), rather than by using a `GRBGenConstr` constructor.

General constraint objects have a number of attributes, which can be queried with the GRBGenConstr.Get method. The full list can be found in the Attributes section of this document.

GRBGenConstr.Get()

Query the value of a general constraint attribute.

```
double  Get (  GRB.DoubleAttr  attr )
```

Query the value of a double-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
int  Get (  GRB.IntAttr  attr )
```

Query the value of an int-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
string  Get (  GRB.StringAttr  attr )
```

Query the value of a string-valued attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBGenConstr.Set()

Set the value of a general constraint attribute.

```
void  Set (  GRB.DoubleAttr  attr,
             string          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

`attr`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

```
void  Set (  GRB.IntAttr  attr,
             string       newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  Set (  GRB.StringAttr  attr,
             string          newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

6.8 GRBExpr

Abstract base class for the GRBLinExpr and GRBQuadExpr classes. Expressions are used to build objectives and constraints. They are temporary objects that typically have short lifespans.

GRBExpr.Value

(Property) The value of an expression for the current solution.

```
double  Value
```

6.9 GRBLinExpr

Gurobi linear expression object. A linear expression consists of a constant term, plus a list of coefficient-variable pairs that capture the linear terms. Linear expressions are used to build constraints. They are temporary objects that typically have short lifespans.

The `GRBLinExpr` class is a sub-class of the abstract base class GRBExpr.

In .NET languages that support operator overloading, you generally build linear expressions using overloaded operators. For example, if `x` is a GRBVar object, then `x + 1` is a GRBLinExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x + 2 * y`, or from other expressions (e.g., `expr2 = 2 * expr1 + x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x`, or `expr2 -= expr1`).

The other option for building expressions is to start with an empty expression (using the GRBLinExpr constructor), and then add terms. Terms can be added individually (using AddTerm) or in groups (using AddTerms or MultAdd). Terms can also be removed from an expression, using Remove.

Given all these options for building expressions, you may wonder which is fastest. For small expressions, you won't need to worry about performance differences between them. If you are building lots of very large expressions (100s of terms), the most efficient approach will be a single call to AddTerms. Using AddTerm. to add individual terms is slightly less efficient, and using overloaded arithemetic operators is the least efficient option.

To add a linear constraint to your model, you generally build one or two linear expression objects (`expr1` and `expr2`) and then use an overloaded comparison operator to build an argument for GRBModel.AddConstr. To give a few examples:

$$\text{model.AddConstr}(expr1 <= expr2)$$
$$\text{model.AddConstr}(expr1 == 1)$$
$$\text{model.AddConstr}(2 * x + 3 * y <= 4)$$

Once you add a constraint to your model, subsequent changes to the expression object you used to build the constraint will not change the constraint (you would use GRBModel.ChgCoeff for that).

Individual terms in a linear expression can be queried using the GetVar and GetCoeff methods. The constant can be queried using the Constant property. You can query the number of terms in the expression using the Size property.

Note that a linear expression may contain multiple terms that involve the same variable. These duplicate terms are merged when creating a constraint from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using GetVar).

GRBLinExpr()

Linear expression constructor. Create an empty linear expression, or copy an existing expression.

```
GRBLinExpr  GRBLinExpr ( )
```

Create an empty linear expression.

Return value:

An empty expression object.

```
GRBLinExpr  GRBLinExpr (  double  a )
```

Create a constant linear expression.

Return value:

A linear expression object.

```
GRBLinExpr  GRBLinExpr ( GRBLinExpr  orig )
```

Copy an existing linear expression.

Arguments:

`orig`: Existing expression to copy.

Return value:

A copy of the input expression object.

GRBLinExpr.Add()

Add one linear expression into another. Upon completion, the invoking linear expression will be equal to the sum of itself and the argument expression.

```
void  Add ( GRBLinExpr  le )
```

Arguments:

`le`: Linear expression to add.

GRBLinExpr.AddConstant()

Add a constant into a linear expression.

```
void  AddConstant ( double  c )
```

Arguments:

`c`: Constant to add to expression.

GRBLinExpr.AddTerm()

Add a single term into a linear expression.

```
void  AddTerm ( double  coeff,
                GRBVar  var )
```

Arguments:

`coeff`: Coefficient for new term.

`var`: Variable for new term.

GRBLinExpr.AddTerms()

Add new terms into a linear expression.

```
void  AddTerms ( double[]  coeffs,
                 GRBVar[]  vars )
```

Add a list of terms into a linear expression. Note that the lengths of the two argument arrays must be equal.

Arguments:

coeffs: Coefficients for new terms.

vars: Variables for new terms.

```
void  AddTerms (  double[]  coeffs,
                  GRBVar[]  vars,
                  int       start,
                  int       len )
```

Add new terms into a linear expression. This signature allows you to use arrays to hold the coefficients and variables that describe the terms in an array without being forced to add a term for each entry in the array. The **start** and **len** arguments allow you to specify which terms to add.

Arguments:

coeffs: Coefficients for new terms.

vars: Variables for new terms.

start: The first term in the list to add.

len: The number of terms to add.

GRBLinExpr.Clear()

Set a linear expression to 0.

You should use the overloaded `expr = 0` instead. The `clear` method is mainly included for consistency with our interfaces to non-overloaded languages.

```
void  Clear ( )
```

GRBLinExpr.Constant

(Property) The constant term from the linear expression.

```
double  Constant
```

GRBLinExpr.GetCoeff()

Retrieve the coefficient from a single term of the expression.

```
double  GetCoeff (  int  i )
```

Return value:

Coefficient for the term at index `i` in the expression.

GRBLinExpr.GetVar()

Retrieve the variable object from a single term of the expression.

```
GRBVar  GetVar (  int  i )
```

Return value:

Variable for the term at index `i` in the expression.

GRBLinExpr.MultAdd()

Add a constant multiple of one linear expression into another. Upon completion, the invoking linear expression is equal the sum of itself and the constant times the argument expression.

```
void  MultAdd (  double      m,
                 GRBLinExpr  le )
```

Arguments:

`m`: Constant multiplier for added expression.

`le`: Linear expression to add.

GRBLinExpr.Remove()

Remove a term from a linear expression.

```
void  Remove (  int  i )
```

Remove the term stored at index `i` of the expression.

Arguments:

`i`: The index of the term to be removed.

```
boolean  Remove (  GRBVar  var )
```

Remove all terms associated with variable `var` from the expression.

Arguments:

`var`: The variable whose term should be removed.

Return value:

Returns `true` if the variable appeared in the linear expression (and was removed).

GRBLinExpr.Size

(Property) The number of terms in the linear expression (not including the constant).

```
int  Size
```

GRBLinExpr.Value

(Property) The value of an expression for the current solution.

```
double  Value
```

6.10 GRBQuadExpr

Gurobi quadratic expression object. A quadratic expression consists of a linear expression, plus a list of coefficient-variable-variable triples that capture the quadratic terms. Quadratic expressions are used to build quadratic objective functions and quadratic constraints. They are temporary objects that typically have short lifespans.

The `GRBQuadExpr` class is a sub-class of the abstract base class GRBExpr.

In .NET languages that support operator overloading, you generally build quadratic expressions using overloaded operators. For example, if `x` is a GRBVar object, then `x * x` is a GRBQuadExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x * x + 2 * x * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x * x`, or `expr2 -= expr1`).

The other option for building expressions is to start with an empty expression (using the GRBQuadExpr constructor), and then add terms. Terms can be added individually (using AddTerm) or in groups (using AddTerms or MultAdd). Terms can also be removed from an expression (using Remove).

Note that the cost of building expressions depends heavily on the approach you use. While you can generally ignore this issue when building small expressions, you should be aware of a few efficiency issues when building large expressions:

- You should avoid using `expr = expr + x*x` or `expr += x*x` in a loop. It will lead to runtimes that are quadratic in the number of terms in the expression.
- Using AddTerm in a loop is reasonably efficient, but it isn't the most efficient approach.
- The most efficient way to build a large expression is to make a single call to AddTerms.

To add a quadratic constraint to your model, you generally build one or two quadratic expression objects (`qexpr1` and `qexpr2`) and then use an overloaded comparison operator to build an argument for GRBModel.AddQConstr. To give a few examples:

$$\begin{aligned}&\text{model.AddQConstr}(qexpr1 <= qexpr2)\\&\text{model.AddQConstr}(qexpr1 == 1)\\&\text{model.AddQConstr}(2 * x * x + 3 * y * y <= 4)\end{aligned}$$

Once you add a constraint to your model, subsequent changes to the expression object you used to build the constraint will have no effect on that constraint.

Individual quadratic terms in a quadratic expression can be queried using the GetVar1 GetVar2, and GetCoeff methods. You can query the number of quadratic terms in the expression using the Size property. To query the constant and linear terms associated with a quadratic expression, first obtain the linear portion of the quadratic expression using LinExpr, and then use the Constant, GetCoeff, or GetVar on the resulting GRBLinExpr object.

Note that a quadratic expression may contain multiple terms that involve the same variable pair. These duplicate terms are merged when creating the model objective from an expression, but they may be visible when inspecting individual quadratic terms in the expression (e.g., when using GetVar1 and GetVar2).

GRBQuadExpr()

Quadratic expression constructor. Create an empty quadratic expression, or copy an existing expression.

```
GRBQuadExpr  GRBQuadExpr ( )
```

Create an empty quadratic expression.

Return value:

An empty expression object.

```
GRBQuadExpr  GRBQuadExpr (  double  a )
```

Create a constant quadratic expression.

Return value:

A quadratic expression object.

```
GRBQuadExpr  GRBQuadExpr (  GRBLinExpr  orig )
```

Initialize a quadratic expression from an existing linear expression.

Arguments:

`orig`: Existing linear expression to copy.

Return value:

Quadratic expression object whose initial value is taken from the input linear expression.

```
GRBQuadExpr  GRBQuadExpr (  GRBQuadExpr  orig )
```

Copy an existing quadratic expression.

Arguments:

`orig`: Existing expression to copy.

Return value:

A copy of the input expression object.

GRBQuadExpr.Add()

Add an expression into a quadratic expression. Upon completion, the invoking quadratic expression will be equal to the sum of itself and the argument expression.

```
void  Add (  GRBLinExpr  le )
```

Add a linear expression.

Arguments:

`le`: Linear expression to add.

```
void  Add (  GRBQuadExpr  qe )
```

Add a quadratic expression.

Arguments:

`qe`: Quadratic expression to add.

GRBQuadExpr.AddConstant()

Add a constant into a quadratic expression.

```
void  AddConstant ( double  c )
```

Arguments:

`c`: Constant to add to expression.

GRBQuadExpr.AddTerm()

Add a single term into a quadratic expression.

```
void  AddTerm ( double  coeff,
                GRBVar  var )
```

Add a single linear term (`coeff*var`) into a quadratic expression.

Arguments:

`coeff`: Coefficient for new term.

`var`: Variable for new term.

```
void  AddTerm ( double  coeff,
                GRBVar  var1,
                GRBVar  var2 )
```

Add a single quadratic term (`coeff*var1*var2`) into a quadratic expression.

Arguments:

`coeff`: Coefficient for new quadratic term.

`var1`: First variable for new quadratic term.

`var2`: Second variable for new quadratic term.

GRBQuadExpr.AddTerms()

Add new terms into a quadratic expression.

```
void  AddTerms ( double[]  coeffs,
                 GRBVar[]  vars )
```

Add a list of linear terms into a quadratic expression. Note that the lengths of the two argument arrays must be equal.

Arguments:

`coeffs`: Coefficients for new terms.

`vars`: Variables for new terms.

```
void  AddTerms ( double[]  coeffs,
                 GRBVar[]  vars,
                 int       start,
                 int       len )
```

Add new linear terms into a quadratic expression. This signature allows you to use arrays to hold the coefficients and variables that describe the linear terms in an array without being forced

to add a term for each entry in the array. The `start` and `len` arguments allow you to specify which terms to add.

Arguments:

`coeffs`: Coefficients for new terms.

`vars`: Variables for new terms.

`start`: The first term in the list to add.

`len`: The number of terms to add.

```
void  AddTerms ( double[]  coeffs,
                 GRBVar[]  vars1,
                 GRBVar[]  vars2 )
```

Add a list of quadratic terms into a quadratic expression. Note that the lengths of the three argument arrays must be equal.

Arguments:

`coeffs`: Coefficients for new quadratic terms.

`vars1`: First variables for new quadratic terms.

`vars2`: Second variables for new quadratic terms.

```
void  AddTerms ( double[]  coeffs,
                 GRBVar[]  vars1,
                 GRBVar[]  vars2,
                 int       start,
                 int       len )
```

Add new quadratic terms into a quadratic expression. This signature allows you to use arrays to hold the coefficients and variables that describe the terms in an array without being forced to add a term for each entry in the array. The `start` and `len` arguments allow you to specify which terms to add.

Arguments:

`coeffs`: Coefficients for new quadratic terms.

`vars1`: First variables for new quadratic terms.

`vars2`: Second variables for new quadratic terms.

`start`: The first term in the list to add.

`len`: The number of terms to add.

GRBQuadExpr.Clear()

Set a quadratic expression to 0.

You should use the overloaded `expr = 0` instead. The `clear` method is mainly included for consistency with our interfaces to non-overloaded languages.

```
void  Clear ( )
```

GRBQuadExpr.GetCoeff()

Retrieve the coefficient from a single quadratic term of the quadratic expression.

```
double  GetCoeff (  int  i )
```

Return value:
Coefficient for the quadratic term at index `i` in the expression.

GRBQuadExpr.GetVar1()

Retrieve the first variable object associated with a single quadratic term from the expression.

```
GRBVar  GetVar1 (  int  i )
```

Return value:
First variable for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr.GetVar2()

Retrieve the second variable object associated with a single quadratic term from the expression.

```
GRBVar  GetVar2 (  int  i )
```

Return value:
Second variable for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr.LinExpr()

(Property) A quadratic expression is represented as a linear expression, plus a list of quadratic terms. This method retrieves the linear expression associated with the quadratic expression.

```
GRBLinExpr  getLinExpr
```

GRBQuadExpr.MultAdd()

Add a constant multiple of one quadratic expression into another. Upon completion, the invoking quadratic expression is equal the sum of itself and the constant times the argument expression.

```
void  MultAdd (  double      m,
                 GRBLinExpr  le )
```

Add a linear expression into a quadratic expression.

Arguments:

`m`: Constant multiplier for added expression.

`le`: Linear expression to add.

```
void  MultAdd (  double       m,
                 GRBQuadExpr  qe )
```

Add a quadratic expression into a quadratic expression.

Arguments:

`m`: Constant multiplier for added expression.

`qe`: Quadratic expression to add.

GRBQuadExpr.Remove()

Remove a quadratic term from a quadratic expression.

```
void  Remove (  int  i )
```

Remove the quadratic term stored at index `i` of the expression.

Arguments:

`i`: The index of the quadratic term to be removed.

```
boolean  Remove (  GRBVar  var )
```

Remove all quadratic terms associated with variable `var` from the expression.

Arguments:

`var`: The variable whose quadratic term should be removed.

Return value:

Returns `true` if the variable appeared in the quadratic expression (and was removed).

GRBQuadExpr.Size

(Property) The number of quadratic terms in the quadratic expression. Use GRBQuadExpr.LinExpr to retrieve constant or linear terms from the quadratic expression.

```
int  size
```

GRBQuadExpr.Value

(Property) The value of an expression for the current solution.

```
double  Value
```

6.11 GRBTempConstr

Gurobi temporary constraint object. Objects of this class are created as intermediate results when building constraints using overloaded operators. There are no public methods on this class. Instead, `GRBTempConstr` objects are created by operators ==, <=, or >=. You will generally never store objects of this class in your own variables.

Consider the following examples:

```
model.AddConstr(x + y <= 1);
model.AddQConstr(x*x + y*y <= 1);
```

The overloaded `<=` operator creates an object of type `GRBTempConstr`, which is then immediately passed to GRBModel.AddConstr or GRBModel.AddQConstr.

6.12 GRBColumn

Gurobi column object. A column consists of a list of coefficient, constraint pairs. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients. They are temporary objects that typically have short lifespans.

You generally build columns by starting with an empty column (using the GRBColumn constructor), and then adding terms. Terms can be added individually, using AddTerm, or in groups, using AddTerms. Terms can also be removed from a column, using Remove.

Individual terms in a column can be queried using the GetConstr, and GetCoeff methods. You can query the number of terms in the column using the Size property.

GRBColumn()

Column constructor. Create an empty column, or copy an existing column.

```
GRBColumn  GRBColumn ( )
```

Create an empty column.

Return value:

An empty column object.

```
GRBColumn  GRBColumn (  GRBColumn  orig )
```

Copy an existing column.

Return value:

A copy of the input column object.

GRBColumn.AddTerm()

Add a single term into a column.

```
void  AddTerm (  double     coeff,
                 GRBConstr  constr )
```

Arguments:

coeff: Coefficient for new term.

constr: Constraint for new term.

GRBColumn.AddTerms()

Add new terms into a column.

```
void  AddTerms (  double[]     coeffs,
                  GRBConstr[]  constrs )
```

Add a list of terms into a column. Note that the lengths of the two argument arrays must be equal.

Arguments:

coeffs: Coefficients for added constraints.

constrs: Constraints to add to column.

```
void  AddTerms ( double[]     coeffs,
                 GRBConstr[]  constrs,
                 int          start,
                 int          len )
```

Add new terms into a column. This signature allows you to use arrays to hold the coefficients and constraints that describe the terms in an array without being forced to add an term for each member in the array. The `start` and `len` arguments allow you to specify which terms to add.

Arguments:

`coeffs`: Coefficients for added constraints.

`constrs`: Constraints to add to column.

`start`: The first term in the list to add.

`len`: The number of terms to add.

GRBColumn.Clear()

Remove all terms from a column.

```
void  Clear ( )
```

GRBColumn.GetCoeff()

Retrieve the coefficient from a single term in the column.

```
double  GetCoeff (  int  i )
```

Return value:

Coefficient for the term at index `i` in the column.

GRBColumn.GetConstr()

Retrieve the constraint object from a single term in the column.

```
GRBConstr  GetConstr (  int  i )
```

Return value:

Constraint for the term at index `i` in the column.

GRBColumn.Remove()

Remove a single term from a column.

```
GRBConstr  Remove (  int  i )
```

Remove the term stored at index `i` of the column.

Arguments:

`i`: The index of the term to be removed.

Return value:

The constraint whose term was removed from the column. Returns `null` if the specified index is out of range.

```
boolean  Remove (  GRBConstr  constr )
```

Remove the term associated with constraint `constr` from the column.

Arguments:

`constr`: The constraint whose term should be removed.

Return value:

Returns `true` if the constraint appeared in the column (and was removed).

GRBColumn.Size

(Property) The number of terms in the column.

```
int  Size
```

6.13 Overloaded Operators

The Gurobi .NET interface overloads several arithmetic and comparison operators. Overloaded arithmetic operators (`+, -, *, /`) are used to create linear and quadratic expressions. Overloaded comparison operators (`<=, >=, and ==`) are used to build linear and quadratic constraints.

Note that the results of overloaded comparison operators are generally never stored in user variables. They are immediately passed to GRBModel.AddConstr or GRBModel.AddQConstr.

operator <=

Create an inequality constraint.

```
GRBTempConstr  operator <= (  GRBQuadExpr  lhsExpr,
                              GRBQuadExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of inequality constraint.

rhsExpr: Right-hand side of inequality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to method GRBModel.AddConstr.

operator >=

Create an inequality constraint.

```
GRBTempConstr  operator >= (  GRBQuadExpr  lhsExpr,
                              GRBQuadExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of inequality constraint.

rhsExpr: Right-hand side of inequality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to method GRBModel.AddConstr.

operator ==

Create an equality constraint.

```
GRBTempConstr  operator == (  GRBLinExpr  lhsExpr,
                              GRBLinExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of equality constraint.

rhsExpr: Right-hand side of equality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to method GRBModel.AddConstr.

operator +

Create a new expression by adding a pair of Gurobi objects.

```
GRBLinExpr  operator + (  GRBLinExpr  expr1,
                          GRBLinExpr  expr2 )
```

Arguments:

expr1: First linear expression argument.

expr2: Second linear expression argument.

Return value:

A linear expression that is equal to the sum of the two argument expressions.

```
GRBLinExpr  operator + (  GRBLinExpr  expr,
                          GRBVar      var )
```

Arguments:

expr: Linear expression argument.

var: Variable argument.

Return value:

A linear expression that is equal to the sum of the argument linear expression and the argument variable.

```
GRBLinExpr  operator + (  GRBVar      var,
                          GRBLinExpr  expr )
```

Arguments:

var: Variable argument.

expr: Linear expression argument.

Return value:

A linear expression that is equal to the sum of the argument linear expression and the argument variable.

```
GRBLinExpr  operator + (  GRBVar  var1,
                          GRBVar  var2 )
```

Arguments:

var1: First variable argument.

var2: Second variable argument.

Return value:

A linear expression that is equal to the sum of the two argument variables.

```
GRBLinExpr  operator + (  double  a,
                          GRBVar  var )
```

Arguments:

a: Constant.

var: Variable.

Return value:
A linear expression that is equal to the sum of the constant and the variable argument.

```
GRBLinExpr  operator + (  GRBVar  var,
                          double  a )
```

Arguments:
var: Variable.
a: Constant.
Return value:
A linear expression that is equal to the sum of the constant and the variable argument.

```
GRBQuadExpr  operator + (  GRBQuadExpr  expr1,
                           GRBQuadExpr  expr2 )
```

Arguments:
expr1: First quadratic expression argument.
expr2: Second quadratic expression argument.
Return value:
A quadratic expression that is equal to the sum of the two argument quadratic expressions.

```
GRBQuadExpr  operator + (  GRBQuadExpr  expr,
                           GRBVar       var )
```

Arguments:
expr: Quadratic expression argument.
var: Variable argument.
Return value:
A quadratic expression that is equal to the sum of the argument quadratic expression and the argument variable.

```
GRBQuadExpr  operator + (  GRBVar       var,
                           GRBQuadExpr  expr )
```

Arguments:
var: Variable argument.
expr: Quadratic expression argument.
Return value:
A quadratic expression that is equal to the sum of the argument quadratic expression and the argument variable.

operator -

Create a new expression by subtracting one Gurobi object from another.

```
GRBLinExpr  operator - (  GRBLinExpr  expr1,
                          GRBLinExpr  expr2 )
```

Arguments:

expr1: First linear expression argument.

expr2: Second linear expression argument.

Return value:

A linear expression that is equal to the first expression minus the second.

```
GRBQuadExpr  operator - (  GRBQuadExpr  expr1,
                           GRBQuadExpr  expr2 )
```

Arguments:

expr1: First quadratic expression argument.

expr2: Second quadratic expression argument.

Return value:

A quadratic expression that is equal to the first expression minus the second.

operator *

Create a new expression by multiplying a pair of Gurobi objects.

```
GRBLinExpr  operator * (  double      multiplier,
                          GRBLinExpr  expr )
```

Arguments:

multiplier: Multiplier for expression argument.

expr: Expression argument.

Return value:

A linear expression that is equal to the input expression times the input multiplier.

```
GRBLinExpr  operator * (  GRBLinExpr  expr,
                          double      multiplier )
```

Arguments:

expr: Linear expression argument.

multiplier: Multiplier for expression argument.

Return value:

A linear expression that is equal to the input expression times the input multiplier.

```
GRBLinExpr  operator * (  double  multiplier,
                          GRBVar  var )
```

Arguments:

multiplier: Multiplier for variable argument.

var: Variable argument.

Return value:

A linear expression that is equal to the input variable times the input multiplier.

```
GRBLinExpr  operator * (  GRBVar  var,
                          double  multiplier )
```

Arguments:

var: Variable argument.

multiplier: Multiplier for variable argument.

Return value:

A linear expression that is equal to the input variable times the input multiplier.

```
GRBQuadExpr  operator * (  double       multiplier,
                           GRBQuadExpr  expr )
```

Arguments:

multiplier: Multiplier for expression argument.

expr: Quadratic expression argument.

Return value:

A quadratic expression that is equal to the input expression times the input multiplier.

```
GRBQuadExpr  operator * (  GRBQuadExpr  expr,
                           double       multiplier )
```

Arguments:

expr: Quadratic expression argument.

multiplier: Multiplier for expression argument.

Return value:

A quadratic expression that is equal to the input expression times the input multiplier.

```
GRBQuadExpr  operator * (  GRBVar  var1,
                           GRBVar  var2 )
```

Arguments:

var1: First variable argument.

var2: Second variable argument.

Return value:

A quadratic expression that is equal to the product of the two input variables.

```
GRBQuadExpr  operator * (  GRBVar      var,
                           GRBLinExpr  expr )
```

Arguments:

var: Input variable.

expr: Input linear expression.

Return value:

A quadratic expression that is equal to the input linear expression times the input variable.

```
GRBQuadExpr  operator * (  GRBLinExpr  expr,
                           GRBVar      var )
```

Arguments:

expr: Input linear expression.

var: Input variable.

Return value:

A quadratic expression that is equal to the input linear expression times the input variable.

```
GRBQuadExpr  operator * (  GRBLinExpr  expr1,
                           GRBLinExpr  expr2 )
```

Arguments:

expr1: First linear expression argument.

expr2: Second linear expression argument.

Return value:

A quadratic expression that is equal to the product of the two input linear expressions.

operator /

Create a new expression by dividing a Gurobi variable by a constant.

```
GRBLinExpr  operator / (  GRBVar  var,
                          double  divisor )
```

Arguments:

var: Variable argument.

divisor: Divisor for variable argument.

Return value:

A linear expression that is equal to the input variable divided by the input divisor.

implicit cast

Create an expression from an implicit cast (e.g., `expr = 0.0` or `expr = x`).

```
GRBLinExpr  GRBLinExpr (  double  value )
```

Arguments:

value: Desired value for linear expression.

Return value:

A linear expression that is equal to specified constant.

```
GRBQuadExpr  GRBQuadExpr (  double  value )
```

Arguments:

value: Desired value for quadratic expression.

Return value:

A quadratic expression that is equal to specified constant.

```
GRBLinExpr  GRBLinExpr (  GRBVar  var )
```

Arguments:

value: Desired value for linear expression.

Return value:

A linear expression that is equal to specified variable.

```
GRBQuadExpr  GRBQuadExpr ( GRBVar var )
```

Arguments:

`value`: Desired value for quadratic expression.

Return value:

A quadratic expression that is equal to specified variable.

```
GRBQuadExpr  GRBQuadExpr ( GRBLinExpr expr )
```

Arguments:

`expr`: Desired value for quadratic expression.

Return value:

A quadratic expression that is equal to specified linear expression.

6.14 GRBCallback

Gurobi callback class. This is an abstract class. To implement a callback, you should create a subclass of this class and implement a `callback()` method. If you pass an object of this subclass to method GRBModel.SetCallback before calling GRBModel.Optimize or GRBModel.ComputeIIS, the `callback()` method of the class will be called periodically. Depending on where the callback is called from, you will be able to obtain various information about the progress of the optimization.

Note that this class contains one protected *int* member variable: `where`. You can query this variable from your `callback()` method to determine where the callback was called from.

Gurobi callbacks can be used both to monitor the progress of the optimization and to modify the behavior of the Gurobi Optimizer. A simple user callback function might call the GRBCallback.GetIntInfo or GRBCallback.GetDoubleInfo methods to produce a custom display, or perhaps to terminate optimization early (using GRBCallback.Abort) or to proceed to the next phase of the computation (using GRBCallback.Proceed). More sophisticated MIP callbacks might use GRBCallback.GetNodeRel or GRBCallback.GetSolution to retrieve values from the solution to the current node, and then use GRBCallback.AddCut or GRBCallback.AddLazy to add a constraint to cut off that solution, or GRBCallback.SetSolution to import a heuristic solution built from that solution. For multi-objective problems, you might use GRBCallback.StopOneMultiObj to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

When solving a model using multiple threads, the user callback is only ever called from a single thread, so you don't need to worry about the thread-safety of your callback.

Note that changing parameters from within a callback is not supported, doing so may lead to undefined behavior.

You can look at the `callback_cs.cs` example for details of how to use Gurobi callbacks.

GRBCallback()

Callback constructor.

```
GRBCallback  GRBCallback ( )
```

Return value:

A callback object.

GRBCallback.Abort()

Abort optimization. When the optimization stops, the Status attribute will be equal to `GRB.Status.INTERRUPTED`.

```
void  Abort ( )
```

GRBCallback.AddCut()

Add a cutting plane to the MIP model from within a callback function. Note that this method can only be invoked when the `where` member variable is equal to `GRB.Callback.MIPNODE` (see the Callback Codes section for more information).

Cutting planes can be added at any node of the branch-and-cut tree. However, they should be added sparingly, since they increase the size of the relaxation model that is solved at each node and can significantly degrade node processing speed.

Cutting planes are typically used to cut off the current relaxation solution. To retrieve the relaxation solution at the current node, you should first call GetNodeRel.

You should consider setting parameter PreCrush to value 1 when adding your own cuts. This setting shuts off a few presolve reductions that can sometimes prevent your cut from being applied to the presolved model (which would result in your cut being silently ignored).

Note that cutting planes added through this method must truly be cutting planes — they can cut off continuous solutions, but they may not cut off integer solutions that respect the original constraints of the model. Ignoring this restriction will lead to incorrect solutions.

```
void  AddCut ( GRBLinExpr  lhsExpr,
               char        sense,
               double      rhsVal )
```

Arguments:

lhsExpr: Left-hand side expression for new cutting plane.

sense: Sense for new cutting plane (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

rhsVal: Right-hand side value for new cutting plane.

```
void  AddConstr ( GRBTempConstr  tempConstr )
```

Arguments:

tempConstr: Temporary constraint object, created by an overloaded comparison operator.

GRBCallback.AddLazy()

Add a lazy constraint to the MIP model from within a callback function. Note that this method can only be invoked when the `where` member variable is `GRB.Callback.MIPNODE` or `GRB.Callback.MIPSOL` (see the Callback Codes section for more information).

Lazy constraints are typically used when the full set of constraints for a MIP model is too large to represent explicitly. By only including the constraints that are actually violated by solutions found during the branch-and-cut search, it is sometimes possible to find a proven optimal solution while only adding a fraction of the full set of constraints.

You would typically add a lazy constraint by first querying the current node solution (by calling GetSolution from a `GRB.Callback.MIPSOL` callback, or GetNodeRel from a `GRB.Callback.MIPNODE` callback), and then calling `AddLazy()` to add a constraint that cuts off the solution. Gurobi guarantees that you will have the opportunity to cut off any solutions that would otherwise be considered feasible.

MIP solutions may be generated outside of a MIP node. Thus, generating lazy constraints is optional when the `where` value in the callback function equals `GRB.Callback.MIPNODE`. To avoid this, we recommend to always check when the `where` value equals `GRB.Callback.MIPSOL`.

Your callback should be prepared to cut off solutions that violate any of your lazy constraints, including those that have already been added. Node solutions will usually respect previously added lazy constraints, but not always.

Note that you must set the LazyConstraints parameter if you want to use lazy constraints.

```
void  AddLazy ( GRBLinExpr  lhsExpr,
                char        sense,
                double      rhsVal )
```

Arguments:

lhsExpr: Left-hand side expression for new lazy constraint.

sense: Sense for new lazy constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhsVal: Right-hand side value for new lazy constraint.

```
void  AddConstr ( GRBTempConstr tempConstr )
```

Arguments:

tempConstr: Temporary constraint object, created by an overloaded comparison operator.

GRBCallback.GetDoubleInfo()

Request double-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the double-valued information that can be queried for different values of **where**, please refer to the Callback section.

```
double  GetDoubleInfo ( int what )
```

Arguments:

what: Information requested (refer the list of Gurobi Callback Codes for possible values).

Return value:

Value of requested callback information.

GRBCallback.GetIntInfo()

Request int-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the int-valued information that can be queried for different values of **where**, please refer to the Callback section.

```
int  GetIntInfo ( int what )
```

Arguments:

what: Information requested (refer the list of Gurobi Callback Codes for possible values).

Return value:

Value of requested callback information.

GRBCallback.GetNodeRel()

Retrieve values from the node relaxation solution at the current node. Only available when the **where** member variable is equal to GRB.Callback.MIPNODE, and GRB.Callback.MIPNODE_STATUS is equal to GRB.Status.OPTIMAL.

```
double  GetNodeRel ( GRBVar v )
```

Arguments:

v: The variable whose value is desired.

Return value:

The value of the specified variable in the node relaxation for the current node.

```
double[]  GetNodeRel ( GRBVar[]  xvars )
```

Arguments:

xvars: The list of variables whose values are desired.

Return value:

The values of the specified variables in the node relaxation for the current node.

```
double[][]  GetNodeRel ( GRBVar[][]  xvars )
```

Arguments:

xvars: The array of variables whose values are desired.

Return value:

The values of the specified variables in the node relaxation for the current node.

GRBCallback.GetSolution()

Retrieve values from the current solution vector. Only available when the **where** member variable is equal to GRB.Callback.MIPSOL or GRB.Callback.MULTIOBJ.

```
double  GetSolution ( GRBVar  v )
```

Arguments:

v: The variable whose value is desired.

Return value:

The value of the specified variable in the current solution vector.

```
double[]  GetSolution ( GRBVar[]  xvars )
```

Arguments:

xvars: The list of variables whose values are desired.

Return value:

The values of the specified variables in the current solution.

```
double[][]  GetSolution ( GRBVar[][]  xvars )
```

Arguments:

xvars: The array of variables whose values are desired.

Return value:

The values of the specified variables in the current solution.

GRBCallback.GetStringInfo()

Request string-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the string-valued information that can be queried for different values of **where**, please refer to the Callback section.

```
string  GetStringInfo ( int  what )
```

Arguments:

what: Information requested (refer the list of Gurobi Callback Codes for possible values).

Return value:

Value of requested callback information.

GRBCallback.Proceed()

Generate a request to proceed to the next phase of the computation. Note that the request is only accepted in a few phases of the algorithm, and it won't be acted upon immediately.

In the current Gurobi version, this callback allows you to proceed from the NoRel heuristic to the standard MIP search. You can determine the current algorithm phase using `MIP_PHASE`, `MIPNODE_PHASE`, or `MIPSOL_PHASE` queries from a callback.

```
void  Proceed ( )
```

GRBCallback.SetSolution()

Import solution values for a heuristic solution. Only available when the `where` member variable is equal to `GRB.Callback.MIP`, `GRB.Callback.MIPNODE`, or `GRB.Callback.MIPSOL` (see the Callback Codes section for more information).

When you specify a heuristic solution from a callback, variables initially take undefined values. You should use this method to specify variable values. You can make multiple calls to `SetSolution` from one callback invocation to specify values for multiple sets of variables. After the callback, if values have been specified for any variables, the Gurobi Optimizer will try to compute a feasible solution from the specified values, possibly filling in values for variables whose values were left undefined. You can also optionally call UseSolution within your callback function to try to immediately compute a feasible solution from the specified values.

Note that this method is not supported in a Compute Server environment.

```
void  SetSolution ( GRBVar  v,
                    double  val )
```

Arguments:

`v`: The variable whose values is being set.

`val`: The value of the variable in the new solution.

```
void  SetSolution ( GRBVar[]  xvars,
                    double[]  sol )
```

Arguments:

`xvars`: The variables whose values are being set.

`sol`: The desired values of the specified variables in the new solution.

GRBCallback.StopOneMultiObj()

Interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process. Only available for multi-objective MIP models and when the `where` member variable is not equal to `GRB.Callback.MULTIOBJ` (see the Callback Codes section for more information).

You would typically stop a multi-objective optimization step by querying the last finished number of multi-objectives steps, and using that number to stop the current step and move on to the next hierarchical objective (if any) as shown in the following example:

Example usage:

```
using Gurobi;

class callback : GRBCallback
{

  private int objcnt;
  private long starttime;

  protected override void Callback() {
    try {
      if (where == GRB.Callback.MULTIOBJ) {
        /* get current objective number */
        objcnt    = GetIntInfo(GRB.Callback.MULTIOBJ_OBJCNT);

        /* reset start time to current time */
        starttime = DateTime.Now.Ticks;

      } else if (DateTime.Now.Ticks - starttime > BIG ||
                 /* takes too long or good enough */) {
        /* stop only this optimization step */
        StopOneMultiObj(objcnt);
      }
    }
  }
}
```

You should refer to the section on Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

```
void  StopOneMultiObj ( int objcnt )
```

Arguments:

`objnum`: The number of the multi-objective optimization step to interrupt. For processes running locally, this argument can have the special value -1, meaning to stop the current step.

GRBCallback.UseSolution()

Once you have imported solution values using SetSolution, you can optionally call `UseSolution` in a `GRB.Callback.MIPNODE` callback to immediately use these values to try to compute a heuristic solution. Alternatively, you can call `UseSolution` in a `GRB.Callback.MIP` or `GRB.Callback.MIPSOL` callback, which will store the solution until it can be processed internally.

```
double  UseSolution ( )
```

Return value:

The objective value for the solution obtained from your solution values. It equals `GRB.INFINITY` if no improved solution is found or the method has been called from a callback other than

`GRB.Callback.MIPNODE` as, in these contexts, the solution is stored instead of being processed immediately.

6.15 GRBException

Gurobi exception object. This is a sub-class of the .NET `Exception` class. A number of useful properties, including `Message()` and `StackTrace()`, are inherited from the parent class. For a list of parent class methods, visit this site.

GRBException()

Exception constructor.

```
GRBException  GRBException (  int  errcode )
```

Create a Gurobi exception.

Arguments:

errcode: Error code for exception.

Return value:

An exception object.

```
GRBException  GRBException (  string  errmsg )
```

Create a Gurobi exception.

Arguments:

errmsg: Error message for exception.

Return value:

An exception object.

```
GRBException  GRBException (  string  errmsg,
                              int     errcode )
```

Create a Gurobi exception.

Arguments:

errmsg: Error message for exception.

errcode: Error code for exception.

Return value:

An exception object.

GRBException.ErrorCode

(Property) The error code associated with a Gurobi exception.

```
int  ErrorCode
```

6.16 GRBBatch

Gurobi batch object. Batch optimization is a feature available with the Gurobi Cluster Manager. It allows a client program to build an optimization model, submit it to a Compute Server cluster (through a Cluster Manager), and later check on the status of the model and retrieve its solution. For more information, please refer to the Batch Optimization section.

Commonly used methods on batch objects include Update (refresh attributes from the Cluster Manager), Abort (abort execution of a batch request), Retry (retry optimization for an interrupted or failed batch), Discard (remove the batch request and all related information from the Cluster Manager), and GetJSONSolution (query solution information for the batch request).

These methods are built on top of calls to the Cluster Manager REST API. They are meant to simplify such calls, but note that you always have the option of calling the REST API directly.

Batch objects have four attributes:

- BatchID: Unique ID for the batch request.
- BatchStatus: Last batch status.
- BatchErrorCode: Last error code.
- BatchErrorMessage: Last error message.

You can access their values , `batch.BatchStatus` or `batch.BatchID`, or by using Get. Note that all Batch attributes are locally cached, and are only updated when you create a client-side batch object or when you explicitly update this cache, which can done by calling Update.

GRBBatch()

Constructor for `GRBBatch`.

Given a `BatchID`, as returned by optimizeBatch, and a Gurobi environment that can connect to the appropriate Cluster Manager (i.e., one where parameters CSManager, UserName, and ServerPassword have been set appropriately), this function returns a GRBBatch object. With it, you can query the current status of the associated batch request and, once the batch request has been processed, you can query its solution. Please refer to the Batch Optimization section for details and examples.

```
GRBBatch  GRBBatch (  GRBEnv&  env,
                      string&  batchID )
```

Arguments:

`env`: The environment in which the new batch object should be created.

`batchID`: ID of the batch request for which you want to access status and other information.

Return value:

New batch object.

Example usage:

```
GRBBatch batch = GRBBatch(env, batchID);
```

GRBBatch.Abort()

This method instructs the Cluster Manager to abort the processing of this batch request, changing its status to `ABORTED`. Please refer to the Batch Status Codes section for further details.

```
void  Abort ( )
```

Example usage:

```
// Abort this batch if it is taking too long
TimeSpan interval = DateTime.Now - start;
if (interval.TotalSeconds > maxwaittime) {
  batch.Abort();
  break;
}
```

GRBBatch.Discard()

This method instructs the Cluster Manager to remove all information related to the batch request in question, including the stored solution if available. Further queries for the associated batch request will fail with error code `GRB_ERROR_DATA_NOT_AVAILABLE`. Use this function with care, as the removed information can not be recovered later on.

```
void  Discard ( )
```

Example usage:

```
// Remove batch request from manager
batch.Discard();
```

GRBBatch.GetJSONSolution()

This method retrieves the solution of a completed batch request from a Cluster Manager. The solution is returned as a JSON solution string. For this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details.

```
string  GetJSONSolution ( )
```

Return value:

The requested solution in JSON format.

Example usage:

```
// Get JSON solution as string
Console.WriteLine("JSON solution:" + batch.GetJSONSolution());
```

GRBBatch.Get()

Query the value of an attribute.

```
int  Get (  GRB_IntAttr  attr )
```

Query the value of an int-valued batch attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

`string Get ( GRB_StringAttr attr )`

Query the value of a string-valued batch attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

GRBBatch.Retry()

This method instructs the Cluster Manager to retry optimization of a failed or aborted batch request, changing its status to `SUBMITTED`. Please refer to the Batch Status Codes section for further details. `void Retry ( )`

Example usage:

```
// If the batch failed, we retry it
if (batch.BatchStatus == GRB.BatchStatus.FAILED) {
  batch.Retry();
  System.Threading.Thread.Sleep(2000);
  batch.Update();
}
```

GRBBatch.Update()

All Batch attribute values are cached locally, so queries return the value received during the last communication with the Cluster Manager. This method refreshes the values of all attributes with the values currently available in the Cluster Manager (which involves network communication). `void Update ( )`

Example usage:

```
// Update the resident attribute cache of the Batch object
// with the latest values from the cluster manager.
batch.Update();
```

GRBBatch.WriteJSONSolution()

This method returns the stored solution of a completed batch request from a Cluster Manager. The solution is returned in a gzip-compressed JSON file. The file name you provide must end with a `.json.gz` extension. The JSON format is described in the JSON solution format section. Note that for this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details. `WriteJSONSolution void ( string& filename )`

Arguments:

`filename`: Name of file where the solution should be stored (in JSON format).

Example usage:

```
// Write the full JSON solution string to a file
batch.WriteJSONSolution("batch-sol.json.gz");
```

6.17 GRB

Class for .NET enums and constants. The enums are used to get or set Gurobi attributes or parameters.

Constants

The following list contains the set of constants needed by the Gurobi .NET interface. You would refer to them using a `GRB.` prefix (e.g., `GRB.Status.OPTIMAL`).

```
// Model status codes (after call to optimize())

/// <summary>
/// Optimization status codes
/// </summary>
public class Status
{
  /// <summary>
  /// Model is loaded, but no solution information is available.
  /// </summary>
  public const int LOADED = 1;

  /// <summary>
  /// Model was solved to optimality (subject to tolerances) and an optimal
  /// solution is available.
  /// </summary>
  public const int OPTIMAL = 2;

  /// <summary>
  /// Model was proven to be infeasible.
  /// </summary>
  public const int INFEASIBLE = 3;

  /// <summary>
  /// Model was proven to be either infeasible or unbounded.
  /// </summary>
  public const int INF_OR_UNBD = 4;

  /// <summary>
  /// Model was proven to be unbounded.
  /// </summary>
  public const int UNBOUNDED = 5;

  /// <summary>
  /// Optimal objective was proven to be worse than the value specified
  /// in the Cutoff parameter.
  /// </summary>
  public const int CUTOFF = 6;

  /// <summary>
  /// Optimization terminated because the total number of simplex
  /// iterations performed exceeded the limit specified in the IterationLimit
  /// parameter.
  /// </summary>
  public const int ITERATION_LIMIT = 7;
```

```
    /// <summary>
    /// Optimization terminated because the total number of branch-and-cut
    /// nodes explored exceeded the limit specified in the NodeLimit
    /// parameter.
    /// </summary>
    public const int NODE_LIMIT = 8;

    /// <summary>
    /// Optimization terminated because the total elapsed time
    /// exceeded the limit specified in the TimeLimit parameter.
    /// </summary>
    public const int TIME_LIMIT = 9;

    /// <summary>
    /// Optimization terminated because the number of solutions found
    /// reached the value specified in the SolutionLimit parameter.
    /// </summary>
    public const int SOLUTION_LIMIT = 10;

    /// <summary>
    /// Optimization was terminated by the user.
    /// </summary>
    public const int INTERRUPTED = 11;

    /// <summary>
    /// Optimization was terminated due to unrecoverable numerical
    /// difficulties.
    /// </summary>
    public const int NUMERIC = 12;

    /// <summary>
    /// Optimization terminated with a sub-optimal solution.
    /// </summary>
    public const int SUBOPTIMAL = 13;

    /// <summary>
    /// Optimization is still in progress.
    /// </summary>
    public const int INPROGRESS = 14;

    /// <summary>
    /// User specified objective limit (a bound on either the best objective
    /// or the best bound), and that limit has been reached.
    /// </summary>
    public const int USER_OBJ_LIMIT = 15;

    /// <summary>
    /// Optimization terminated because the total elapsed work
    /// exceeded the limit specified in the WorkLimit parameter.
    /// </summary>
    public const int WORK_LIMIT = 16;

    /// <summary>
    /// Optimization terminated because the total amount of allocated
    /// memory exceeded the limit specified in the SoftMemLimit parameter.
    /// </summary>
```

```
  public const int MEM_LIMIT = 17;
}

/// <summary>
/// Batch status codes
/// </summary>
public class BatchStatus
{

  /// <summary>
  /// Batch object was created, but is not ready to be scheduled.
  /// See the Batch Optimization section of the manual for more details.
  /// </summary>
  public const int CREATED   = 1;

  /// <summary>
  /// The Batch object has been completely specified, and now is waiting
  /// for a job to finish processing the request.
  /// See the Batch Optimization section of the manual for more details.
  /// </summary>
  public const int SUBMITTED = 2;

  /// <summary>
  /// Batch processing was aborted by the user.
  /// See the Batch Optimization section of the manual for more details.
  /// </summary>
  public const int ABORTED   = 3;

  /// <summary>
  /// Batch processing failed.
  /// See the Batch Optimization section of the manual for more details.
  /// </summary>
  public const int FAILED    = 4;

  /// <summary>
  /// A Batch Job successfully processed the Batch request.
  /// See the Batch Optimization section of the manual for more details.
  /// </summary>
  public const int COMPLETED = 5;
}

// Constraint senses

/// <summary>
/// Less or equal constraint sense.
/// </summary>
public const char LESS_EQUAL    = '<';
/// <summary>
/// Greater or equal constraint sense.
/// </summary>
public const char GREATER_EQUAL = '>';
/// <summary>
/// Equal constraint sense.
/// </summary>
public const char EQUAL         = '=';
```

```
// Variable types

/// <summary>
/// Continuous variable.
/// </summary>
public const char CONTINUOUS   = 'C';
/// <summary>
/// Binary variable.
/// </summary>
public const char BINARY       = 'B';
/// <summary>
/// Integer variable.
/// </summary>
public const char INTEGER      = 'I';
/// <summary>
/// Semi-continuous variable.
/// </summary>
public const char SEMICONT     = 'S';
/// <summary>
/// Semi-integer variable.
/// </summary>
public const char SEMIINT      = 'N';

// Objective sense

/// <summary>
/// Minimization objective.
/// </summary>
public const int MINIMIZE = 1;
/// <summary>
/// Maximization objective.
/// </summary>
public const int MAXIMIZE = -1;

// SOS types

/// <summary>
/// SOS of type 1.
/// </summary>
public const int SOS_TYPE1      = 1;
/// <summary>
/// SOS of type 2.
/// </summary>
public const int SOS_TYPE2      = 2;

// General constraint types

/// <summary>
/// General constraint maximum.
/// </summary>
public const int GENCONSTR_MAX       = 0;
/// <summary>
/// General constraint minimum.
/// </summary>
public const int GENCONSTR_MIN       = 1;
/// <summary>
```

```
/// General constraint absolute value.
/// </summary>
public const int GENCONSTR_ABS       = 2;
/// <summary>
/// General constraint logical AND.
/// </summary>
public const int GENCONSTR_AND       = 3;
/// <summary>
/// General constraint logical OR.
/// </summary>
public const int GENCONSTR_OR        = 4;
/// <summary>
/// General constraint norm.
/// </summary>
public const int GENCONSTR_NORM      = 5;
/// <summary>
/// General constraint indicator.
/// </summary>
public const int GENCONSTR_INDICATOR = 6;
/// <summary>
/// General constraint piecewise-linear.
/// </summary>
public const int GENCONSTR_PWL       = 7;
/// <summary>
/// General constraint polynomial function.
/// </summary>
public const int GENCONSTR_POLY      = 8;
/// <summary>
/// General constraint natural exponential function.
/// </summary>
public const int GENCONSTR_EXP       = 9;
/// <summary>
/// General constraint exponential function.
/// </summary>
public const int GENCONSTR_EXPA      = 10;
/// <summary>
/// General constraint natural logarithmic function.
/// </summary>
public const int GENCONSTR_LOG       = 11;
/// <summary>
/// General constraint logarithmic function.
/// </summary>
public const int GENCONSTR_LOGA      = 12;
/// <summary>
/// General constraint power function.
/// </summary>
public const int GENCONSTR_POW       = 13;
/// <summary>
/// General constraint sine function.
/// </summary>
public const int GENCONSTR_SIN       = 14;
/// <summary>
/// General constraint cosine function.
/// </summary>
public const int GENCONSTR_COS       = 15;
/// <summary>
```

```
/// General constraint tangent function.
/// </summary>
public const int GENCONSTR_TAN       = 16;
/// <summary>
/// General constraint logistic function.
/// </summary>
public const int GENCONSTR_LOGISTIC  = 17;

// Basis status info

/// <summary>
/// Variable is basic.
/// </summary>
public const int BASIC           = 0;
/// <summary>
/// Variable is non-basic at lower bound.
/// </summary>
public const int NONBASIC_LOWER  = -1;
/// <summary>
/// Variable is non-basic at upper bound.
/// </summary>
public const int NONBASIC_UPPER  = -2;
/// <summary>
/// Variable is superbasic.
/// </summary>
public const int SUPERBASIC      = -3;

// Numeric constants

/// <summary>
/// Infinity value.
/// </summary>
public const double INFINITY    = 1e100;
/// <summary>
/// Undefined value.
/// </summary>
public const double UNDEFINED   = 1e101;
/// <summary>
/// Maximum integer value.
/// </summary>
public const int    MAXINT      = 2000000000;

// Limits

/// <summary>
/// Maximum string length.
/// </summary>
public const int MAX_STRLEN     = 512;
/// <summary>
/// Maximum name length.
/// </summary>
public const int MAX_NAMELEN    = 255;
/// <summary>
/// Maximum tag length.
/// </summary>
public const int MAX_TAGLEN     = 10240;
```

```
/// <summary>
/// Maximum concurrent threads.
/// </summary>
public const int MAX_CONCURRENT = 64;

// Other constants

/// <summary>
/// Default compute server port.
/// </summary>
public const int DEFAULT_CS_PORT = 61000;

// Version numbers

/// <summary>
/// Major number
/// </summary>
public const int VERSION_MAJOR     = 11;
/// <summary>
/// Minor number
/// </summary>
public const int VERSION_MINOR     = 0;
/// <summary>
/// Technical number
/// </summary>
public const int VERSION_TECHNICAL = 0;

// Callback constants

/// <summary>
/// Gurobi callback codes.
/// </summary>
public class Callback
{
  /// <summary>
  /// Periodic polling callback.
  /// </summary>
  public const int POLLING       =     0;

  /// <summary>
  /// Currently performing presolve.
  /// </summary>
  public const int PRESOLVE      =     1;

  /// <summary>
  /// Currently in simplex.
  /// </summary>
  public const int SIMPLEX       =     2;

  /// <summary>
  /// Currently in MIP.
  /// </summary>
  public const int MIP           =     3;

  /// <summary>
  /// Found a new MIP incumbent.
```

```
    /// </summary>
    public const int MIPSOL        =     4;

    /// <summary>
    /// Currently exploring a MIP node.
    /// </summary>
    public const int MIPNODE       =     5;

    /// <summary>
    /// Currently in barrier.
    /// </summary>
    public const int BARRIER       =     7;

    /// <summary>
    /// Printing a log message.
    /// </summary>
    public const int MESSAGE       =     6;

    /// <summary>
    /// Currently in multi-objective optimization.
    /// </summary>
    public const int MULTIOBJ      =     8;

    /// <summary>
    /// Currently computing an IIS.
    /// </summary>
    public const int IIS           =     9;

    /// <summary>
    /// Returns the number of columns removed by presolve to this point.
    /// </summary>
    public const int PRE_COLDEL      = 1000;
    /// <summary>
    /// Returns the number of rows removed by presolve to this point.
    /// </summary>
    public const int PRE_ROWDEL      = 1001;
    /// <summary>
    /// Returns the number of constraint senses changed by presolve to this
    /// point.
    /// </summary>
    public const int PRE_SENCHG      = 1002;
    /// <summary>
    /// Returns the number of bounds changed by presolve to this point.
    /// </summary>
    public const int PRE_BNDCHG      = 1003;
    /// <summary>
    /// Returns the number of coefficients changed by presolve to this point.
    /// </summary>
    public const int PRE_COECHG      = 1004;
    /// <summary>
    /// Returns the current simplex iteration count.
    /// </summary>
    public const int SPX_ITRCNT      = 2000;
    /// <summary>
    /// Returns the current simplex objective value.
    /// </summary>
```

```
public const int SPX_OBJVAL      = 2001;
/// <summary>
/// Returns the current simplex primal infeasibility.
/// </summary>
public const int SPX_PRIMINF     = 2002;
/// <summary>
/// Returns the current simplex dual infeasibility.
/// </summary>
public const int SPX_DUALINF     = 2003;
/// <summary>
/// Returns 1 if the model has been perturbed.
/// </summary>
public const int SPX_ISPERT      = 2004;
/// <summary>
/// Returns the current best objective value.
/// </summary>
public const int MIP_OBJBST      = 3000;
/// <summary>
/// Returns the current best objective bound.
/// </summary>
public const int MIP_OBJBND      = 3001;
/// <summary>
/// Returns the current explored node count.
/// </summary>
public const int MIP_NODCNT      = 3002;
/// <summary>
/// Returns the current solution count.
/// </summary>
public const int MIP_SOLCNT      = 3003;
/// <summary>
/// Returns the current cutting plane count.
/// </summary>
public const int MIP_CUTCNT      = 3004;
/// <summary>
/// Returns the current unexplored node count.
/// </summary>
public const int MIP_NODLFT      = 3005;
/// <summary>
/// Returns the current simplex iteration count.
/// </summary>
public const int MIP_ITRCNT      = 3006;
/// <summary>
/// For a multi-scenario model, returns the number of scenarios that are still open.
/// </summary>
public const int MIP_OPENSCENARIOS = 3007;
/// <summary>
/// Returns the current phase of the MIP algorithm.
/// </summary>
public const int MIP_PHASE         = 3008;

/// <summary>
/// Returns the new solution.
/// </summary>
public const int MIPSOL_SOL      = 4001;
/// <summary>
/// Returns the objective value for the new solution.
```

```
    /// </summary>
    public const int MIPSOL_OBJ      = 4002;
    /// <summary>
    /// Returns the current best objective value.
    /// </summary>
    public const int MIPSOL_OBJBST   = 4003;
    /// <summary>
    /// Returns the current best objective bound.
    /// </summary>
    public const int MIPSOL_OBJBND   = 4004;
    /// <summary>
    /// Returns the current explored node count.
    /// </summary>
    public const int MIPSOL_NODCNT   = 4005;
    /// <summary>
    /// Returns the current solution count.
    /// </summary>
    public const int MIPSOL_SOLCNT   = 4006;
    /// <summary>
    /// For a multi-scenario model, returns the number of scenarios that are still open.
    /// </summary>
    public const int MIPSOL_OPENSCENARIOS = 4007;
    /// <summary>
    /// Returns the current phase of the MIP algorithm.
    /// </summary>
    public const int MIPSOL_PHASE   = 4008;

    /// <summary>
    /// Returns the status of the current node relaxation.
    /// </summary>
    public const int MIPNODE_STATUS  = 5001;
    /// <summary>
    /// Returns the current node relaxation solution or ray.
    /// </summary>
    public const int MIPNODE_REL     = 5002;
    /// <summary>
    /// Returns the current best objective value.
    /// </summary>
    public const int MIPNODE_OBJBST  = 5003;
    /// <summary>
    /// Returns the current best objective bound.
    /// </summary>
    public const int MIPNODE_OBJBND  = 5004;
    /// <summary>
    /// Returns the current explored node count.
    /// </summary>
    public const int MIPNODE_NODCNT  = 5005;
    /// <summary>
    /// Returns the current solution count.
    /// </summary>
    public const int MIPNODE_SOLCNT  = 5006;
    /// <summary>
    /// Returns the branching variable for the current node.
    /// </summary>
    public const int MIPNODE_BRVAR  = 5007;
    /// <summary>
```

```
/// For a multi-scenario model, returns the number of scenarios that are still open.
/// </summary>
public const int MIPNODE_OPENSCENARIOS = 5008;
/// <summary>
/// Returns the current phase of the MIP algorithm.
/// </summary>
public const int MIPNODE_PHASE  = 5009;

/// <summary>
/// Returns the current barrier iteration count.
/// </summary>
public const int BARRIER_ITRCNT  = 7001;
/// <summary>
/// Returns the current barrier primal objective value.
/// </summary>
public const int BARRIER_PRIMOBJ = 7002;
/// <summary>
/// Returns the current barrier dual objective value.
/// </summary>
public const int BARRIER_DUALOBJ = 7003;
/// <summary>
/// Returns the current barrier primal infeasibility.
/// </summary>
public const int BARRIER_PRIMINF = 7004;
/// <summary>
/// Returns the current barrier dual infeasibility.
/// </summary>
public const int BARRIER_DUALINF = 7005;
/// <summary>
/// Returns the current barrier complementarity violation.
/// </summary>
public const int BARRIER_COMPL   = 7006;
/// <summary>
/// Returns the message that is being printed.
/// </summary>
public const int MSG_STRING      = 6001;
/// <summary>
/// Returns the elapsed solver runtime (in seconds).
/// </summary>
public const int RUNTIME         = 6002;
/// <summary>
/// Returns the elapsed solver work (in work units).
/// </summary>
public const int WORK            = 6003;
/// <summary>
/// Returns the number of objectives that have already been optimized.
/// </summary>
public const int MULTIOBJ_OBJCNT = 8001;
/// <summary>
/// Returns the current solution count.
/// </summary>
public const int MULTIOBJ_SOLCNT = 8002;
/// <summary>
/// Returns the new solution.
/// </summary>
public const int MULTIOBJ_SOL    = 8003;
```

```
    /// <summary>
    /// Returns the minimum number of constraints in the IIS.
    /// </summary>
    public const int IIS_CONSTRMIN   = 9001;
    /// <summary>
    /// Returns the maximum number of constraints in the IIS.
    /// </summary>
    public const int IIS_CONSTRMAX   = 9002;
    /// <summary>
    /// Returns the estimated number of constraints in the IIS.
    /// </summary>
    public const int IIS_CONSTRGUESS = 9003;
    /// <summary>
    /// Returns the minimum number of bounds in the IIS.
    /// </summary>
    public const int IIS_BOUNDMIN    = 9004;
    /// <summary>
    /// Returns the maximum number of bounds in the IIS.
    /// </summary>
    public const int IIS_BOUNDMAX    = 9005;
    /// <summary>
    /// Returns the estimated number of bounds in the IIS.
    /// </summary>
    public const int IIS_BOUNDGUESS  = 9006;
  }

  // Errors

  /// <summary>
  /// Gurobi error codes.
  /// </summary>
  public class Error
  {
    /// <summary>
    /// Available memory was exhausted.
    /// </summary>
    public const int OUT_OF_MEMORY            = 10001;

    /// <summary>
    /// NULL input value provided for a required argument.
    /// </summary>
    public const int NULL_ARGUMENT            = 10002;

    /// <summary>
    /// Invalid input value.
    /// </summary>
    public const int INVALID_ARGUMENT         = 10003;

    /// <summary>
    /// Tried to query or set an unknown attribute.
    /// </summary>
    public const int UNKNOWN_ATTRIBUTE        = 10004;

    /// <summary>
    /// Tried to query or set an attribute that could not be accessed.
    /// </summary>
```

```
public const int DATA_NOT_AVAILABLE       = 10005;

/// <summary>
/// Index for attribute query was out of range.
/// </summary>
public const int INDEX_OUT_OF_RANGE       = 10006;

/// <summary>
/// Tried to query or set an unknown parameter.
/// </summary>
public const int UNKNOWN_PARAMETER        = 10007;

/// <summary>
/// Tried to set a parameter to a value that is outside its valid range.
/// </summary>
public const int VALUE_OUT_OF_RANGE       = 10008;

/// <summary>
/// Failed to obtain a Gurobi license.
/// </summary>
public const int NO_LICENSE               = 10009;

/// <summary>
/// Attempted to solve a model that is larger than the limit for a
/// trial license.
/// </summary>
public const int SIZE_LIMIT_EXCEEDED      = 10010;

/// <summary>
/// Problem in callback.
/// </summary>
public const int CALLBACK                 = 10011;

/// <summary>
/// Failed to read the requested file.
/// </summary>
public const int FILE_READ                = 10012;

/// <summary>
/// Failed to write the requested file.
/// </summary>
public const int FILE_WRITE               = 10013;

/// <summary>
/// Numerical error during requested operation.
/// </summary>
public const int NUMERIC                  = 10014;

/// <summary>
/// Attempted to perform infeasibility analysis on a feasible model.
/// </summary>
public const int IIS_NOT_INFEASIBLE       = 10015;

/// <summary>
/// Requested operation not valid for a MIP model.
/// </summary>
```

```
public const int NOT_FOR_MIP              = 10016;

/// <summary>
/// Tried to access a model while optimization was in progress.
/// </summary>
public const int OPTIMIZATION_IN_PROGRESS = 10017;

/// <summary>
/// Constraint, variable, or SOS contained duplicate indices.
/// </summary>
public const int DUPLICATES               = 10018;

/// <summary>
/// Error in reading or writing a MIP node file.
/// </summary>
public const int NODEFILE                 = 10019;

/// <summary>
/// non-PSD Q matrix in the objective or in a quadratic constraint.
/// </summary>
public const int Q_NOT_PSD                = 10020;

/// <summary>
/// Equality quadratic constraints.
/// </summary>
public const int QCP_EQUALITY_CONSTRAINT  = 10021;

/// <summary>
/// Network error.
/// </summary>
public const int NETWORK                  = 10022;

/// <summary>
/// Job rejected from Compute Server queue.
/// </summary>
public const int JOB_REJECTED             = 10023;

/// <summary>
/// Operation is not supported in the current usage environment.
/// </summary>
public const int NOT_SUPPORTED            = 10024;

/// <summary>
/// Result is larger than return value allows.
/// </summary>
public const int EXCEED_2B_NONZEROS       = 10025;

/// <summary>
/// Problem with piecewise-linear objective function.
/// </summary>
public const int INVALID_PIECEWIE_OBJ     = 10026;

/// <summary>
/// Not allowed to change UpdateMode parameter once model
/// has been created.
/// </summary>
```

```
  public const int UPDATEMODE_CHANGE        = 10027;

  /// <summary>
  /// Problem launching Instant Cloud job.
  /// </summary>
  public const int CLOUD                    = 10028;

  /// <summary>
  /// An error occurred during model modification or update.
  /// </summary>
  public const int MODEL_MODIFICATION       = 10029;

  /// <summary>
  /// An error occured with the client-server application.
  /// </summary>
  public const int CSWORKER                 = 10030;

  /// <summary>
  /// Multi-model tuning invoked on models of different types.
  /// </summary>
  public const int TUNE_MODEL_TYPES         = 10031;

  /// <summary>
  /// Multi-model tuning invoked on models of different types.
  /// </summary>
  public const int SECURITY                 = 10032;

  /// <summary>
  /// Tried to access a constraint or variable that is not in the model.
  /// </summary>
  public const int NOT_IN_MODEL             = 20001;

  /// <summary>
  /// Failed to create the requested model.
  /// </summary>
  public const int FAILED_TO_CREATE_MODEL   = 20002;

  /// <summary>
  /// Internal Gurobi error.
  /// </summary>
  public const int INTERNAL                 = 20003;
}

// Cuts parameter values

/// <summary>
/// Constant for Cuts parameter -
/// choose cuts control automatically.
/// </summary>
public const int CUTS_AUTO           = -1;
/// <summary>
/// Constant for Cuts parameter -
/// shuts off cuts.
/// </summary>
public const int CUTS_OFF            = 0;
/// <summary>
```

```
/// Constant for Cuts parameter -
/// moderate cut generation.
/// </summary>
public const int CUTS_CONSERVATIVE   = 1;
/// <summary>
/// Constant for Cuts parameter -
/// aggressive cut generation.
/// </summary>
public const int CUTS_AGGRESSIVE     = 2;
/// <summary>
/// Constant for Cuts parameter -
/// very aggressive cut generation.
/// </summary>
public const int CUTS_VERYAGGRESSIVE = 3;

// Presolve parameter values

/// <summary>
/// Constant for Presolve parameter -
/// automatic presolve setting.
/// </summary>
public const int PRESOLVE_AUTO         = -1;
/// <summary>
/// Constant for Presolve parameter -
/// turns off presolve.
/// </summary>
public const int PRESOLVE_OFF          = 0;
/// <summary>
/// Constant for Presolve parameter -
/// moderate presolve setting.
/// </summary>
public const int PRESOLVE_CONSERVATIVE = 1;
/// <summary>
/// Constant for Presolve parameter -
/// aggressive presolve setting.
/// </summary>
public const int PRESOLVE_AGGRESSIVE   = 2;

// Method parameter values

/// <summary>
/// Constant for Method parameter -
/// choose method automatically.
/// </summary>
public const int METHOD_NONE                           = -1;
public const int METHOD_AUTO                           = -1;
/// <summary>
/// Constant for Method and NodeMethod parameters -
/// use primal simplex.
/// </summary>
public const int METHOD_PRIMAL                         = 0;
/// <summary>
/// Constant for Method and NodeMethod parameters -
/// use dual simplex.
/// </summary>
public const int METHOD_DUAL                           = 1;
```

GRB.CharAttr

This enum is used to get or set char-valued attributes (through GRBModel.Get or GRBModel.Set). Please refer to the Attributes section to see a list of all char attributes and their functions.

GRB.DoubleAttr

This enum is used to get or set double-valued attributes (through GRBModel.Get or GRBModel.Set). Please refer to the Attributes section to see a list of all double attributes and their functions.

GRB.DoubleParam

This enum is used to get or set double-valued parameters (through GRBModel.Get, GRBModel.Set, GRBEnv.Get, or GRBEnv.Set). Please refer to the Parameters section to see a list of all double parameters and their functions.

GRB.IntAttr

This enum is used to get or set int-valued attributes (through GRBModel.Get or GRBModel.Set). Please refer to the Attributes section to see a list of all int attributes and their functions.

GRB.IntParam

This enum is used to get or set int-valued parameters (through GRBModel.Get, GRBModel.Set, GRBEnv.Get, or GRBEnv.Set). Please refer to the Parameters section to see a list of all int parameters and their functions.

GRB.StringAttr

This enum is used to get or set string-valued attributes (through GRBModel.Get or GRBModel.Set). Please refer to the Attributes section to see a list of all string attributes and their functions.

GRB.StringParam

This enum is used to get or set string-valued parameters (through GRBModel.Get, GRBModel.Set, GRBEnv.Get, or GRBEnv.Set). Please refer to the Parameters section to see a list of all string parameters and their functions.

Python API Overview

This section documents the Gurobi Python interface. It begins with an overview of the global functions, which can be called without referencing any Python objects. It then discusses the different types of objects that are available in the interface, and the most important methods on those objects. Finally, it gives a comprehensive presentation of all of the available classes and methods.

If you are new to the Gurobi Optimizer, we suggest that you start with the Getting Started Knowledge Base article or the Example Tour. These documents provide concrete examples of how to use the classes and methods described here.

Important note for AIX users: due to limited Python support on AIX, our AIX port does not include the Python interface.

Global Functions

The Gurobi shell contains a set of Global Functions that can be called without referring to any Gurobi objects. The most important of these functions is probably the read function, which allows you to read a model from a file. Another useful global function is disposeDefaultEnv, which disposes of the default environment. Other global functions allow you to read, modify, or write Gurobi parameters (readParams, setParam, and writeParams).

Models

Most actions in the Gurobi Python interface are performed by calling methods on Gurobi objects. The most commonly used object is the Model. A model consists of a set of decision variables (objects of class Var or MVar), a linear or quadratic objective function on these variables (specified using Model.setObjective), and a set of constraints on these variables (objects of class Constr, MConstr, QConstr, MQConstr, SOS, GenConstr, or MGenConstr). Each variable has an associated lower bound, upper bound, and type (continuous, binary, etc.). Each linear or quadratic constraint has an associated sense (less-than-or-equal, greater-than-or-equal, or equal), and right-hand side value. Refer to this section for more information on variables, constraints, and objectives.

An optimization model may be specified all at once, by loading the model from a file (using the previously mentioned read function), or it may be built incrementally, by first constructing an empty object of class Model and then subsequently calling Model.addVar, Model.addVars, or Model.addMVar to add additional variables, and Model.addConstr, Model.addConstrs, Model.addLConstr, Model.addQConstr, Model.addSOS, or any of the Model.addGenConstrXxx methods to add additional constraints.

Linear constraints are specified by building linear expressions (objects of class LinExpr or MLinExpr), and then specifying relationships between these expressions (for example, requiring that one expression be equal to another). Quadratic constraints are built in a similar fashion, but using quadratic expressions (objects of class QuadExpr or MQuadExpr) instead. General constraints are built using a set of dedicated methods, or a set of general constraint helper functions plus overloaded operators.

Models are dynamic entities; you can always add or remove variables or constraints.

We often refer to the *class* of an optimization model. At the highest level, a model can be continuous or discrete, depending on whether the modeling elements present in the model require discrete decisions to be made. Among continuous models...

- A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*.
- If the objective is quadratic, the model is a *Quadratic Program (QP)*.
- If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We sometimes refer to a few special cases of QCP: QCPs with convex constraints, QCPs with non-convex constraints, *bilinear programs*, and *Second-Order Cone Programs (SOCP)*.
- If any of the constraints are non-linear (chosen from among the available general constraints), the model is a *Non-Linear Program (NLP)*.

A model that contains any integer variables, semi-continuous variables, semi-integer variables, Special Ordered Set (SOS) constraints, or general constraints, is discrete, and is referred to as a *Mixed Integer Program (MIP)*. The special cases of MIP, which are the discrete versions of the continuous models types we've already described, are...

- *Mixed Integer Linear Programs (MILP)*
- *Mixed Integer Quadratic Programs (MIQP)*
- *Mixed Integer Quadratically-Constrained Programs (MIQCP)*
- *Mixed Integer Second-Order Cone Programs (MISOCP)*
- *Mixed Integer Non-Linear Programs (MINLP)*

The Gurobi Optimizer handles all of these model classes. Note that the boundaries between them aren't as clear as one might like, because we are often able to transform a model from one class to a simpler class.

Environments

Environments play a much smaller role in the Gurobi Python interface than they do in our other language APIs, mainly because the Python interface has a default environment. Unless you explicitly pass your own environment to routines that require an environment, the default environment will be used.

The main situation where you may want to create your own environment is when you want precise control over when the resources associated with an environment (specifically, a licensing token or a Compute Server) are released. If you use your own environment to create models (using read or the Model constructor), then the resources associated with the environment will be released as soon your program no longer references your environment or any models created with that environment.

Note that you can manually remove the reference to the default environment, thus making it available for garbage collection, by calling disposeDefaultEnv. After calling this, and after all models built within the default environment are garbage collected, the default environment will be

garbage collected as well. A new default environment will be created automatically if you call a routine that needs one.

For more advanced use cases, you can use an empty environment to create an uninitialized environment and then, programmatically, set all required options for your specific requirements. For further details see the Environment section.

Solving a Model

Once you have built a model, you can call Model.optimize to compute a solution. By default, optimize will use the concurrent optimizer to solve LP models, the barrier algorithm to solve QP models with convex objectives and QCP models with convex constraints, and the branch-and-cut algorithm otherwise. The solution is stored in a set of *attributes* of the model, which can be subsequently queried (we will return to this topic shortly).

The Gurobi algorithms keep careful track of the state of the model, so calls to Model.optimize will only perform further optimization if relevant data has changed since the model was last optimized. If you would like to discard previously computed solution information and restart the optimization from scratch without changing the model, you can call Model.reset.

After a MIP model has been solved, you can call Model.fixed to compute the associated *fixed* model. This model is identical to the original, except that the integer variables are fixed to their values in the MIP solution. If your model contains SOS constraints, some continuous variables that appear in these constraints may be fixed as well. In some applications, it can be useful to compute information on this fixed model (e.g., dual variables, sensitivity information, etc.), although you should be careful in how you interpret this information.

Multiple Solutions, Objectives, and Scenarios

By default, the Gurobi Optimizer assumes that your goal is to find one proven optimal solution to a single model with a single objective function. Gurobi provides the following features that allow you to relax these assumptions:

- Solution Pool: Allows you to find more solutions.
- Multiple Scenarios: Allows you to find solutions to multiple, related models.
- Multiple Objectives: Allows you to specify multiple objective functions and control the trade-off between them.

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call Model.computeIIS to compute an Irreducible Inconsistent Subsystem (IIS). This method can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. This method populates a set of IIS attributes.

To attempt to repair an infeasibility, call Model.feasRelaxS or Model.feasRelax to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Querying and Modifying Attributes

Most of the information associated with a Gurobi model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. To give a simple example, solving an optimization model causes the `x` variable attribute to be populated. Attributes such as `x` that are computed by the Gurobi Optimizer cannot be modified directly by the user, while others, such as the variable lower bound (the `lb` attribute) can.

Attributes can be accessed in two ways in the Python interface. The first is to use the `getAttr()` and `setAttr()` methods, which are available on variables (Var.getAttr/ Var.setAttr), matrix variables (MVar.getAttr/ MVar.setAttr), linear constraints (Constr.getAttr/ Constr.setAttr), matrix constraints (MConstr.getAttr/ MConstr.setAttr), quadratic constraints (QConstr.getAttr/ QConstr.setAttr), matrix constraints (MQConstr.getAttr/ MQConstr.setAttr), SOSs (SOS.getAttr), general constraints (GenConstr.getAttr/ GenConstr.setAttr), and models (Model.getAttr/ Model.setAttr). These are called with the attribute name as the first argument (e.g., `var.getAttr("x")` or `constr.setAttr("rhs", 0.0)`). The full list of available attributes can be found in the Attributes section of this manual.

Attributes can also be accessed more directly: you can follow an object name by a period, followed by the name of an attribute of that object. Note that upper/lower case is ignored when referring to attributes. Thus, `b = constr.rhs` is equivalent to `b = constr.getAttr("rhs")`, and `constr.rhs = 0.0` is equivalent to `constr.setAttr("rhs", 0.0)`.

Additional Model Modification Information

Most modifications to an existing model are done through the attribute interface (e.g., changes to variable bounds, constraint right-hand sides, etc.). The main exceptions are modifications to the constraint matrix and to the objective function.

The constraint matrix can be modified in a few ways. The first is to call the Model.chgCoeff method. This method can be used to modify the value of an existing non-zero, to set an existing non-zero to zero, or to create a new non-zero. The constraint matrix is also modified when you remove a variable or constraint from the model (through the Model.remove method). The non-zero values associated with the deleted constraint or variable are removed along with the constraint or variable itself.

The model objective function can also be modified in a few ways. The easiest is to build an expression that captures the objective function (a LinExpr, MLinExpr, QuadExpr, or MQuadExpr object), and then pass that expression to method Model.setObjective. If you wish to modify the objective, you can simply call `setObjective` again with a new `LinExpr` or `QuadExpr` object.

For linear objective functions, an alternative to `setObjective` is to use the `Obj` variable attribute to modify individual linear objective coefficients.

If your variables have piecewise-linear objectives, you can specify them using the Model.setPWLObj method. Call this method once for each relevant variable. The Gurobi simplex solver includes algorithmic support for convex piecewise-linear objective functions, so for continuous models you should see a substantial performance benefit from using this feature. To clear a previously specified piecewise-linear objective function, simply set the `Obj` attribute on the corresponding variable to 0.

Lazy Updates

One important item to note about model modification in the Gurobi optimizer is that it is performed in a *lazy* fashion, meaning that modifications don't affect the model immediately. Rather, they are queued and applied later. If your program simply creates a model and solves it, you will probably never notice this behavior. However, if you ask for information about the model before your modifications have been applied, the details of the lazy update approach may be relevant to you.

As we just noted, model modifications (bound changes, right-hand side changes, objective changes, etc.) are placed in a queue. These queued modifications can be applied to the model in three different ways. The first is by an explicit call to Model.update. The second is by a call to Model.optimize. The third is by a call to Model.write to write out the model. The first case gives you fine-grained control over when modifications are applied. The second and third make the assumption that you want all pending modifications to be applied before you optimize your model or write it to disk.

Why does the Gurobi interface behave in this manner? There are a few reasons. The first is that this approach makes it much easier to perform multiple modifications to a model, since the model remains unchanged between modifications. The second is that processing model modifications can be expensive, particularly in a Compute Server environment, where modifications require communication between machines. Thus, it is useful to have visibility into exactly when these modifications are applied. In general, if your program needs to make multiple modifications to the model, you should aim to make them in phases, where you make a set of modifications, then update, then make more modifications, then update again, etc. Updating after each individual modification can be extremely expensive.

If you forget to call update, your program won't crash. Your query will simply return the value of the requested data from the point of the last update. If the object you tried to query didn't exist then, you'll get a `NOT_IN_MODEL` exception instead.

The semantics of lazy updates have changed since earlier Gurobi versions. While the vast majority of programs are unaffected by this change, you can use the UpdateMode parameter to revert to the earlier behavior if you run into an issue.

Managing Parameters

The Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization. Parameters are set using method Model.setParam. Current values may also be retrieved with Model.getParamInfo. You can also access parameters more directly through the `Model.Params` class. To set the MIPGap parameter to 0.0 for model `m`, for example, you can do either `m.setParam('MIPGap', 0)` or `m.Params.MIPGap = 0`.

You can read a set of parameter settings from a file using Model.read, or write the set of changed parameters using Model.write.

We also include an automated parameter tuning tool that explores many different sets of parameter changes in order to find a set that improves performance. You can call Model.tune to invoke the tuning tool on a model. Refer to the parameter tuning tool section for more information.

One thing we should note is that changing a parameter for one model has no effect on the parameter value for other models. Use the global setParam method to set a parameter for all

loaded models.

The full list of Gurobi parameters can be found in the Parameters section.

Monitoring Progress - Logging and Callbacks

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging behavior. You can set the LogFile parameter if you wish to also direct the Gurobi log to a file. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter.

Log output is also sent to a Python logger named `gurobipy`, at level `INFO`. You can use the Python `logging` module to connect to this log.

More detailed progress monitoring can be done through a callback function. If you pass a function taking two arguments, `model` and `where`, to Model.optimize, your function will be called periodically from within the optimization. Your callback can then call Model.cbGet to retrieve additional information on the state of the optimization. You can refer to the Callback class for additional information.

Modifying Solver Behavior - Callbacks

Callbacks can also be used to modify the behavior of the Gurobi optimizer. The simplest control callback is Model.terminate, which asks the optimizer to terminate at the earliest convenient point. Method Model.cbSetSolution allows you to inject a feasible solution (or partial solution) during the solution of a MIP model. Methods Model.cbCut and Model.cbLazy allow you to add *cutting planes* and *lazy constraints* during a MIP optimization, respectively. Method Model.cbStopOneMultiObj allows you to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

Batch Optimization

Gurobi Compute Server enables programs to offload optimization computations onto dedicated servers. The Gurobi Cluster Manager adds a number of additional capabilities on top of this. One important one, *batch optimization*, allows you to build an optimization model with your client program, submit it to a Compute Server cluster (through the Cluster Manager), and later check on the status of the model and retrieve its solution. You can use a Batch object to make it easier to work with batches. For details on batches, please refer to the Batch Optimization section.

Error Handling

All of the methods in the Gurobi Python library can throw an exception of type GurobiError. When an exception occurs, additional information on the error can be obtained by retrieving the `errno` or `message` members of the `GurobiError` object. A list of possible values for the `errno` field can be found in the Error Code section.

7.1 Global Functions

Gurobi global functions. These functions can be accessed from the main Gurobi shell prompt. In contrast to all other methods in the Gurobi Python interface, these functions do not require a Gurobi object to invoke them.

disposeDefaultEnv()

```
disposeDefaultEnv ( )
```

Dispose of the default environment.

Calling this function releases the default environment created by the Gurobi Python module. This function is particularly useful in a long-running Python session (e.g., within a Jupyter notebook), where the Gurobi environment would otherwise continue to exist for the full duration of the session.

Note that models built with the default environment must be garbage collected before the default environment can be freed. You can force a model `m` to be garbage collected with the statement `del m`. A call to `disposeDefaultEnv` prints the message

```
Freeing default Gurobi environment
```

and then attempts to free the default environment. Note however that this attempt may fail if some models referencing the default environment have not yet been disposed of. If that is the case, the default environment will be automatically freed when the last model referencing it is disposed of.

Example usage:

```
disposeDefaultEnv()
```

multidict()

```
multidict ( data )
```

This function splits a single dictionary into multiple dictionaries. The input dictionary should map each key to a list of `n` values. The function returns a list of the shared keys as its first result, followed by the `n` individual Gurobi tuple dictionaries (stored as tupledict objects).

Arguments:

`data`: A Python dictionary. Each key should map to a list of values.

Return value:

A list, where the first member contains the shared key values, and the following members contain the dictionaries that result from splitting the value lists from the input dictionary.

Example usage:

```
keys, dict1, dict2 = multidict( {
  'key1': [1, 2],
  'key2': [1, 3],
  'key3': [1, 4] } )
```

paramHelp()

```
paramHelp  ( paramname )
```

Obtain help about a Gurobi parameter.

Arguments:

paramname: String containing the name of parameter that you would like help with. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed. Note that case is ignored.

Example usage:

```
paramHelp("Cuts")
paramHelp("Heu*")
paramHelp("*cuts")
```

quicksum()

```
quicksum  ( data )
```

A version of the Python **sum** function that is much more efficient for building large Gurobi expressions (LinExpr or QuadExpr objects). The function takes a list of terms as its argument.

Note that while **quicksum** is much faster than **sum**, it isn't the fastest approach for building a large expression. Use addTerms or the LinExpr() constructor if you want the quickest possible expression construction.

Arguments:

data: List of terms to add. The terms can be constants, Var objects, LinExpr objects, or QuadExpr objects.

Return value:

An expression that represents the sum of the terms in the input list.

Example usage:

```
expr = quicksum([2*x, 3*y+1, 4*z*z])
expr = quicksum(model.getVars())
```

read()

```
read  ( filename, env=defaultEnv )
```

Read a model from a file.

Arguments:

filename: Name of file containing model. Note that the type of the file is encoded in the file name suffix. Valid suffixes are **.mps**, **.rew**, **.lp**, **.rlp**, **.dua**, **.dlp**, **.ilp**, or **.opb**. The files can be compressed, so additional suffixes of **.gz**, **.bz2**, **.zip**, or **.7z** are accepted. The file name may contain ***** or **?** wildcards. No file is read when no wildcard match is found. If more than one match is found, this routine will attempt to read the first matching file.

env: Environment in which to create the model. Creating your environment (using the Env constructor) gives you more control over Gurobi licensing, but it can make your program

more complex. Use the default environment unless you know that you need to control your own environments.

Return value:

Model object containing the model that was read from the input file.

Example usage:

```
m = read("afiro.mps")
m.optimize()
```

readParams()

```
readParams  ( filename )
```

Read a set of parameter settings from a file. The file name must end in `.prm`, and the file must be in PRM format.

Arguments:

`filename`: Name of file containing parameter settings.

Example usage:

```
readParams("params.prm")
```

resetParams()

```
resetParams  (  )
```

Reset the values of all parameters to their default values. Note that existing models that are stored inside Python data structures (lists, dictionaries, etc.), or inside user classes aren't affected.

Example usage:

```
resetParams()
```

setParam()

```
setParam  ( paramname, newvalue )
```

Set the value of a parameter to a new value. Note that existing models that are stored inside Python data structures (lists, dictionaries, etc.), or inside user classes aren't affected.

Arguments:

`paramname`: String containing the name of parameter that you would like to modify. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed and none are modified. Note that case is ignored.

`newvalue`: Desired new value for parameter. Can be 'default', which indicates that the parameter should be reset to its default value.

Example usage:

```
setParam("Cuts", 2)
setParam("Heu*", 0.5)
setParam("*Interval", 10)
```

writeParams()

```
writeParams  ( filename )
```

Write all modified parameters to a file. The file is written in PRM format.

Example usage:

```
setParam("Heu*", 0.5)
writeParams("params.prm")  # file will contain changed parameter
system("cat params.prm")
```

7.2 Model

Gurobi model object. Commonly used methods on the model object in the Gurobi shell include optimize (optimizes the model), printStats (prints statistics about the model), printAttr (prints the values of an attribute), and write (writes information about the model to a file). Commonly used methods when building a model include addVar (adds a new variable), addVars (adds multiple new variables), addMVar (adds an a NumPy ndarray of Gurobi variables), addConstr (adds a new constraint), and addConstrs (adds multiple new constraints).

Model()

```
Model  ( name="", env=defaultEnv )
```

Model constructor.

Arguments:

name: Name of new model. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

env: Environment in which to create the model. Creating your environment (using the Env constructor) gives you more control over Gurobi licensing, but it can make your program more complex. Use the default environment unless you know that you need to control your own environments.

Return value:

New model object. Model initially contains no variables or constraints.

Example usage:

```
m = Model("NewModel")
x0 = m.addVar()

env = Env("my.log")
m2 = Model("NewModel2", env)
```

Model.addConstr()

```
addConstr  ( constr, name="" )
```

Add a constraint to a model.

This method accepts a TempConstr as its first argument, and the constraint name as its optional second argument. You use operator overloading to create the argument (see this section for details). This one method allows you to add linear constraints, matrix constraints, quadratic constraints, and general constraints.

Arguments:

constr: TempConstr argument.

name: Name for new constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII

character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New constraint object. This can be a Constr, an MConstr, a QConstr, or an MQConstr, depending on the type of the argument.

Example usage:

```
model.addConstr(x + y <= 2.0, "c1")
model.addConstr(x*x + y*y <= 4.0, "qc0")
model.addConstr(x + y + z == [1, 2], "rgc0")
model.addConstr(A @ t >= b)
model.addConstr(z == and_(x, y, w), "gc0")
model.addConstr(z == min_(x, y), "gc1")
model.addConstr((w == 1) >> (x + y <= 1), "ic0")
```

Warning: A constraint can only have a single comparison operator. While `1 <= x + y <= 2` may look like a valid constraint, `addConstr` won't accept it.

Model.addConstrs()

```
addConstrs ( generator, name="" )
```

Add multiple constraints to a model using a Python generator expression. Returns a Gurobi tupledict that contains the newly created constraints, indexed by the values generated by the generator expression.

The first argument to `addConstrs` is a Python generator expression, a special feature of the Python language that allows you to iterate over a Python expression. In this case, the Python expression will be a Gurobi constraint and the generator expression provides values to plug into that constraint. A new Gurobi constraint is added to the model for each iteration of the generator expression.

To give an example, if `x` is a Gurobi variable, then

```
m.addConstr(x <= 1, name='c0')
```

would add a single linear constraint involving this variable. In contrast, if `x` is a list of Gurobi variables, then

```
m.addConstrs((x[i] <= 1 for i in range(4)), name='c')
```

would add four constraints to the model. The entire first argument is a generator expression, where the indexing is controlled by the statement `for i in range(4)`, The first constraint that results from this expression would be named `c[0]`, and would involve variable `x[0]`. The second would be named `c[1]`, and would involve variable `x[1]`.

Generator expressions can be much more complex than this. They can involve multiple variables and conditional tests. For example, you could do:

```
m.addConstrs((x[i,j] == 0 for i in range(4)
                          for j in range(4)
                          if i != j), name='c')
```

One restriction that `addConstrs` places on the generator expression is that each variable must always take a scalar value (`int`, `float`, `string`, ...). Thus, `for i in [1, 2.0, 'a', 'bc']` is fine, but `for i in [(1, 2), [1, 2, 3]]` isn't.

This method can be used to add linear constraints, quadratic constraints, or general constraints to the model. Refer to the TempConstr documentation for more information on all of the different constraint types that can be added.

Note that if you supply a name argument, the generator expression must be enclosed in parenthesis. This requirement comes from the Python language interpreter.

Arguments:

`generator`: A generator expression, where each iteration produces a constraint.

`name`: Name pattern for new constraints. The given name will be subscripted by the index of the generator expression, so if the index is an integer, `c` would become `c[0]`, `c[1]`, etc. Note that the generated names will be stored as ASCII strings, so you should avoid using names that contain non-ASCII characters. In addition, names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

A dictionary of Constr objects, indexed by the values specified by the generator expression.

Example usage:

```
model.addConstrs(x.sum(i, '*') <= capacity[i] for i in range(5))
model.addConstrs(x[i] + x[j] <= 1 for i in range(5) for j in range(5))
model.addConstrs(x[i]*x[i] + y[i]*y[i] <= 1 for i in range(5))
model.addConstrs(x.sum(i, '*') == [0, 2] for i in [1, 2, 4])
model.addConstrs(z[i] == max_(x[i], y[i]) for i in range(5))
model.addConstrs((x[i] == 1) >> (y[i] + z[i] <= 5) for i in range(5))
```

Warning: A constraint can only have a single comparison operator. While `1 <= x + y <= 2` may look like a valid constraint, `addConstr` won't accept it.

Model.addGenConstrXxx()

Each of the functions described below adds a new *general constraint* to a model.

Mathematical programming has traditionally defined a set of fundamental constraint types: variable bound constraints, linear constraints, quadratic constraints, integrality constraints, and SOS constraints. These are typically treated directly by the underlying solver (although not always), and are fundamental to the overall algorithm.

Gurobi accepts a number of additional constraint types, which we collectively refer to as *general (function) constraints*. These are typically *not* treated directly by the solver. Rather, they are transformed by presolve into constraints (and variables) chosen from among the fundamental types listed above. In some cases, the resulting constraint or constraints are mathematically equivalent to the original; in others, they are approximations. If such constraints appear in your model, but if you prefer to reformulate them yourself using fundamental constraint types instead, you can certainly do so. However, note that Gurobi can sometimes exploit information contained in the other constraints in the model to build a more efficient formulation than what you might create.

The additional constraint types that fall under this *general constraint* umbrella are:

- addGenConstrMax: $y = \max(x_1, x_2, ..., c)$

- addGenConstrMin: $y = \min(x_1, x_2, ..., c)$
- addGenConstrAbs: $y = |x|$
- addGenConstrAnd: $y = x_1 \wedge x_2 \wedge x_3 ...$
- addGenConstrOr: $y = x_1 \vee x_2 \vee x_3 ...$
- addGenConstrNorm: $y = norm(x_1, x_2, x_3 ...)$
- addGenConstrIndicator: $y = 1 \rightarrow a'x \leq b$ (an indicator constraint)
- addGenConstrPWL: $y = pwl(x)$ (a piecewise-linear function, specified using breakpoints)
- addGenConstrPoly: $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$
- addGenConstrExp: $y = e^x$
- addGenConstrExpA: $y = a^x$
- addGenConstrLog: $y = \log_e(x)$
- addGenConstrLogA: $y = \log_a(x)$
- addGenConstrLogistic: $y = \frac{1}{1+e^{-x}}$
- addGenConstrPow: $y = x^a$
- addGenConstrSin: $y = \sin(x)$
- addGenConstrCos: $y = \cos(x)$
- addGenConstrTan: $y = \tan(x)$

You can also add several types of general constraints through addConstr or addConstrs, using overloaded operators and a few general constraint helper functions. The descriptions below will make note of these equivalent, more concise alternatives.

Please refer to this section for additional details on general constraints.

Model.addGenConstrMax()

```
addGenConstrMax  ( resvar, vars, constant=None, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_MAX` to a model.

A MAX constraint $r = \max\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the maximum of the operand variables $x_1, \ldots, x_n$ and the constant c.

You can also add a MAX constraint using the max_ function.

Arguments:

`resvar (Var)`: The variable whose value will be equal to the maximum of the other variables.

`vars (list of Var, or tupledict of Var values)`: The variables over which the MAX will be taken.

constant (float, optional): The constant to include among the arguments of the MAX operation.

name (string, optional): Name for the new general constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = max(x1, x3, x4, 2.0)
model.addGenConstrMax(x5, [x1, x3, x4], 2.0, "maxconstr")

# overloaded forms
model.addConstr(x5 == max_([x1, x3, x4], constant=2.0), name="maxconstr")
model.addConstr(x5 == max_(x1, x3, x4, constant=2.0), name="maxconstr")
```

Model.addGenConstrMin()

```
addGenConstrMin  ( resvar, vars, constant=None, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_MIN` to a model.

A MIN constraint $r = \min\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the minimum of the operand variables $x_1, \ldots, x_n$ and the constant c.

You can also add a MIN constraint using the min_ function.

Arguments:

resvar (Var): The variable whose value will be equal to the minimum of the other variables.

vars (list of Var, or tupledict of Var values): The variables over which the MIN will be taken.

constant (float, optional): The constant to include among the arguments of the MIN operation.

name (string, optional): Name for the new general constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = min(x1, x3, x4, 2.0)
model.addGenConstrMin(x5, [x1, x3, x4], 2.0, "minconstr")

# overloaded forms
model.addConstr(x5 == min_([x1, x3, x4], constant=2.0), name="minconstr")
model.addConstr(x5 == min_(x1, x3, x4, constant=2.0), name="minconstr")
```

Model.addGenConstrAbs()

```
addGenConstrAbs  ( resvar, argvar, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_ABS` to a model.

An ABS constraint $r = \text{abs}\{x\}$ states that the resultant variable r should be equal to the absolute value of the argument variable x.

You can also add an ABS constraint using the abs_ function.

Arguments:

resvar (Var): The variable whose value will be to equal the absolute value of the argument variable.

argvar (Var): The variable for which the absolute value will be taken.

name (string, optional): Name for the new general constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = abs(x1)
model.addGenConstrAbs(x5, x1, "absconstr")

# overloaded form
model.addConstr(x5 == abs_(x1), name="absconstr")
```

Model.addGenConstrAnd()

```
addGenConstrAnd  ( resvar, vars, name="" )
```

Add a new general constraint of type GRB.GENCONSTR_AND to a model.

An AND constraint $r = \text{and}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if all of the operand variables $x_1, \ldots, x_n$ are equal to 1. If any of the operand variables is 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

You can also add an AND constraint using the and_ function.

Arguments:

resvar (Var): The variable whose value will be equal to the AND concatenation of the other variables.

vars (list of Var): The variables over which the AND concatenation will be taken.

name (string, optional): Name for the new general constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = and(x1, x3, x4)
model.addGenConstrAnd(x5, [x1, x3, x4], "andconstr")

# overloaded forms
model.addConstr(x5 == and_([x1, x3, x4]), "andconstr")
model.addConstr(x5 == and_(x1, x3, x4), "andconstr")
```

Model.addGenConstrOr()

`addGenConstrOr ( resvar, vars, name="" )`

Add a new general constraint of type `GRB.GENCONSTR_OR` to a model.

An OR constraint $r = \text{or}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if any of the operand variables $x_1, \ldots, x_n$ is equal to 1. If all operand variables are 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

You can also add an OR constraint using the or_ function.

Arguments:

`resvar (Var)`: The variable whose value will be equal to the OR concatenation of the other variables.

`vars (list of Var)`: The variables over which the OR concatenation will be taken.

`name (string, optional)`: Name for the new general constraint. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = or(x1, x3, x4)
model.addGenConstrOr(x5, [x1, x3, x4], "orconstr")

# overloaded forms
model.addConstr(x5 == or_([x1, x3, x4]), "orconstr")
model.addConstr(x5 == or_(x1, x3, x4), "orconstr")
```

Model.addGenConstrNorm()

`addGenConstrNorm ( resvar, vars, which, name="" )`

Add a new general constraint of type `GRB.GENCONSTR_NORM` to a model.

A NORM constraint $r = \text{norm}\{x_1, \ldots, x_n\}$ states that the resultant variable r should be equal to the vector norm of the argument vector $x_1, \ldots, x_n$.

Arguments:

`resvar (Var)`: The variable whose value will be equal to the vector norm of the other variables.

`vars (list of Var, or tupledict of Var values, or 1-dim MVar)`: The variables over which the vector norm will be taken. Note that this may not contain duplicates.

`which (float)`: Which norm to use. Options are 0, 1, 2, and any value greater than or equal to GRB.INFINITY.

`name (string, optional)`: Name for the new general constraint. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = 2-norm(x1, x3, x4)
model.addGenConstrNorm(x5, [x1, x3, x4], 2.0, "normconstr")
```

Model.addGenConstrIndicator()

```
addGenConstrIndicator ( binvar, binval, lhs, sense=None, rhs=None, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_INDICATOR` to a model.

An INDICATOR constraint $z = f \rightarrow a^T x \leq b$ states that if the binary indicator variable z is equal to f, where $f \in \{0, 1\}$, then the linear constraint $a^T x \leq b$ should hold. On the other hand, if $z = 1 - f$, the linear constraint may be violated. The sense of the linear constraint can also be specified to be $=$ or $\geq$.

Note that the indicator variable z of a constraint will be forced to be binary, independent of how it was created.

You can also add an INDICATOR constraint using a special overloaded syntax. See the examples below for details.

Multiple INDICATOR constraints can be added in a single `addGenConstrIndicator` call by using matrix-friendly modeling objects. In this case, an MGenConstr object will be returned. The input arguments follow NumPy's broadcasting rules, with some restrictions:

- `binvar` cannot be broadcasted over `binval`, and
- the linear constraints defined by `(lhs, sense, rhs)` cannot be broadcasted over the indicator variable.

This means that via broadcasting, you can use a single indicator variable to control whether multiple linear constraints should hold. We refer you to the NumPy documentation for further information regarding broadcasting behaviour.

Arguments:

`binvar (Var or MVar)`: The binary indicator variable or matrix variable.

`binval (Boolean or ndarray)`: The value for the binary indicator variable that would force the linear constraint to be satisfied. Can be provided as an `ndarray` of distinct values if `binvar` is an MVar.

`lhs (float, Var, LinExpr, MVar, MLinExpr, or TempConstr)`: Left-hand side expression for the linear constraint triggered by the indicator. Can be a constant, a Var, a LinExpr, an MVar, or an MLinExpr. Alternatively, a temporary constraint object can be used to define the linear constraint that is triggered by the indicator. The temporary constraint object is created using an overloaded comparison operator. See TempConstr for more information. In this case, the "sense" and "rhs" parameters must stay at their default values `None`.

`sense (char)`: Sense for the linear constraint. Options are `GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`.

`rhs (float or ndarray)`: Right-hand side value for the linear constraint. Can be provided as an `ndarray` of distinct values if `lhs` is an MVar or an MLinExpr.

`name (string, optional)`: Name for the new general constraint. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→'

can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New general constraint object. This can be a GenConstr or an MGenConstr depending on the types of the input arguments.

Example usage:

```
# x7 = 1 -> x1 + 2 x3 + x4 = 1
model.addGenConstrIndicator(x7, True, x1 + 2*x2 + x4, GRB.EQUAL, 1.0)

# alternative form
model.addGenConstrIndicator(x7, True, x1 + 2*x2 + x4 == 1.0)

# overloaded form
model.addConstr((x7 == 1) >> (x1 + 2*x2 + x4 == 1.0))

# Matrix-friendly form where Z is an MVar. Creates multiple
# indicator constraints, each specifying
#   z_i = 1 -> sum a_ij x_j = b_i.
model.addGenConstrIndicator(z, 1.0, A @ x == b)

# Matrix-friendly form where z is an Var. Creates multiple
# indicator constraints, each specifying
#   z = 1 -> sum a_ij x_j = b_i
# (the indicator variable is broadcasted).
model.addGenConstrIndicator(z, 1.0, A @ x == b)
```

Model.addGenConstrPWL()

```
addGenConstrPWL  ( xvar, yvar, xpts, ypts, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_PWL` to a model.

A piecewise-linear (PWL) constraint states that the relationship $y = f(x)$ must hold between variables x and y, where f is a piecewise-linear function. The breakpoints for f are provided as arguments. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`xpts (list of float)`: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

`ypts (list of float)`: The y values for the points that define the piecewise-linear function.

`name (string, optional)`: Name for the new general constraint.

Return value:

New general constraint.

Example usage:

```
gc = model.addGenConstrPWL(x, y, [0, 1, 2], [1.5, 0, 3], "myPWLConstr")
```

Model.addGenConstrPoly()

```
addGenConstrPoly  ( xvar, yvar, p, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_POLY` to a model.

A polynomial function constraint states that the relationship $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$ should hold between variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`p`: The coefficients for the polynomial function (starting with the coefficient for the highest power).

`name (string, optional)`: Name for the new general constraint.

`options (string, optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = 2 x^3 + 1.5 x^2 + 1
gc = model.addGenConstrPoly(x, y, [2, 1.5, 0, 1])
```

Model.addGenConstrExp()

```
addGenConstrExp  ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_EXP` to a model.

A natural exponential function constraint states that the relationship $y = exp(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

yvar (Var): The y variable.

name (string, optional): Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = exp(x)
gc = model.addGenConstrExp(x, y)
```

Model.addGenConstrExpA()

```
addGenConstrExpA  ( xvar, yvar, a, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_EXPA` to a model.

An exponential function constraint states that the relationship $y = a^x$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

xvar (Var): The x variable.

yvar (Var): The y variable.

a (float): The base of the function, $a > 0$.

name (string, optional): Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = 3^x
gc = model.addGenConstrExpA(x, y, 3.0, "expa")
```

Model.addGenConstrLog()

```
addGenConstrLog  ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_LOG` to a model.

A natural logarithmic function constraint states that the relationship $y = log(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

xvar (Var): The x variable.

yvar (Var): The y variable.

name (string, optional): Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = ln(x)
gc = model.addGenConstrLog(x, y)
```

Model.addGenConstrLogA()

```
addGenConstrLogA  ( xvar, yvar, a, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_LOGA` to a model.

A logarithmic function constraint states that the relationship $y = log_a(x)$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

xvar (Var): The x variable.

yvar (Var): The y variable.

a (float): The base of the function, $a > 0$.

name (string, optional): Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = log10(x)
gc = model.addGenConstrLogA(x, y, 10.0, "log10", "FuncPieces=-1 FuncPieceError=1e-5")
```

Model.addGenConstrLogistic()

```
addGenConstrLogistic  ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_LOGISTIC` to a model.

A logistic function constraint states that the relationship $y = \frac{1}{1+e^{-x}}$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`name (string, optional)`: Name for the new general constraint.

`options (string, optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = 1 / (1 + exp(-x))
gc = model.addGenConstrLogistic(x, y)
```

Model.addGenConstrPow()

```
addGenConstrPow  ( xvar, yvar, a, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_POW` to a model.

A power function constraint states that the relationship $y = x^a$ should hold for variables x and y, where a is the (constant) exponent.

If the exponent a is negative, the lower bound on x must be strictly positive. If the exponent isn't an integer, the lower bound on x must be non-negative.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

xvar (Var): The x variable.
yvar (Var): The y variable.
a (float): The exponent of the function.
name (string, optional): Name for the new general constraint.
options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = x^3.5
gc = model.addGenConstrPow(x, y, 3.5, "gf", "FuncPieces=1000")
```

Model.addGenConstrSin()

```
addGenConstrSin  ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_SIN` to a model.

A sine function constraint states that the relationship $y = sin(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

xvar (Var): The x variable.
yvar (Var): The y variable.
name (string, optional): Name for the new general constraint.
options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = sin(x)
gc = model.addGenConstrSin(x, y)
```

Model.addGenConstrCos()

```
addGenConstrCos  ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_COS` to a model.

A cosine function constraint states that the relationship $y = cos(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`name (string, optional)`: Name for the new general constraint.

`options (string, optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = cos(x)
gc = model.addGenConstrCos(x, y)
```

Model.addGenConstrTan()

```
addGenConstrTan  ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_TAN` to a model.

A tangent function constraint states that the relationship $y = tan(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`name (string, optional)`: Name for the new general constraint.

`options (string, optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = tan(x)
gc = model.addGenConstrTan(x, y)
```

Model.addLConstr()

```
addLConstr  ( lhs, sense=None, rhs=None, name="" )
```

Add a linear constraint to a model. This method is faster than addConstr() (as much as 50% faster for very sparse constraints), but can only be used to add linear constraints.

Note that this method also accepts a TempConstr as its first argument (with the name as its second argument). This allows you to use operator overloading to create constraints. See TempConstr for more information.

Arguments:

lhs: Left-hand side for the new constraint. Can be a constant, a Var, a LinExpr, or a TempConstr (while the TempConstr can only be of linear form).

sense: Sense for the new constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhs: Right-hand side for the new constraint. Can be a constant, a Var, or a LinExpr.

name: Name for new constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New constraint object.

Example usage:

```
model.addLConstr(x + 2*y, GRB.EQUAL, 3*z, "c0")
model.addLConstr(x + y <= 2.0, "c1")
model.addLConstr(LinExpr([1.0,1.0], [x,y]), GRB.LESS_EQUAL, 1)
```

Model.addMConstr()

```
addMConstr  ( A, x, sense, b, name="" )
```

Add a set of linear constraints to the model using matrix semantics. The added constraints are $Ax = b$ (except that the constraint sense is determined by the **sense** argument). The **A** argument must be a NumPy dense ndarray or a SciPy sparse matrix.

Note that you will typically use overloaded operators to build and add constraints using matrix semantics. The overloaded **@** operator can be used to build a linear matrix expression, which can then be used with an overloaded comparison operator to build a TempConstr object. This can then be passed to addConstr.

Arguments:

A: The constraint matrix - a NumPy 2-D dense ndarray or a SciPy sparse matrix.

x: Decision variables. Argument can be an MVar object, a list of Var objects, or **None** (**None** uses all variables in the model). The length of the argument must match the size of the second dimension of **A**.

sense: Constraint senses, provided as a NumPy 1-D ndarray or as a single character. Valid values are <, >, or =. The length of the array must be equal the size of the first dimension of `A`. A character will be promoted to an ndarray of the appropriate length.

b: Right-hand side vector, stored as a NumPy 1-D ndarray. The length of the array must be equal the size of the first dimension of `A`.

name: Single string defining the name for new constraints. The given name will be subscripted by the index of the constraint in the matrix.

Return value:

MConstr object.

Example usage:

```
A = np.full((5, 10), 1)
x = model.addMVar(10)
b = np.full(5, 1)

model.addMConstr(A, x, '=', b)
```

Model.addMQConstr()

```
addMQConstr  ( Q, c, sense, rhs, xQ_L=None, xQ_R=None, xc=None, name="" )
```

Add a quadratic constraint to the model using matrix semantics. The added constraint is $x'_{Q_L} Q x_{Q_R} + c' x_c$ = rhs (except that the constraint sense is determined by the **sense** argument). The `Q` argument must be a NumPy ndarray or a SciPy sparse matrix.

Note that you will typically use overloaded operators to build and add constraints using matrix semantics. The overloaded `@` operator can be used to build a linear matrix expression or quadratic matrix expression. An overloaded comparison operator can then be used to build a TempConstr object, which is then passed to addConstr.

Arguments:

Q: The quadratic constraint matrix - a NumPy 2-D ndarray or a SciPy sparse matrix.

c: The linear constraint vector - a NumPy 1-D ndarray. This can be `None` if there are no linear terms.

sense: Constraint sense. Valid values are <, >, or =.

rhs: Right-hand-side value.

xQ_L: Decision variables for quadratic terms; left multiplier for Q. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the size of the first dimension of `Q`.

xQ_R: Decision variables for quadratic terms; right multiplier for Q. The length of the argument must match the size of the second dimension of `Q`.

xc: Decision variables for linear terms. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the length of `c`.

name: Name for new constraint.

Return value:

The QConstr object.

Example usage:

```
Q = np.full((2, 3), 1)
```

```
xL = model.addMVar(2)
xR = model.addMVar(3)
model.addMQConstr(Q, None, '<', 1.0, xL, xR)
```

Model.addMVar()

```
addMVar ( shape, lb=0.0, ub=float('inf'), obj=0.0, vtype=GRB.CONTINUOUS,
          name="" )
```

Add an MVar object to a model. An `MVar` acts like a NumPy ndarray of Gurobi decision variables. An `MVar` can have an arbitrary number of dimensions, defined by the `shape` argument.

You can use arithmetic operations with `MVar` objects to create linear matrix expressions or quadratic matrix expressions, which can then be used to build linear or quadratic objectives or constraints.

The returned MVar object supports standard NumPy indexing and slicing. An MVar of size 1 can be passed in all places where gurobipy accepts a Var object.

Arguments:

`shape`: An int, or tuple of int. The `shape` of the array.

`lb (optional)`: Lower bound(s) for new variables.

`ub (optional)`: Upper bound(s) for new variables.

`obj (optional)`: Objective coefficient(s) for new variables.

`vtype (optional)`: Variable type(s) for new variables.

`name (optional)`: Names for new variables. The given name will be subscripted by the index of the generator expression, so if the index is an integer, `c` would become `c[0]`, `c[1]`, etc. Note that the generated names will be stored as ASCII strings, so you should avoid using names that contain non-ASCII characters. In addition, names that contain spaces are strongly discouraged, because they can't be written to LP format files.

The values of the `lb`, `ub`, `obj`, and `vtype` arguments can either be scalars, lists, or ndarrays. Their shapes should match the shape of the new `MVar` object, or they should be broadcastable to the given shape.

The `name` argument can either be a single string, used as a common base name that will be suffixed for each variable by its indices, or an ndarray of strings matching the shape of the new `MVar` object.

Return value:

New `MVar` object.

Example usage:

```
# Add a 4-by-2 matrix binary variable
x = model.addMVar((4,2), vtype=GRB.BINARY)
# Add a vector of three variables with non-default lower bounds
y = model.addMVar((3,), lb=[-1, -2, -1])
```

Model.addQConstr()

```
addQConstr ( lhs, sense=None, rhs=None, name="" )
```

Add a quadratic constraint to a model.

Important note: Gurobi can handle both convex and non-convex quadratic constraints. The differences between them can be both important and subtle. Refer to this discussion for additional information.

Arguments:

lhs: Left-hand side for new quadratic constraint. Can be a constant, a Var, a LinExpr, or a QuadExpr.

sense: Sense for new quadratic constraint (GRB.LESS_EQUAL, GRB.EQUAL, or GRB.GREATER_EQUAL).

rhs: Right-hand side for new quadratic constraint. Can be a constant, a Var, a LinExpr, or a QuadExpr.

name: Name for new constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New quadratic constraint object.

Example usage:

```
model.addQConstr(x*x + y*y, GRB.LESS_EQUAL, z*z, "c0")
model.addQConstr(x*x + y*y <= 2.0, "c1")
```

Model.addRange()

```
addRange  ( expr, lower, upper, name="" )
```

Add a range constraint to a model. A range constraint states that the value of the input expression must be between the specified **lower** and **upper** bounds in any solution.

Note that range constraints are stored internally as equality constraints. We add an extra variable to the model to capture the range information. Thus, the Sense attribute on a range constraint will always be GRB.EQUAL. In particular introducing a range constraint

$$L \leq a^T x \leq U$$

is equivalent to adding a slack variable s and the following constraints

$$\begin{aligned} a^T x - s &= L \\ 0 \leq s &\leq U - L. \end{aligned}$$

Arguments:

expr: Linear expression for new range constraint. Can be a Var or a LinExpr.

lower: Lower bound for linear expression.

upper: Upper bound for linear expression.

name: Name for new constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New constraint object.

Example usage:

```
# 1 <= x + y <= 2
model.addRange(x + y, 1.0, 2.0, "range0")

# overloaded forms
model.addConstr(x + y == [1.0, 2.0], name="range0")
```

Model.addSOS()

`addSOS ( type, vars, wts=None )`

Add an SOS constraint to the model. Please refer to this section for details on SOS constraints.

Arguments:

type: SOS type (can be GRB.SOS_TYPE1 or GRB.SOS_TYPE2).

vars: List of variables that participate in the SOS constraint.

weights (optional): Weights for the variables in the SOS constraint. Default weights are 1, 2, ...

Return value:

New SOS object.

Example usage:

```
model.addSOS(GRB.SOS_TYPE1, [x, y, z], [1, 2, 4])
```

Model.addVar()

`addVar ( lb=0.0, ub=float('inf'), obj=0.0, vtype=GRB.CONTINUOUS, name="", column=None )`

Add a decision variable to a model.

Arguments:

lb (optional): Lower bound for new variable.

ub (optional): Upper bound for new variable.

obj (optional): Objective coefficient for new variable.

vtype (optional): Variable type for new variable (GRB.CONTINUOUS, GRB.BINARY, GRB.INTEGER, GRB.SEMICONT, or GRB.SEMIINT).

name (optional): Name for new variable. Note that name will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

column (optional): Column object that indicates the set of constraints in which the new variable participates, and the associated coefficients.

Return value:

New variable object.

Example usage:

```
x = model.addVar()                                    # all default arguments
y = model.addVar(vtype=GRB.INTEGER, obj=1.0, name="y") # arguments by name
z = model.addVar(0.0, 1.0, 1.0, GRB.BINARY, "z")       # arguments by position
```

Model.addVars()

```
addVars  ( *indices, lb=0.0, ub=float('inf'), obj=0.0, vtype=GRB.CONTINUOUS,
           name="" )
```

Add multiple decision variables to a model.

Returns a Gurobi tupledict object that contains the newly created variables. The keys for the `tupledict` are derived from the `indices` argument(s). The arguments for this method can take several different forms, which will be described now.

The first arguments provide the indices that will be used as keys to access the variables in the returned `tupledict`. In its simplest version, you would specify one or more integer values, and this method would create the equivalent of a multi-dimensional array of variables. For example, `x = model.addVars(2, 3)` would create six variables, accessed as `x[0,0]`, `x[0,1]`, `x[0,2]`, `x[1,0]`, `x[1,1]`, and `x[1,2]`.

In a more complex version, you can specify arbitrary lists of immutable objects, and this method will create variables for each member of the cross product of these lists. For example, `x = model.addVars([3, 7], ['a', 'b', 'c'])` would create six variables, accessed as `x[3,'a']`, `x[7,'c']`, etc.

You can also provide your own list of tuples as indices. For example, `x = model.addVars([(3,'a'), (3,'b'), (7,'b'), (7,'c')])` would be accessed in the same way as the previous example (`x[3,'a']`, `x[7,'c']`, etc.), except that not all combinations will be present. This is typically how sparse indexing is handled.

Note that while the indices can be provided as multiple lists of objects, or as a list of tuples, the member values for a specific index must always be scalars (`int`, `float`, `string`, ...). For example, `x = model.addVars([(1, 3), 7], ['a'])` is not allowed, since the first argument for the first member would be `(1, 3)`. Similarly, `x = model.addVars([((1, 3),'a'), (7,'a')])` is also not allowed.

The named arguments (`lb`, `obj`, etc.) can take several forms. If you provide a scalar value (or use the default), then every variable will use that value. Thus, for example, `lb=1.0` will give every created variable a lower bound of 1.0. Note that a scalar value for the name argument has a special meaning, which will be discussed separately.

You can also provide a Python `dict` as the argument. In that case, the value for each variable will be pulled from the dict, using the indices argument to build the keys. For example, if the variables created by this method are indexed as `x[i,j]`, then the `dict` provided for the argument should have an entry for each possible `(i,j)` value.

Finally, if your `indices` argument is a single list, you can provide a Python `list` of the same length for the named arguments. For each variable, it will pull the value from the corresponding position in the list.

As noted earlier, the `name` argument is special. If you provide a scalar argument for the name, that argument will be transformed to have a subscript that corresponds to the index of the associated variable. For example, if you do `x = model.addVars(2,3,name="x")`, the variables will get names `x[0,0]`, `x[0,1]`, etc.

Arguments:

`indices`: Indices for accessing the new variables.

`lb (optional)`: Lower bound(s) for new variables.

`ub (optional)`: Upper bound(s) for new variables.

obj (optional): Objective coefficient(s) for new variables.

vtype (optional): Variable type(s) for new variables.

name (optional): Names for new variables. The given name will be subscripted by the index of the generator expression, so if the index is an integer, `c` would become `c[0]`, `c[1]`, etc. Note that the generated names will be stored as ASCII strings, so you should avoid using names that contain non-ASCII characters. In addition, names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New tupledict object that contains the new variables as values, using the provided indices as keys.

Example usage:

```
# 3-D array of binary variables
x = model.addVars(3, 4, 5, vtype=GRB.BINARY)

# variables index by tuplelist
l = tuplelist([(1, 2), (1, 3), (2, 3)])
y = model.addVars(l, ub=[1, 2, 3])

# variables with predetermined names
z = model.addVars(3, name=["a","b","c"])
```

Model.cbCut()

```
cbCut  ( lhs, sense, rhs )
```

Add a new cutting plane to a MIP model from within a callback function. Note that this method can only be invoked when the `where` value on the callback function is equal to `GRB.Callback.MIPNODE` (see the Callback Codes section for more information).

Cutting planes can be added at any node of the branch-and-cut tree. However, they should be added sparingly, since they increase the size of the relaxation model that is solved at each node and can significantly degrade node processing speed.

Cutting planes are typically used to cut off the current relaxation solution. To retrieve the relaxation solution at the current node, you should first call cbGetNodeRel.

You should consider setting parameter PreCrush to value 1 when adding your own cuts. This setting shuts off a few presolve reductions that can sometimes prevent your cut from being applied to the presolved model (which would result in your cut being silently ignored).

One very important note: you should only add cuts that are implied by the constraints in your model. If you cut off an integer solution that is feasible according to the original model constraints, *you are likely to obtain an incorrect solution to your MIP problem*.

Note that this method also accepts a TempConstr as its first argument. This allows you to use operator overloading to create constraints. See TempConstr for more information.

Arguments:

lhs: Left-hand side for new cut. Can be a constant, a Var, or a LinExpr.

sense: Sense for new cut (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

rhs: Right-hand side for new cut. Can be a constant, a Var, or a LinExpr.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPNODE:
    status = model.cbGet(GRB.Callback.MIPNODE_STATUS)
    if status == GRB.OPTIMAL:
      rel = model.cbGetNodeRel([model._vars[0], model._vars[1]])
      if rel[0] + rel[1] > 1.1:
        model.cbCut(model._vars[0] + model._vars[1] <= 1)

model._vars = model.getVars()
model.optimize(mycallback)
```

Model.cbGet()

`cbGet ( what )`

Query the optimizer from within the user callback.

Arguments:

what: Integer code that indicates what type of information is being requested by the callback. The set of valid codes depends on the **where** value that is passed into the user callback function. Please refer to the Callback Codes section for a list of possible **where** and **what** values.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.SIMPLEX:
    print(model.cbGet(GRB.Callback.SPX_OBJVAL))

model.optimize(mycallback)
```

Model.cbGetNodeRel()

`cbGetNodeRel ( vars )`

Retrieve values from the node relaxation solution at the current node. Note that this method can only be invoked when the **where** value on the callback function is equal to `GRB.Callback.MIPNODE`, and `GRB.Callback.MIPNODE_STATUS` is equal to `GRB.OPTIMAL` (see the Callback Codes section for more information).

Arguments:

vars: The variables whose relaxation values are desired. Can be a variable, a matrix variable, a list of variables or matrix variables, or a dict of variables.

Return value:

Values for the specified variables in the node relaxation for the current node. The format will depend on the input argument (a scalar, an ndarray, a list of values or ndarrays, or a dict).

Example usage:

```
def mycallback(model, where):
```

```
    if where == GRB.Callback.MIPNODE:
      status = model.cbGet(GRB.Callback.MIPNODE_STATUS)
      if status == GRB.OPTIMAL:
        print(model.cbGetNodeRel(model._vars))

model._vars = model.getVars()
model.optimize(mycallback)
```

Model.cbGetSolution()

```
cbGetSolution  ( vars )
```

Retrieve values from the new MIP solution. Note that this method can only be invoked when the `where` value on the callback function is equal to `GRB.Callback.MIPSOL` or `GRB.Callback.MULTIOBJ` (see the Callback Codes section for more information).

Arguments:

`vars`: The variables whose solution values are desired. Can be a variable, a matrix variable, a list of variables or matrix variables, or a dict of variables.

Return value:

Values for the specified variables in the solution. The format will depend on the input argument (a scalar, an ndarray, a list of values or ndarrays, or a dict).

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPSOL:
    print(model.cbGetSolution(model._vars))

model._vars = model.getVars()
model.optimize(mycallback)
```

Model.cbLazy()

```
cbLazy  ( lhs, sense, rhs )
```

Add a new lazy constraint to a MIP model from within a callback function. Note that this method can only be invoked when the `where` value on the callback function is `GRB.Callback.MIPNODE` or `GRB.Callback.MIPSOL` (see the Callback Codes section for more information).

Lazy constraints are typically used when the full set of constraints for a MIP model is too large to represent explicitly. By only including the constraints that are actually violated by solutions found during the branch-and-cut search, it is sometimes possible to find a proven optimal solution while only adding a fraction of the full set of constraints.

You would typically add a lazy constraint by first querying the current node solution (by calling cbGetSolution from a `GRB.Callback.MIPSOL` callback, or cbGetNodeRel from a `GRB.Callback.MIPNODE` callback), and then calling `cbLazy()` to add a constraint that cuts off the solution. Gurobi guarantees that you will have the opportunity to cut off any solutions that would otherwise be considered feasible.

MIP solutions may be generated outside of a MIP node. Thus, generating lazy constraints is optional when the `where` value in the callback function equals `GRB.Callback.MIPNODE`. To avoid this, we recommend to always check when the `where` value equals `GRB.Callback.MIPSOL`.

Your callback should be prepared to cut off solutions that violate any of your lazy constraints, including those that have already been added. Node solutions will usually respect previously added lazy constraints, but not always.

Note that you must set the LazyConstraints parameter if you want to use lazy constraints.

Note that this method also accepts a TempConstr as its first argument. This allows you to use operator overloading to create constraints. See TempConstr for more information.

Arguments:

lhs: Left-hand side for new lazy constraint. Can be a constant, a Var, or a LinExpr.

sense: Sense for new lazy constraint (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

rhs: Right-hand side for new lazy constraint. Can be a constant, a Var, or a LinExpr.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPSOL:
    sol = model.cbGetSolution([model._vars[0], model._vars[1]])
    if sol[0] + sol[1] > 1.1:
      model.cbLazy(model._vars[0] + model._vars[1] <= 1)

model._vars = model.getVars()
model.optimize(mycallback)
```

Model.cbProceed()

```
cbProceed ( )
```

Generate a request to proceed to the next phase of the computation. This routine can be called from any callback. Note that the request is only accepted in a few phases of the algorithm, and it won't be acted upon immediately.

In the current Gurobi version, this callback allows you to proceed from the NoRel heuristic to the standard MIP search. You can determine the current algorithm phase using `MIP_PHASE`, `MIPNODE_PHASE`, or `MIPSOL_PHASE` queries from a callback.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPSOL:
    phase = model.cbGet(GRB.Callback.MIPSOL_PHASE)
    obj = model.cbGet(GRB.Callback.MIPSOL_OBJ)
    if phase == GRB.PHASE_MIP_NOREL and obj < target_obj:
      model.cbProceed()

model.optimize(mycallback)
```

Model.cbSetSolution()

`cbSetSolution ( vars, solution )`

Import solution values for a heuristic solution. Only available when the `where` value on the callback function is equal to `GRB.Callback.MIP`, `GRB.Callback.MIPNODE`, or `GRB.Callback.MIPSOL` (see the Callback Codes section for more information).

When you specify a heuristic solution from a callback, variables initially take undefined values. You should use this method to specify variable values. You can make multiple calls to `cbSetSolution` from one callback invocation to specify values for multiple sets of variables. After the callback, if values have been specified for any variables, the Gurobi Optimizer will try to compute a feasible solution from the specified values, possibly filling in values for variables whose values were left undefined. You can also optionally call cbUseSolution within your callback function to try to immediately compute a feasible solution from the specified values.

Note that this method is not supported in a Compute Server environment.

Arguments:

`vars`: The variables whose values are being set. This can be a list of variables or a single variable.

`solution`: The desired values of the specified variables in the new solution.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPNODE:
    model.cbSetSolution(vars, newsolution)

model.optimize(mycallback)
```

Model.cbStopOneMultiObj()

`cbStopOneMultiObj ( objcnt )`

Interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process. Only available for multi-objective MIP models and when the `where` member variable is not equal to `GRB.Callback.MULTIOBJ` (see the Callback Codes section for more information).

You would typically stop a multi-objective optimization step by querying the last finished number of multi-objectives steps, and using that number to stop the current step and move on to the next hierarchical objective (if any) as shown in the following example:

Example usage:

```
import time

def mycallback(model, where):
  if where == GRB.Callback.MULTIOBJ:
    # get current objective number
    model._objcnt = model.cbGet(GRB.Callback.MULTIOBJ_OBJCNT)

    # reset start time to current time
```

```
      model._starttime = time.time()

    # See if we want to stop current multiobjective step
    else if time.time() - model._starttime > 1000:  # or solution is good enough
      # stop only this optimization step
      model.cbStopOneMultiObj(model._objcnt)

model._objcnt = 0
model._starttime = time.time()
model.optimize(mycallback)
```

You should refer to the section on Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Arguments:

`objnum`: The number of the multi-objective optimization step to interrupt. For processes running locally, this argument can have the special value -1, meaning to stop the current step.

Model.cbUseSolution()

```
cbUseSolution  (  )
```

Once you have imported solution values using cbSetSolution, you can optionally call `cbUseSolution` in a `GRB.Callback.MIPNODE` callback to immediately use these values to try to compute a heuristic solution. Alternatively, you can call `cbUseSolution` in a `GRB.Callback.MIP` or `GRB.Callback.MIPSOL` callback, which will store the solution until it can be processed internally.

Return value:

The objective value for the solution obtained from your solution values. It equals `GRB.INFINITY` if no improved solution is found or the method has been called from a callback other than `GRB.Callback.MIPNODE` as, in these contexts, the solution is stored instead of being processed immediately.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPNODE:
    model.cbSetSolution(vars, newsolution)
    objval = model.cbUseSolution()

model.optimize(mycallback)
```

Model.chgCoeff()

```
chgCoeff  ( constr, var, newvalue )
```

Change one coefficient in the model. The desired change is captured using a `Var` object, a `Constr` object, and a desired coefficient for the specified variable in the specified constraint. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

Arguments:

`constr`: Constraint for coefficient to be changed.

`var`: Variable for coefficient to be changed.

`newvalue`: Desired new value for coefficient.

Example usage:

```
model.chgCoeff(c0, x, 2.0)
```

Model.close()

```
close ( )
```

Free all resources associated with this Model object. This method is a synonym for dispose. After this method is called, this Model object must no longer be used.

Example usage:

```
env = Env()
model = read("misc07.mps", env)
model.optimize()
model.close()
env.close()
```

Model.computeIIS()

```
computeIIS ( callback=None )
```

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the smallest one; there may exist others with fewer constraints or bounds.

IIS results are returned in a number of attributes: IISConstr, IISLB, IISUB, IISSOS, IISQConstr, and IISGenConstr. Each indicates whether the corresponding model element is a member of the computed IIS.

Note that for models with general function constraints, piecewise-linear approximation of the constraints may cause unreliable IIS results.

The IIS log provides information about the progress of the algorithm, including a guess at the eventual IIS size.

If an IIS computation is interrupted before completion, Gurobi will return the smallest infeasible subsystem found to that point.

The IISConstrForce, IISLBForce, IISUBForce, IISSOSForce, IISQConstrForce, and IISGenConstrForce attributes allow you mark model elements to either include or exclude from the computed

IIS. Setting the attribute to 1 forces the corresponding element into the IIS, setting it to 0 forces it out of the IIS, and setting it to -1 allows the algorithm to decide.

To give an example of when these attributes might be useful, consider the case where an initial model is known to be feasible, but it becomes infeasible after adding constraints or tightening bounds. If you are only interested in knowing which of the changes caused the infeasibility, you can force the unmodified bounds and constraints into the IIS. That allows the IIS algorithm to focus exclusively on the new constraints, which will often be substantially faster.

Note that setting any of the `Force` attributes to 0 may make the resulting subsystem feasible, which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

This method populates the IISConstr, IISQConstr, and IISGenConstr constraint attributes, the IISSOS, SOS attribute, and the IISLB and IISUB variable attributes. You can also obtain information about the results of the IIS computation by writing an `.ilp` format file (see Model.write). This file contains only the IIS from the original model.

Use the IISMethod parameter to adjust the behavior of the IIS algorithm.

Note that this method can be used to compute IISs for both continuous and MIP models.

Arguments:

`callback`: Callback function. The callback function should take two arguments, `model` and `where`. While the IIS algorithm is running, the function will be called periodically, with `model` set to the model being optimized, and `where` indicating where in the optimization the callback is called from. See the Callback class for additional information.

Example usage:

```
model.computeIIS()
model.write("model.ilp")
```

Model.copy()

```
copy  ( targetenv=None )
```

Copy a model.

With no arguments, the copy will live in the same environment as the original model. Provide an argument to copy an existing model to a different environment, typically to enable different threads to operate on different versions of the same model, since multiple threads can not simultaneously work within the same environment.

Note that this method itself is not thread safe, so you should either call it from the main thread or protect access to it with a lock.

Note that pending updates will not be applied to the model, so you should call update before copying if you would like those to be included in the copy.

For Compute Server users, note that you can copy a model from a client to a Compute Server environment, but it is not possible to copy models from a Compute Server environment to another (client or Compute Server) environment.

Arguments:

`targetenv (optional)`: Environment to copy the model into.

Return value:

Copy of model.

Example usage:

```
model.update() # If you have unstaged changes in the model
copy = model.copy()
```

Model.discardConcurrentEnvs()

```
discardConcurrentEnvs ( )
```

Discard concurrent environments for a model.

The concurrent environments created by getConcurrentEnv will be used by every subsequent call to the concurrent optimizer until the concurrent environments are discarded.

Example usage:

```
env0 = model.getConcurrentEnv(0)
env1 = model.getConcurrentEnv(1)

env0.setParam('Method', 0)
env1.setParam('Method', 1)

model.optimize()

model.discardConcurrentEnvs()
```

Model.discardMultiobjEnvs()

```
discardMultiobjEnvs ( )
```

Discard all multi-objective environments associated with the model, thus restoring multi objective optimization to its default behavior.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Use getMultiobjEnv to create a multi-objective environment.

Example usage:

```
env0 = model.getMultiobjEnv(0)
env1 = model.getMultiobjEnv(1)

env0.setParam('Method', 2)
env1.setParam('Method', 1)

model.optimize()

model.discardMultiobjEnvs()
```

Model.dispose()

```
dispose ( )
```

Free all resources associated with this Model object. This method is a synonym for close. After this method is called, this Model object must no longer be used.

Example usage:

```
env = Env()
model = read("misc07.mps", env)
model.optimize()
model.dispose()
env.dispose()
```

Model.feasRelaxS()

```
feasRelaxS  ( relaxobjtype, minrelax, vrelax, crelax )
```

Modifies the `Model` object to create a feasibility relaxation. Note that you need to call optimize on the result to compute the actual relaxed solution. Note also that this is a simplified version of this method - use feasRelax for more control over the relaxation performed.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This method provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the magnitudes of the bound and constraint violations.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the sum of the squares of the bound and constraint violations.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the total number of bound and constraint violations.

To give an example, if a constraint is violated by 2.0, it would contribute `2.0` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2.0*2.0` for `relaxobjtype=1`, and it would contribute `1.0` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=False`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=True`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `feasRelaxS` must solve an optimization problem to find the minimum possible relaxation when `minrelax=True`, which can be quite expensive.

Note that this is a destructive method: it modifies the model on which it is invoked. If you don't want to modify your original model, use copy to create a copy before invoking this method.

Arguments:

`relaxobjtype`: The cost function used when finding the minimum cost relaxation.

`minrelax`: The type of feasibility relaxation to perform.

`vrelax`: Indicates whether variable bounds can be relaxed.

`crelax`: Indicates whether constraints can be relaxed.

Return value:

Zero if `minrelax` is False. If `minrelax` is True, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

Example usage:

```
if model.status == GRB.INFEASIBLE:
  model.feasRelaxS(1, False, False, True)
  model.optimize()
```

Model.feasRelax()

```
feasRelax  ( relaxobjtype, minrelax, vars, lbpen, ubpen, constrs, rhspen )
```

Modifies the `Model` object to create a feasibility relaxation. Note that you need to call optimize on the result to compute the actual relaxed solution. Note also that this is a more complex version of this method - use feasRelaxS for a simplified version.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This method provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost per unit violation in the lower bounds, upper bounds, and linear constraints, respectively.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the coefficients on the squares of the lower bound, upper bound, and linear constraint violations, respectively.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the weighted count of bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost of violating a lower bound, upper bound, and linear constraint, respectively.

To give an example, if a constraint with `rhspen` value `p` is violated by 2.0, it would contribute `2*p` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2*2*p` for `relaxobjtype=1`, and it would contribute `p` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=False`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=True`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `feasRelax` must solve an optimization problem to find the minimum possible relaxation when `minrelax=True`, which can be quite expensive.

Note that this is a destructive method: it modifies the model on which it is invoked. If you don't want to modify your original model, use copy to create a copy before invoking this method.

Arguments:

`relaxobjtype`: The cost function used when finding the minimum cost relaxation.

`minrelax`: The type of feasibility relaxation to perform.

`vars`: Variables whose bounds are allowed to be violated.

`lbpen`: Penalty for violating a variable lower bound. One entry for each variable in argument `vars`.

`ubpen`: Penalty for violating a variable upper bound. One entry for each variable in argument `vars`.

`constrs`: Linear constraints that are allowed to be violated.

rhspen: Penalty for violating a linear constraint. One entry for each constraint in argument `constrs`.

Return value:

Zero if `minrelax` is False. If `minrelax` is True, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

Example usage:

```
if model.status == GRB.INFEASIBLE:
  vars = model.getVars()
  ubpen = [1.0]*model.numVars
  model.feasRelax(1, False, vars, None, ubpen, None, None)
  model.optimize()
```

Model.fixed()

```
fixed ( )
```

Create the fixed model associated with a MIP model. The MIP model must have a solution loaded (e.g., after a call to the optimize method). In the fixed model, each integer variable is fixed to the value that variable takes in the MIP solution. In addition, continuous variables may be fixed to satisfy SOS or general constraints. The result is a model without any integrality constraints, SOS constraints, or general constraints.

Note that, while the fixed problem is always a continuous model, it may contain a non-convex quadratic objective or non-convex quadratic constraints. As a result, it may still be solved using the MIP algorithm.

Return value:

Fixed model associated with calling object.

Example usage:

```
fixed = model.fixed()
```

Model.getA()

```
getA ( )
```

Query the linear constraint matrix of the model. You'll need to have `scipy` installed for this function to work.

Arguments:

Return value:

The matrix as a `scipy.sparse` matrix in CSR format.

Example usage:

```
A     = model.getA()
sense = numpy.array(model.getAttr("Sense",model.getConstrs()))
# extract sub matrices of (in)equality constraints
Aeq = A[sense == '=', :]
Ale = A[sense == '<', :]
Age = A[sense == '>', :]
```

Model.getAttr()

```
getAttr ( attrname, objs=None )
```

Query the value of an attribute. When called with a single argument, it returns the value of a model attribute. When called with two arguments, it returns the value of an attribute for either a list or a dictionary containing either variables or constraints. If called with a list, the result is a list. If called with a dictionary, the result is a dictionary that uses the same keys, but is populated with the requested attribute values. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `Model` object.

Arguments:

`attrname`: Name of the attribute.

`objs (optional)`: List or dictionary containing either constraints or variables

Example usage:

```
print(model.numintvars)
print(model.getAttr("numIntVars"))
print(model.getAttr(GRB.Attr.NumIntVars))
print(model.getAttr("X", model.getVars()))
print(model.getAttr("Pi", model.getConstrs()))
```

Model.getCoeff()

```
getCoeff ( constr, var )
```

Query the coefficient of variable `var` in linear constraint `constr` (note that the result can be zero).

Arguments:

`constr`: The requested constraint.

`var`: The requested variable.

Return value:

The current value of the requested coefficient.

Example usage:

```
print(model.getCoeff(constr, var))
```

Model.getCol()

```
getCol ( var )
```

Retrieve the list of constraints in which a variable participates, and the associated coefficients. The result is returned as a Column object.

Arguments:

`var`: The variable of interest.

Return value:

A Column object that captures the set of constraints in which the variable participates.

Example usage:

```
print(model.getCol(model.getVars()[0]))
```

Model.getConcurrentEnv()

`getConcurrentEnv ( num )`

Create/retrieve a concurrent environment for a model.

This method provides fine-grained control over the concurrent optimizer. By creating your own concurrent environments and setting appropriate parameters on these environments (e.g., the Method parameter), you can control exactly which strategies the concurrent optimizer employs. For example, if you create two concurrent environments, and set `Method` to primal simplex for one and dual simplex for the other, subsequent concurrent optimizer runs will use the two simplex algorithms rather than the default choices.

Note that you must create contiguously numbered concurrent environments, starting with `num=0`. For example, if you want three concurrent environments, they must be numbered 0, 1, and 2.

Once you create concurrent environments, they will be used for every subsequent concurrent optimization on that model. Use discardConcurrentEnvs to revert back to default concurrent optimizer behavior.

Arguments:

`num (int)`: The concurrent environment number.

Return value:

The concurrent environment for the model.

Example usage:

```
env0 = model.getConcurrentEnv(0)
env1 = model.getConcurrentEnv(1)

env0.setParam('Method', 0)
env1.setParam('Method', 1)

model.optimize()

model.discardConcurrentEnvs()
```

Model.getConstrByName()

`getConstrByName ( name )`

Retrieve a linear constraint from its name. If multiple linear constraints have the same name, this method chooses one arbitrarily.

Arguments:

`name`: Name of desired constraint.

Return value:

Constraint with the specified name.

Example usage:

```
c0 = model.getConstrByName("c0")
```

Model.getConstrs()

```
getConstrs ( )
```

Retrieve a list of all linear constraints in the model.

Return value:

All linear constraints in the model.

Example usage:

```
constrs = model.getConstrs()
c0 = constrs[0]
```

Model.getGenConstrMax()

```
getGenConstrMax ( genconstr )
```

Retrieve the data associated with a general constraint of type MAX. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrMax for a description of the semantics of this general constraint type.

Arguments:

genconstr: The general constraint object of interest.

Return value:

A tuple (resvar, vars, constant) that contains the data associated with the general constraint:
resvar (Var): Resultant variable of the MAX constraint.
vars (list of Var): Operand variables of the MAX constraint.
constant (float): Additional constant operand of the MAX constraint.

Example usage:

```
# x5 = max(x1, x3, x4, 2.0)
maxconstr = model.addGenConstrMax(x5, [x1, x3, x4], 2.0, "maxconstr")
model.update()
(resvar, vars, constant) = model.getGenConstrMax(maxconstr)
```

Model.getGenConstrMin()

```
getGenConstrMin ( genconstr )
```

Retrieve the data associated with a general constraint of type MIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrMin for a description of the semantics of this general constraint type.

Arguments:

genconstr: The general constraint object of interest.

Return value:

A tuple (resvar, vars, constant) that contains the data associated with the general constraint:
resvar (Var): Resultant variable of the MIN constraint.
vars (list of Var): Operand variables of the MIN constraint.
constant (float): Additional constant operand of the MIN constraint.

Example usage:

```
# x5 = min(x1, x3, x4, 2.0)
minconstr = model.addGenConstrMin(x5, [x1, x3, x4], 2.0, "minconstr")
model.update()
(resvar, vars, constant) = model.getGenConstrMin(minconstr)
```

Model.getGenConstrAbs()

```
getGenConstrAbs ( genconstr )
```

Retrieve the data associated with a general constraint of type ABS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrAbs for a description of the semantics of this general constraint type.

Arguments:

genconstr: The general constraint object of interest.

Return value:

A tuple (resvar, argvar) that contains the data associated with the general constraint:
resvar (Var): Resultant variable of ABS constraint.
argvar (Var): Argument variable of ABS constraint.

Example usage:

```
# x5 = abs(x1)
absconstr = model.addGenConstrAbs(x5, x1, "absconstr")
model.update()
(resvar, argvar) = model.getGenConstrAbs(absconstr)
```

Model.getGenConstrAnd()

```
getGenConstrAnd ( genconstr )
```

Retrieve the data associated with a general constraint of type AND. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrAnd for a description of the semantics of this general constraint type.

Arguments:

genconstr: The general constraint object of interest.

Return value:

A tuple (resvar, vars) that contains the data associated with the general constraint:
resvar (Var): Resultant variable of AND constraint.
vars (list of Var): Operand variables of AND constraint.

Example usage:

```
# x5 = and(x1, x3, x4)
```

```
andconstr = model.addGenConstrAnd(x5, [x1, x3, x4], "andconstr")
model.update()
(resvar, vars) = model.getGenConstrAnd(andconstr)
```

Model.getGenConstrOr()

`getGenConstrOr ( genconstr )`

Retrieve the data associated with a general constraint of type OR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrOr for a description of the semantics of this general constraint type.

Arguments:

genconstr: The general constraint object of interest.

Return value:

A tuple (resvar, vars) that contains the data associated with the general constraint:

resvar (Var): Resultant variable of OR constraint.

vars (list of Var): Operand variables of OR constraint.

Example usage:

```
# x5 = or(x1, x3, x4)
orconstr = model.addGenConstrOr(x5, [x1, x3, x4], "orconstr")
model.update()
(resvar, vars) = model.getGenConstrOr(orconstr)
```

Model.getGenConstrNorm()

`getGenConstrNorm ( genconstr )`

Retrieve the data associated with a general constraint of type NORM. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrNorm for a description of the semantics of this general constraint type.

Arguments:

genconstr: The general constraint object of interest.

Return value:

A tuple (resvar, vars, which) that contains the data associated with the general constraint:

resvar (Var): Resultant variable of NORM constraint.

vars (list of Var): Operand variables of NORM constraint.

which (float): Which norm (possible values are 0, 1, 2, or GRB.INFINITY).

Example usage:

```
# x5 = norm2(x1, x3, x4)
normconstr = model.addGenConstrNorm(x5, [x1, x3, x4], 2.0, "normconstr")
model.update()
(resvar, vars, which) = model.getGenConstrNorm(normconstr)
```

Model.getGenConstrIndicator()

`getGenConstrIndicator ( genconstr )`

Retrieve the data associated with a general constraint of type INDICATOR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrIndicator for a description of the semantics of this general constraint type.

Arguments:

genconstr: The general constraint object of interest.

Return value:

A tuple (binvar, binval, expr, sense, rhs) that contains the data associated with the general constraint:

binvar (Var): Antecedent variable of indicator constraint.

binval (Boolean): Value of antecedent variable that activates the linear constraint.

expr (LinExpr): LinExpr object containing the left-hand side of the constraint triggered by the indicator.

sense (char): Sense of linear constraint triggered by the indicator (e.g., `GRB.LESS_EQUAL`).

rhs (float): Right-hand side of linear constraint triggered by the indicator.

Example usage:

```
# x7 = 1 -> x1 + 2 x3 + x4 = 1
indconstr = model.addGenConstrIndicator(x7, True, x1 + 2*x2 + x4, GRB.EQUAL, 1.0)
model.update()
(binvar, binval, expr, sense, rhs) = model.getGenConstr(indconstr)
```

Model.getGenConstrPWL()

`getGenConstrPWL ( genconstr )`

Retrieve the data associated with a general constraint of type PWL. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrPWL for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar, xpts, ypts) that contains the data associated with the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

xpts (list of float): The x values for the points that define the piecewise-linear function.

ypts (list of float): The y values for the points that define the piecewise-linear function.

Example usage:

```
pwlconstr = model.addGenConstrPWL(x, y, [0, 1, 2], [1.5, 0, 3], "myPWLConstr")
```

```
model.update()
(xvar, yvar, xpts, ypts) = model.getGenConstrPWL(pwlconstr)
```

Model.getGenConstrPoly()

`getGenConstrPoly ( genconstr )`

Retrieve the data associated with a general constraint of type POLY. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrPoly for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar, p) that contains the data associated with the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

p (list of float): The coefficients for polynomial function.

Example usage:

```
# y = 2 x^3 + 1.5 x^2 + 1
polyconstr = model.addGenConstrPoly(x, y, [2, 1.5, 0, 1])
model.update()
(xvar, yvar, p) = model.getGenConstrPoly(polyconstr)
```

Model.getGenConstrExp()

`getGenConstrExp ( genconstr )`

Retrieve the data associated with a general constraint of type EXP. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrExp for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data associated with the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

Example usage:

```
# y = exp(x)
expconstr = model.addGenConstrExp(x, y)
model.update()
(xvar, yvar) = model.getGenConstrExp(expconstr)
```

Model.getGenConstrExpA()

`getGenConstrExpA ( genconstr )`

Retrieve the data associated with a general constraint of type EXPA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrExpA for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar, a) that contains the data associated with the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

a (float): The base of the function.

Example usage:

```
# y = 3^x
expaconstr = model.addGenConstrExpA(x, y, 3.0, "expa")
model.update()
(xvar, yvar, a) = model.getGenConstrExpA(expaconstr)
```

Model.getGenConstrLog()

`getGenConstrLog ( genconstr )`

Retrieve the data associated with a general constraint of type LOG. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLog for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data associated with the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

Example usage:

```
# y = ln(x)
lnconstr = model.addGenConstrLog(x, y)
model.update()
(xvar, yvar) = model.getGenConstrLog(lnconstr)
```

Model.getGenConstrLogA()

`getGenConstrLogA ( genconstr )`

Retrieve the data associated with a general constraint of type LOGA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLogA for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar, a) that contains the data associated with the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

a (float): The base of the function.

Example usage:

```
# y = log10(x)
log10constr = model.addGenConstrLogA(x, y, 10.0, "log10")
model.update()
(xvar, yvar, a) = model.getGenConstrLogA(log10constr)
```

Model.getGenConstrLogistic()

```
getGenConstrLogistic  ( genconstr )
```

Retrieve the data associated with a general constraint of type LOGISTIC. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLogistic for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data associated with the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

Example usage:

```
# y = 1 / (1 + exp(-x))
expconstr = model.addGenConstrLogistic(x, y)
model.update()
(xvar, yvar) = model.getGenConstrLogistic(expconstr)
```

Model.getGenConstrPow()

```
getGenConstrPow  ( genconstr )
```

Retrieve the data associated with a general constraint of type POW. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrPow for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar, a) that contains the data associated with the general constraint:
xvar (Var): The x variable.
yvar (Var): The y variable.
a (float): The exponent of the function.

Example usage:

```
# y = x^3.5
powconstr = model.addGenConstrPow(x, y, 3.5, "gf")
model.update()
(xvar, yvar, a) = model.getGenConstrPow(powconstr)
```

Model.getGenConstrSin()

`getGenConstrSin ( genconstr )`

Retrieve the data associated with a general constraint of type SIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrSin for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data associated with the general constraint:
xvar (Var): The x variable.
yvar (Var): The y variable.

Example usage:

```
# y = sin(x)
sinconstr = model.addGenConstrSin(x, y)
model.update()
(xvar, yvar) = model.getGenConstrSin(sinconstr)
```

Model.getGenConstrCos()

`getGenConstrCos ( genconstr )`

Retrieve the data associated with a general constraint of type COS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrCos for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data associated with the general constraint:
xvar (Var): The x variable.
yvar (Var): The y variable.

Example usage:

```
# y = cos(x)
cosconstr = model.addGenConstrCos(x, y)
model.update()
(xvar, yvar) = model.getGenConstrCos(cosconstr)
```

Model.getGenConstrTan()

`getGenConstrTan ( genconstr )`

Retrieve the data associated with a general constraint of type TAN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrTan for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data associated with the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

Example usage:

```
# y = tan(x)
tanconstr = model.addGenConstrTan(x, y)
model.update()
(xvar, yvar) = model.getGenConstrTan(tanconstr)
```

Model.getGenConstrs()

`getGenConstrs ( )`

Retrieve a list of all general constraints in the model.

Return value:

All general constraints in the model.

Example usage:

```
gencons = model.getGenConstrs()
for gc in gencons:
  print(model.getGenConstr(gc))
```

Model.getJSONSolution()

`getJSONSolution ( )`

After a call to optimize, this method returns the resulting solution and related model attributes as a JSON string. Please refer to the JSON solution format section for details.

Return value:

A JSON string.

Example usage:

```
model = read('p0033.mps')
model.optimize()
print(model.getJSONSolution())
```

Model.getMultiobjEnv()

```
getMultiobjEnv  ( index )
```

Create/retrieve a multi-objective environment for the optimization pass with the given index. This environment enables fine-grained control over the multi-objective optimization process. Specifically, by changing parameters on this environment, you modify the behavior of the optimization that occurs during the corresponding pass of the multi-objective optimization.

Each multi-objective environment starts with a copy of the current model environment.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Please refer to the discussion on Combining Blended and Hierarchical Objectives for information on the optimization passes to solve multi-objective models.

Use discardMultiobjEnvs to discard multi-objective environments and return to standard behavior.

Arguments:

index (int): The optimization pass index, starting from 0.

Return value:

The multi-objective environment for that optimization pass when solving the model.

Example usage:

```
env0 = model.getMultiobjEnv(0)
env1 = model.getMultiobjEnv(1)

env0.setParam('TimeLimit', 100)
env1.setParam('TimeLimit', 10)

model.optimize()

model.discardMultiobjEnvs()
```

Model.getObjective()

```
getObjective  ( index=None )
```

Retrieve the model objective(s).

Call this with no argument to retrieve the primary objective, or with an integer argument to retrieve the corresponding alternative objective.

Arguments:

index (int, optional): The index for the requested alternative objective.

Return value:

The model objective. A LinExpr object for a linear objective, or a QuadExpr object for a quadratic objective. Note that alternative objectives are always linear.

Example usage:

```
obj = model.getObjective()
print(obj.getValue())
```

Model.getParamInfo()

```
getParamInfo  ( paramname )
```

Retrieve information about a Gurobi parameter, including the type, the current value, the minimum and maximum allowed values, and the default value.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Arguments:

paramname: String containing the name of the parameter of interest. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed and the method returns `None`.

Return value:

Returns a 6-entry tuple that contains: the parameter name, the parameter type, the current value, the minimum value, the maximum value, and the default value.

Example usage:

```
print(model.getParamInfo('Heuristics'))
```

Model.getPWLObj()

```
getPWLObj  ( var )
```

Retrieve the piecewise-linear objective function for a variable. The function returns a list of tuples, where each provides the x and y coordinates for the points that define the piecewise-linear objective function.

Refer to this discussion for additional information on what the values in x and y mean.

Arguments:

var: A Var object that gives the variable whose objective function is being retrieved.

Return value:

The points that define the piecewise-linear objective function.

Example usage:

```
> print(model.getPWLObj(var))
[(1, 1), (2, 2), (3, 4)]
```

Model.getQConstrs()

```
getQConstrs  (  )
```

Retrieve a list of all quadratic constraints in the model.

Return value:

All quadratic constraints in the model.

Example usage:

```
qconstrs = model.getQConstrs()
qc0 = qconstrs[0]
```

Model.getQCRow()

`getQCRow ( qconstr )`

Retrieve the left-hand side expression from a quadratic constraint. The result is returned as a QuadExpr object.

Arguments:

qconstr: The constraint of interest. A QConstr object, typically obtained from addQConstr or getQConstrs.

Return value:

A QuadExpr object that captures the left-hand side of the quadratic constraint.

Example usage:

```
print(model.getQCRow(model.getQConstrs()[0]))
```

Model.getRow()

`getRow ( constr )`

Retrieve the list of variables that participate in a constraint, and the associated coefficients. The result is returned as a LinExpr object.

Arguments:

constr: The constraint of interest. A Constr object, typically obtained from addConstr or getConstrs.

Return value:

A LinExpr object that captures the set of variables that participate in the constraint.

Example usage:

```
constrs = model.getConstrs()
print(model.getRow(constrs[0]))
```

Model.getSOS()

`getSOS ( sos )`

Retrieve information about an SOS constraint. The result is a tuple that contains the SOS type (1 or 2), the list of participating Var objects, and the list of associated SOS weights.

Arguments:

sos: The SOS constraint of interest. An SOS object, typically obtained from addSOS or getSOSs.

Return value:

A tuple that contains the SOS type (1 or 2), a list of participating Var objects, and a list of associated SOS weights.

Example usage:

```
(sostype, vars, weights) = model.getSOS(model.getSOSs()[0])
```

Model.getSOSs()

```
getSOSs ( )
```

Retrieve a list of all SOS constraints in the model.

Return value:

All SOS constraints in the model.

Example usage:

```
sos = model.getSOSs()
for s in sos:
  print(model.getSOS(s))
```

Model.getTuneResult()

```
getTuneResult ( )
```

Use this routine to retrieve the results of a previous tune call. Calling this method with argument `n` causes tuned parameter set `n` to be copied into the model. Parameter sets are stored in order of decreasing quality, with parameter set 0 being the best. The number of available sets is stored in attribute TuneResultCount.

Once you have retrieved a tuning result, you can call optimize to use these parameter settings to optimize the model, or write to write the changed parameters to a `.prm` file.

Please refer to the parameter tuning section for details on the tuning tool.

Arguments:

`n`: The index of the tuning result to retrieve. The best result is available as index 0. The number of stored results is available in attribute TuneResultCount.

Example usage:

```
model.tune()
for i in range(model.tuneResultCount):
  model.getTuneResult(i)
  model.write('tune'+str(i)+'.prm')
```

Model.getVarByName()

```
getVarByName ( name )
```

Retrieve a variable from its name. If multiple variables have the same name, this method chooses one arbitrarily.

Arguments:

`name`: Name of desired variable.

Return value:

Variable with the specified name.

Example usage:

```
x0 = model.getVarByName("x0")
```

Model.getVars()

```
getVars ( )
```

Retrieve a list of all variables in the model.

Return value:

All variables in the model.

Example usage:

```
vars = model.getVars()
x0 = vars[0]
```

Model.message()

```
message ( msg )
```

Append a string to the Gurobi log file.

Arguments:

msg: String to append to Gurobi log file.

Example usage:

```
model.message('New message')
```

Model.optimize()

```
optimize ( callback=None )
```

Optimize the model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion, this method will populate the solution related attributes of the model. See the Attributes section for more information on attributes.

Please consult this section for a discussion of some of the practical issues associated with solving a precisely defined mathematical model using finite-precision floating-point arithmetic.

Note that this method will process all pending model modifications.

Arguments:

callback: Callback function. The callback function should take two arguments, `model` and `where`. During the optimization, the function will be called periodically, with `model` set to the model being optimized, and `where` indicating where in the optimization the callback is called from. See the Callback class for additional information.

Example usage:

```
model.optimize()
```

Model.optimizeBatch()

```
optimizeBatch ( )
```

Submit a new batch request to the Cluster Manager. Returns the `BatchID` (a string), which uniquely identifies the job in the Cluster Manager and can be used to query the status of this request (from this program or from any other). Once the request has completed, the `BatchID` can

also be used to retrieve the associated solution. To submit a batch request, you must tag at least one element of the model by setting one of the VTag, CTag or QCTag attributes. For more details on batch optimization, please refer to the Batch Optimization section.

Note that this routine will process all pending model modifications.

Example usage:

```
# Submit batch request
batchID = model.optimizeBatch()
```

Return value:

A unique string identifier for the batch request.

Model.presolve()

```
presolve ( )
```

Perform presolve on a model.

Please note that the presolved model computed by this function may be different from the presolved model computed when optimizing the model.

Return value:

Presolved version of original model.

Example usage:

```
p = model.presolve()
p.printStats()
```

Model.printAttr()

```
printAttr ( attrs, filter='*' )
```

Print the value of one or more attributes. If `attrs` is a constraint or variable attribute, print all non-zero values of the attribute, along with the associated constraint or variable names. If `attrs` is a list of attributes, print attribute values for all listed attributes. The method takes an optional `filter` argument, which allows you to select which specific attribute values to print (by filtering on the constraint or variable name).

See the Attributes section for a list of all available attributes.

Arguments:

`attrs`: Name of attribute or attributes to print. The value can be a single attribute or a list of attributes. If a list is given, all listed attributes must be of the same type (model, variable, or constraint).

`filter (optional)`: Filter for values to print -- name of constr/var must match filter to be printed.

Example usage:

```
model.printAttr('x')          # all non-zero solution values
model.printAttr('lb', 'x*')   # bounds for vars whose names begin with 'x'
model.printAttr(['lb', 'ub']) # lower and upper bounds
```

Model.printQuality()

```
printQuality ( )
```

Print statistics about the quality of the computed solution (constraint violations, integrality violations, etc.).

For continuous models, the output will include the maximum unscaled and scaled violation, as well as the variable or constraint name associated with the worst unscaled violation. For MIP models, the output will include the maximum unscaled violation and the associated variable or constraint name.

Example usage:

```
model.optimize()
model.printQuality()
```

Model.printStats()

```
printStats ( )
```

Print statistics about the model (number of constraints and variables, number of non-zeros in constraint matrix, smallest and largest coefficients, etc.).

Example usage:

```
model.printStats()
```

Model.read()

```
read ( filename )
```

This method is the general entry point for importing data from a file into a model. It can be used to read basis files for continuous models, start vectors for MIP models, or parameter settings. The type of data read is determined by the file suffix. File formats are described in the File Format section.

Note that reading a file does **not** process all pending model modifications. These modifications can be processed by calling Model.update.

Note also that this is **not** the method to use if you want to read a new model from a file. For that, use the read command.

Arguments:

`filename`: Name of the file to read. The suffix on the file must be either `.bas` (for an LP basis), `.mst` or `.sol` (for a MIP start), `.hnt` (for MIP hints), `.ord` (for a priority order), or `.prm` (for a parameter file). The suffix may optionally be followed by `.zip`, `.gz`, `.bz2`, or `.7z`. The file name may contain `*` or `?` wildcards. No file is read when no wildcard match is found. If more than one match is found, this method will attempt to read the first matching file.

Example usage:

```
model.read('input.bas')
model.read('input.mst')
```

Model.relax()

```
relax ( )
```

Create the relaxation of a MIP model. Transforms integer variables into continuous variables, and removes SOS and general constraints.

Return value:

Relaxed version of model.

Example usage:

```
r = model.relax()
```

Model.remove()

```
remove ( items )
```

Remove variables, linear constraints, quadratic constraints, SOS constraints, or general constraints from a model.

Arguments:

`items`: The items to remove from the model. Argument can be a single Var, MVar, Constr, MConstr, QConstr, MQConstr, SOS, or GenConstr, or a `list`, `tuple`, or `dict` containing these objects. If the argument is a `dict`, the values will be removed, not the keys.

Example usage:

```
model.remove(model.getVars()[0])
model.remove(model.getVars()[0:10])
model.remove(model.getConstrs()[0])
model.remove(model.getConstrs()[1:3])
model.remove(model.getQConstrs()[0])
model.remove(model.getSOSs()[0])
model.remove(model.getGenConstrs()[0])
```

Model.reset()

```
reset ( clearall=0 )
```

Reset the model to an unsolved state, discarding any previously computed solution information.

Arguments:

`clearall (int, optional)`: A value of 1 discards additional information that affects the solution process but not the actual model (currently MIP starts, variable hints, branching priorities, lazy flags, and partition information). The default value just discards the solution.

Example usage:

```
model.reset(0)
```

Model.resetParams()

```
resetParams ( )
```

Reset all parameters to their default values.

Example usage:

```
model.resetParams()
```

Model.setAttr()

```
setAttr ( attrname, objects, newvalues )
```

Change the value of an attribute.

Call this method with two arguments (i.e., `setAttr(attrname, newvalue)`) to set a model attribute.

Call it with three arguments (i.e., `setAttr(attrname, objects, newvalues)`) to set attribute values for a list or dict of model objects (`Var` objects, `Constr` objects, etc.). To set the same value for all objects in the second argument, you can pass a scalar value in the third argument. If the second argument is a list, the third argument should be a list of the same length. If the second argument is a dict, the third argument should be dict with a value for every key from the second.

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `Model` object.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

Arguments:

`attrname`: Name of attribute to set.

`objs`: List of model objects (Var or Constr or ...)

`newvalue`: Desired new value(s) for attribute.

Example usage:

```
model.setAttr("objCon", 0)
model.setAttr(GRB.Attr.ObjCon, 0)
model.setAttr("LB", model.getVars(), [0]*model.numVars)
model.setAttr("RHS", model.getConstrs(), [1.0]*model.numConstrs)
model.setAttr("vType", model.getVars(), GRB.CONTINUOUS)
model.objcon = 0
```

Model.setMObjective()

```
setMObjective ( Q, c, constant, xQ_L=None, xQ_R=None, xc=None, sense=None )
```

Set the model objective equal to a quadratic (or linear) expression using matrix semantics.

Note that you will typically use overloaded operators to set the objective using matrix objects. The overloaded `@` operator can be used to build a linear matrix expression or a quadratic matrix expression, which is then passed to setObjective.

Arguments:

`Q`: The quadratic objective matrix - a NumPy 2-D dense ndarray or a SciPy sparse matrix. This can be `None` if there are no quadratic terms.

`c`: The linear constraint vector - a NumPy 1-D ndarray. This can be `None` if there are no linear terms.

`constant`: Objective constant.

`xQ_L (optional)`: Decision variables for quadratic objective terms; left multiplier for Q. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the size of the first dimension of `Q`.

`xQ_R (optional)`: Decision variables for quadratic objective terms; right multiplier for Q. The length of the argument must match the size of the second dimension of `Q`.

`xc (optional)`: Decision variables for linear objective terms. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the length of `c`.

`sense (optional)`: Optimization sense (`GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization). Omit this argument to use the `ModelSense` attribute value to determine the sense.

Example usage:

```
c = np.full(10, 1.0)
xc = model.addMVar(10)

model.setMObjective(None, c, 0.0, None, None, xc, GRB.MAXIMIZE)

Q = np.full((2, 3), 1.0)
xL = model.addMVar(2)
xR = model.addMVar(3)

model.setMObjective(Q, None, 0.0, xL, xR, None, GRB.MINIMIZE)
```

Model.setObjective()

```
setObjective ( expr, sense=None )
```

Set the model objective equal to a linear or quadratic expression (for multi-objective optimization, see setObjectiveN).

Note that you can also modify a linear model objective using the Obj variable attribute. If you wish to mix and match these two approaches, please note that this method will replace the existing objective.

Arguments:

`expr`: New objective expression. Argument can be a linear or quadratic expression (an objective of type LinExpr or QuadExpr).

`sense (optional)`: Optimization sense (`GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization). Omit this argument to use the `ModelSense` attribute value to determine the sense.

Example usage:

```
model.setObjective(x + y, GRB.MAXIMIZE)
model.setObjective(x*x + y*y)
```

Model.setObjectiveN()

```
setObjectiveN  ( expr, index, priority=0, weight=1, abstol=1e-6, reltol=0,
                 name="" )
```

Set an alternative optimization objective equal to a linear expression.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

Note that you can also modify an alternative objective using the ObjN variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `ObjN` attribute can be used to modify individual terms.

Arguments:

`expr (LinExpr)`: New alternative objective.

`index (int)`: Index for new objective. If you use an index of 0, this routine will change the primary optimization objective.

`priority (int, optional)`: Priority for the alternative objective. This initializes the ObjNPriority attribute for this objective.

`weight (float, optional)`: Weight for the alternative objective. This initializes the ObjNWeight attribute for this objective.

`abstol (float, optional)`: Absolute tolerance for the alternative objective. This initializes the ObjNAbsTol attribute for this objective.

`reltol (float, optional)`: Relative tolerance for the alternative objective. This initializes the ObjNRelTol attribute for this objective.

`name (string, optional)`: Name of the alternative objective. This initializes the ObjNName attribute for this objective. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# Primary objective: x + 2 y
model.setObjectiveN(x + 2*y, 0, 2)
# Alternative, lower priority objectives: 3 y + z and x + z
model.setObjectiveN(3*y + z, 1, 1)
model.setObjectiveN(x + z, 2, 0)
```

Model.setPWLObj()

```
setPWLObj  ( var, x, y )
```

Set a piecewise-linear objective function for a variable.

The arguments to this method specify a list of points that define a piecewise-linear objective function for a single variable. Specifically, the x and y arguments give coordinates for the vertices of the function.

For additional details on piecewise-linear objective functions, refer to this discussion.

Arguments:

`var`: A Var object that gives the variable whose objective function is being set.

x: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

y: The y values for the points that define the piecewise-linear function.

Example usage:

```
model.setPWLObj(var, [1, 3, 5], [1, 2, 4])
```

Model.setParam()

```
setParam ( paramname, newvalue )
```

Set the value of a parameter to a new value. Note that this method only affects the parameter setting for this model. Use global function setParam to change the parameter for all models.

You can also set parameters using the `Model.Params` class. For example, to set parameter `MIPGap` to value 0 for model `m`, you can do either `m.setParam('MIPGap', 0)` or `m.Params.MIPGap=0`.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Arguments:

paramname: String containing the name of parameter that you would like to modify. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed and none are modified. Note that case is ignored.

newvalue: Desired new value for parameter. Can be 'default', which indicates that the parameter should be reset to its default value.

Example usage:

```
model.setParam("heu*", 0.5)
model.setParam(GRB.Param.heuristics, 0.5)
model.setParam("heu*", "default")
```

Model.singleScenarioModel()

```
singleScenarioModel ( )
```

Capture a single scenario from a multi-scenario model. Use the ScenarioNumber parameter to indicate which scenario to capture.

The model on which this method is invoked must be a multi-scenario model, and the result will be a single-scenario model.

Return value:

Model for a single scenario.

Example usage:

```
model.params.ScenarioNumber = 0
s = model.singleScenarioModel()
```

Model.terminate()

```
terminate ( )
```

Generate a request to terminate the current optimization. This method can be called at any time during an optimization (from a callback, from another thread, from an interrupt handler, etc.). Note that, in general, the request won't be acted upon immediately.

When the optimization stops, the Status attribute will be equal to `GRB_INTERRUPTED`.

Example usage:

```
model.terminate()
```

Model.tune()

```
tune ( )
```

Perform an automated search for parameter settings that improve performance. Upon completion, this method stores the best parameter sets it found. The number of stored parameter sets can be determined by querying the value of the TuneResultCount attribute. The actual settings can be retrieved using getTuneResult.

Please refer to the parameter tuning section for details on the tuning tool.

Example usage:

```
model.tune()
```

Model.update()

```
update ( )
```

Process any pending model modifications.

Example usage:

```
model.update()
```

Model.write()

```
write ( filename )
```

This method is the general entry point for writing optimization data to a file. It can be used to write optimization models, solutions vectors, basis vectors, start vectors, or parameter settings. The type of data written is determined by the file suffix. File formats are described in the File Format section.

Note that writing a model to a file will process all pending model modifications. This is also true when writing other model information such as solutions, bases, etc.

Note also that when you write a Gurobi parameter file (PRM), both integer or double parameters not at their default value will be saved, but no string parameter will be saved into the file.

Arguments:

`filename`: The name of the file to be written. The file type is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, or `.rlp` for writing the model itself, `.dua` or `.dlp` for writing the dualized model (only pure LP), `.ilp` for writing just the IIS associated with an infeasible model (see Model.computeIIS for further information), `.sol` for writing the solution selected by the SolutionNumber parameter, `.mst` for writing a start vector, `.hnt` for writing a hint file, `.bas` for writing an LP basis, `.prm` for writing modified parameter settings, `.attr` for writing model attributes, or `.json` for writing

solution information in JSON format. If your system has compression utilities installed (e.g., `7z` or `zip` for Windows, and `gzip`, `bzip2`, or `unzip` for Linux or macOS), then the files can be compressed, so additional suffixes of `.gz`, `.bz2`, or `.7z` are accepted.

Example usage:

```
model.write("out.mst")
model.write("out.sol")
```

7.3 Var

Gurobi variable object. Variables are always associated with a particular model. You create a variable object by adding a variable to a model (using Model.addVar), rather than by using a `Var` constructor.

Variable objects have a number of attributes. The full list can be found in the Attributes section of this document. Some variable attributes can only be queried, while others can also be set. Recall that the Gurobi Optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

We should point out a few things about variable attributes. Consider the `lb` attribute. Its value can be queried using `var.lb`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `var.LB`. It can be set using a standard assignment statement (e.g., `var.lb = 0`). However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `x` attribute), so attempts to assign new values to them will raise an exception.

You can also use Var.getAttr/ Var.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.LB`).

To build expressions using variable objects, you generally use operator overloading. You can build either linear or quadratic expressions:

```
expr1 = x + 2 * y + 3 * z + 4.0
expr2 = x ** 2 + 2 * x * y + 3 * z + 4.0
```

The first expression is linear, while the second is quadratic. An expression is typically then passed to setObjective (to set the optimization objective) or addConstr (to add a constraint).

Var.getAttr()

```
getAttr ( attrname )
```

Query the value of a variable attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `Var` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(var.getAttr(GRB.Attr.X))
print(var.getAttr("x"))
```

Var.sameAs()

```
sameAs ( var2 )
```

Check whether two variable objects refer to the same variable.

Arguments:

`var2`: The other variable.

Return value:

Boolean result indicates whether the two variable objects refer to the same model variable.

Example usage:

```
print(model.getVars()[0].sameAs(model.getVars()[1]))
```

Var.index

```
index
```

This property returns the current index, or order, of the variable in the underlying constraint matrix.

Return value:

-2: removed, -1: not in model, otherwise: index of the variable in the model

Note that the index of a variable may change after subsequent model modifications.

Example usage:

```
v = model.getVars()[0]
print(v.index) # Index will be 0
```

Var.setAttr()

```
setAttr  ( attrname, newvalue )
```

Set the value of a variable attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `Var` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

Example usage:

```
var.setAttr(GRB.Attr.UB, 0.0)
var.setAttr("ub", 0.0)
```

7.4 MVar

Gurobi matrix variable object. An `MVar` is a NumPy ndarray of Gurobi variables. Variables are always associated with a particular model. You typically create these objects using Model.addMVar.

Many concepts, properties and methods of the MVar class lean on equivalents in NumPy's ndarray class. For explanations of concepts like shape, dimensions, or broadcasting, we refer you to the NumPy documentation.

You generally use `MVar` objects to build matrix expressions, typically using overloaded operators. You can build linear matrix expressions or quadratic matrix expressions:

```
expr1 = A @ x
expr2 = A @ x + B @ y + z
expr3 = x @ A @ x + y @ B @ y
```

The first two expressions are linear, while the third is quadratic.

In the examples above (and in general), A and B can be NumPy ndarray objects or any of the sparse matrix classes defined in SciPy.sparse. Dimensions of the operands must compatible, in the usual sense of Python's matrix multiplication operator. For example, in the expression $A@x$, both A and x must have at least one dimension, and their inner-most dimensions must agree. For a complete description of shape compatibility rules, we refer you to Python's documentation

An expression is typically then passed to setObjective (to set the optimization objective) or addConstr (to add a constraint).

MVar objects support standard NumPy indexing and slicing. An MVar of size 1 can be passed in all places where gurobipy accepts a Var object.

Variable objects have a number of attributes. The full list can be found in the Attributes section of this document. Some variable attributes can only be queried, while others can also be set. Recall that the Gurobi Optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

We should point out a few things about variable attributes. Consider the `lb` attribute. Its value can be queried using `mvar.lb`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `var.LB`. Attribute values are returned as a NumPy `ndarray` that has the same shape as `mvar`, where each element contains the attribute value for the corresponding element of the `MVar` object. An attribute can be set, using a standard assignment statement (e.g., `var.lb = l`), with `l` being either an `ndarray` with the appropriate shape, or a scalar which is then applied to all of the associated variables. However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `x` attribute), so attempts to assign new values to them will raise an exception.

You can also use MVar.getAttr/ MVar.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.LB`).

MVar.copy()

```
copy ( )
```

Create a copy of this MVar.

Return value:

The new object.

Example usage:

```
orig = model.addMVar(3)
copy = orig.copy()
```

MVar.diagonal()

```
diagonal  ( offset=0, axis1=0, axis2=1 )
```

Create an MVar corresponding to the variables on the specified diagonal of this MVar.

Arguments:

offset (optional): Offset of the diagonal w.r.t. the main diagonal. Values >0 mean above it, and values <0 below it

axis1 (optional): Axis to be used as the first axis of the 2-D sub-MVar from which the diagonal should be taken. Defaults to 0. You need to consider this argument only for MVar objects with more than 2 dimensions.

axis2 (optional): Axis to be used as the second axis of the 2-D sub-MVar from which the diagonal should be taken. Defaults to 1. You need to consider this argument only for MVar objects with more than 2 dimensions.

Return value:

An MVar representing the requested diagonal of this MVar.

Example usage:

```
x = model.addMVar((8, 8))
diag_main = x.diagonal()  # The main diagonal of x
diag_sup = x.diagonal(1)  # The first superdiagonal of x
diag_sup = x.diagonal(-2)  # The second subdiagonal of x
adiag_main = x[:, ::-1].diagonal()  # The main anti-diagonal of x
```

MVar.fromlist()

```
fromlist  ( varlist )
```

Convert a list of variables into an MVar object. The shape is inferred from the contents of the list - a list of Var objects produces a 1-D MVar object, a list of lists of Var objects produces a 2-D MVar, etc.

Arguments:

varlist: A list of Var objects to populate the returned MVar.

Return value:

MVar object corresponding to the input variables.

Example usage:

```
x0 = model.addVar()
x1 = model.addVar()
x2 = model.addVar()
x3 = model.addVar()
x_1d = MVar.fromlist([x0, x1, x2, x3])  # 1-D MVar
x_2d = MVar.fromlist([[x0, x1], [x2, x3]])  # 2-D MVar
```

MVar.fromvar()

```
fromvar ( var )
```

Convert a Var object into a 0-dimensional MVar object.

Arguments:

var: The variable object to populate the returned MVar.

Return value:

MVar object corresponding to the input variable.

Example usage:

```
x = model.addVar()
x_as_mvar = MVar.fromvar(x)
```

MVar.getAttr()

```
getAttr ( attrname )
```

Query the value of an attribute for a matrix variable. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `MVar` object (e.g., it was removed from the model).

The result is returned in a NumPy ndarray with the same shape as the `MVar` object.

Arguments:

attrname: The attribute being queried.

Return value:

Current values for the requested attribute.

Example usage:

```
print(var.getAttr(GRB.Attr.X))
print(var.getAttr("x"))
```

MVar.item()

```
item ( )
```

For an MVar that contains a single element, returns a copy of that element as a Var object. Calling this method on an MVar with more than one element will raise a ValueError.

Return value:

A Var object

Example usage:

```
x = model.addMVar((2, 2))
x_sub = x[0, 1]  # A 0-D MVar encapsulating one Var object
x_var = x[0, 1].item()  # The resident Var object itself
```

MVar.ndim

```
ndim
```

The number of dimensions in this matrix variable.

Return value:

An int

Example usage:

```
x1 = model.addMVar((3,))
print(x1.ndim)  #  "1"
x2 = model.addMVar((1, 3))
print(x2.ndim)  #  "2"
```

MVar.reshape()

```
reshape  ( shape, order='C' )
```

Return a copy of this MVar with the same variables, but with a new shape.

Arguments:

shape: An int, or a tuple of int. The new shape should be compatible with this MVar's shape. The special value of -1 can be passed in at one position, which then infers the length of that dimension from the overall number of Var objects in the MVar and the provided lengths of the other dimensions.

order (optional): A string 'C' or 'F'. Read the elements of this MVar using C-like ('C') or Fortran-like ('F') order, and write the elements into the reshaped array in this order.

Return value:

An MVar of requested shape

Example usage:

```
x = model.addMVar((2, 2))
x_vec = x.reshape(-1, order='C')  # 1-D result, rows of x stacked
x_vec = x.reshape(-1, order='F')  # 1-D result, columns of x stacked
```

MVar.setAttr()

```
setAttr  ( attrname, newvalue )
```

Set the value of a matrix variable attribute.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `MVar` object (e.g., it was removed from the model).

Arguments:

attrname: The attribute being modified.

newvalue: The desired new value of the attribute. The shape must be the same as the `MVar` object. Alternatively, you can pass a scalar argument, which will automatically be promoted to have the right shape.

Example usage:

```
var.setAttr("ub", np.full((5,), 0)
```

```
var.setAttr(GRB.Attr.UB, 0.0)
var.setAttr("ub", 0.0)
```

MVar.shape

`shape`

The shape of this MVar.

Return value:

A tuple of int

Example usage:

```
x1 = model.addMVar((3,))
print(x1.shape)  #  "(3,)"
x2 = model.addMVar((1, 3))
print(x2.shape)  #  "(1, 3)"
```

MVar.size

`size`

The total number of elements in this matrix variable.

Return value:

An int

Example usage:

```
x1 = model.addMVar((3,))
print(x1.size)  #  "3"
x2 = model.addMVar((2, 3))
print(x2.size)  #  "6"
```

MVar.sum()

`sum  ( axis=None )`

Sum the elements of the MVar; returns an MLinExpr object.

Arguments:

`axis`: An int, or None. Sum along the specified axis. If set to None, summation takes place along all axes of this MVar.

Return value:

An MLinExpr representing the sum.

Example usage:

```
x = model.addMVar((2, 2))
sum_row = x.sum(axis=0)  # Sum along the rows of X
sum_col = x.sum(axis=1)  # Sum along the columns of X
sum_all = x.sum()  # Sum all variables in this MVar
```

MVar.T

```
T
```

Synonymous property for transpose.

Example usage:

```
x = model.addMVar((4, 1))  # Resembles a "column vector"
x_t = x.T  # Same variables, but as a "row vector"
y = model.addMVar((3, 2))
y_t = x.T  # Has shape (2, 3)
```

MVar.tolist()

```
tolist  (  )
```

Return the variables associated with this matrix variable as a list of individual Var objects.

Return value:

List of `Var` objects.

Example usage:

```
mvar = model.addMVar(5)
varlist = mvar.tolist()
# Do something with the Var corresponding to mvar[3]
print(varlist[3])
```

MVar.transpose()

```
transpose  (  )
```

Transpose this MVar; create a new MVar object by reversing the order of the original MVar's axes. For 1-D MVar objects, this routine simply returns a copy of the original MVar.

Return value:

An MVar object representing the transpose.

Example usage:

```
x = model.addMVar((4, 1))  # Resembles a "column vector"
x_t = x.transpose()  # Same variables, but as a "row vector"
y = model.addMVar((3, 2))
y_t = x.transpose()  # Has shape (2, 3)
```

7.5 Constr

Gurobi constraint object. Constraints are always associated with a particular model. You create a constraint object by adding a constraint to a model (using Model.addConstr), rather than by using a `Constr` constructor.

Constraint objects have a number of attributes. The full list can be found in the Attributes section of this document. Some constraint attributes can only be queried, while others can also be set. Recall that the Gurobi Optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

We should point out a few things about constraint attributes. Consider the `rhs` attribute. Its value can be queried using `constr.rhs`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `constr.rhs`. It can be set using a standard assignment statement (e.g., `constr.rhs = 0`). However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `Pi` attribute), so attempts to assign new values to them will raise an exception.

You can also use Constr.getAttr/ Constr.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.RHS`).

Constr.getAttr()

```
getAttr ( attrname )
```

Query the value of a constraint attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `Constr` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(constr.getAttr(GRB.Attr.Slack))
print(constr.getAttr("slack"))
```

Constr.index

```
index
```

This property returns the current index, or order, of the constraint in the underlying constraint matrix.

Note that the index of a constraint may change after subsequent model modifications.

Return value:

-2: removed, -1: not in model, otherwise: index of the constraint in the model

Example usage:

```
c = model.getConstrs()[0]
print(c.index) # Index will be 0
```

Constr.sameAs()

```
sameAs ( constr2 )
```

Check whether two constraint objects refer to the same constraint.

Arguments:

constr2: The other constraint.

Return value:

Boolean result indicates whether the two constraint objects refer to the same model constraint.

Example usage:

```
print(model.getConstrs()[0].sameAs(model.getConstrs()[1]))
```

Constr.setAttr()

```
setAttr ( attrname, newvalue )
```

Set the value of a constraint attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `Constr` object (e.g., it was removed from the model).

Arguments:

attrname: The attribute being modified.

newvalue: The desired new value of the attribute.

Example usage:

```
constr.setAttr(GRB.Attr.RHS, 0.0)
constr.setAttr("rhs", 0.0)
```

7.6 MConstr

Gurobi matrix constraint object. An `MConstr` object is an array-like data structure that represents multiple linear constraints (in contrast to a Constr object, which represents a single constraint). It behaves similar to NumPy's `ndarray`s, e.g., it has a shape and can be indexed and sliced. Matrix constraints are always associated with a particular model. You typically create these objects with Model.addConstr, using overloaded comparison operators on matrix variables and linear matrix expressions, or with the method Model.addMConstr.

Constraint objects have a number of attributes. The full list can be found in the Attributes section of this document. Some constraint attributes can only be queried, while others can also be set. Recall that the Gurobi Optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

We should point out a few things about constraint attributes. Consider the `rhs` attribute. The values for a matrix constraint `mc` can be queried using `mc.rhs`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `mc.RHS`. Attribute values are returned as a NumPy `ndarray` that has the same shape as `mc`. An attribute can be set, using a standard assignment statement (e.g., `constr.rhs = b`), with `b` being either an `ndarray` with the appropriate shape, or a scalar which is then applied to all of the associated constraints. However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `Pi` attribute), so attempts to assign new values to them will raise an exception.

You can also use MConstr.getAttr/ MConstr.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.RHS`).

MConstr.fromlist()

```
fromlist ( constrlist )
```

Convert a list of constraints into an `MConstr` object. The shape is inferred from the contents of the list - a list of `Constr` objects produces a 1-D `MConstr` object, a list of lists of `Constr` objects produces a 2-D `MConstr`, etc.

Arguments:

`constrlist`: A list of Constr objects to populate the returned MConstr.

Return value:

MConstr object corresponding to the input constraints.

Example usage:

```
constrs = model.getConstrs()
mc = MConstr.fromlist(constrs)  # 1-D MConstr
```

MConstr.getAttr()

```
getAttr ( attrname )
```

Query the value of an attribute for a matrix constraint. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `MConstr` object (e.g., it was removed from the model).

The result is returned as a NumPy `ndarray` with the same shape as the `MConstr` object.

Arguments:

`attrname`: The attribute being queried.

Return value:

`ndarray` of current values for the requested attribute.

Example usage:

```
mc = model.addConstr(A @ x <= b)
rhs = mc.getAttr("RHS")
```

MConstr.setAttr()

```
setAttr  ( attrname, newvalue )
```

Set the value of a matrix constraint attribute.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `MConstr` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being modified.

`newvalue`: `ndarray` of desired new values for the attribute. The shape must be the same as the `MConstr` object. Alternatively, you can pass a scalar argument, which will automatically be promoted to have the right shape.

Example usage:

```
mc = model.addConstr(A @ x <= b)
mc.setAttr("RHS", np.arange(A.shape[0]))
mc.setAttr(GRB.Attr.RHS, 0.0)  # broadcast
```

MConstr.tolist()

```
tolist  (  )
```

Return the constraints associated with this matrix constraint as a list of individual Constr objects.

Return value:

List of `Constr` objects.

Example usage:

```
mc = model.addConstr(A @ x <= b)
constrlist = mc.tolist()
# Do something with the Constr corresponding to mc[3]
```

```
print(constrlist[3])
```

7.7 MQConstr

Gurobi matrix quadratic constraint object. An `MQConstr` object is an array-like data structure that represents multiple quadratic constraints (in contrast to a QConstr object, which represents a single quadratic constraint). It behaves similar to NumPy's `ndarrays`, e.g., it has a shape and can be indexed and sliced. Matrix quadratic constraints are always associated with a particular model. You typically create these objects with Model.addConstr, using overloaded comparison operators on matrix variables, matrix linear expressions, and matrix quadratic expressions.

Quadratic constraint objects have a number of attributes. The full list can be found in the Attributes section of this document. Some constraint attributes can only be queried, while others can also be set. Recall that the Gurobi Optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

We should point out a few things about quadratic constraint attributes. Consider the `QCRHS` attribute. The values for a matrix quadratic constraint `mqc` can be queried using `mc.QCRHS`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `mc.qcrhs`. Attribute values are returned as a NumPy `ndarray` that has the same shape as `qmc`. An attribute can be set, using a standard assignment statement (e.g., `mqc.qcrhs = b`), with `b` being either an `ndarray` with the appropriate shape, or a scalar which is then applied to all of the associated quadratic constraints. However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `QCPi` attribute), so attempts to assign new values to them will raise an exception.

You can also use MQConstr.getAttr/ MQConstr.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.QCRHS`).

MQConstr.fromlist()

```
fromlist  ( qconstrlist )
```

Convert a list of quadratic constraints into an `MQConstr` object. The shape is inferred from the contents of the list - a list of `QConstr` objects produces a 1-D `MQConstr` object, a list of lists of `QConstr` objects produces a 2-D `MQConstr`, etc.

Arguments:

`constrlist`: A list of QConstr objects to populate the returned MQConstr.

Return value:

MQConstr object corresponding to the input constraints.

Example usage:

```
qconstrs = model.getQConstrs()
mqc = MQConstr.fromlist(qconstrs)  # 1-D MQConstr
```

MQConstr.getAttr()

```
getAttr  ( attrname )
```

Query the value of an attribute for a matrix quadratic constraint. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `MQConstr` object (e.g., it was removed from the model).

The result is returned as a NumPy `ndarray` with the same shape as the `MQConstr` object.

Arguments:

`attrname`: The attribute being queried.

Return value:

`ndarray` of current values for the requested attribute.

Example usage:

```
mqc = model.addConstr(x**2 + y <= 1)
qcrhs = mc.getAttr("QCRHS")
```

MQConstr.setAttr()

```
setAttr  ( attrname, newvalue )
```

Set the value of a matrix quadratic constraint attribute.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `MQConstr` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being modified.

`newvalue`: `ndarray` of desired new values for the attribute. The shape must be the same as the `MQConstr` object. Alternatively, you can pass a scalar argument, which will automatically be promoted to have the right shape.

Example usage:

```
mqc = model.addConstr(x * y - x - y <= 0)
mqc.setAttr("QCRHS", np.arange(x.size))
mqc.setAttr(GRB.Attr.RHS, 1.0)  # broadcast scalar
```

MQConstr.tolist()

```
tolist  (  )
```

Return the quadratic constraints associated with this matrix quadratic constraint as a list of individual QConstr objects.

Return value:

List of `QConstr` objects.

Example usage:

```
mqc = model.addConstr(x * y <= b)
qconstrlist = mqc.tolist()
# Do something with the QConstr corresponding to mqc[3]
```

```
print(qconstrlist[3])
```

7.8 QConstr

Gurobi quadratic constraint object. Quadratic constraints are always associated with a particular model. You create a quadratic constraint object by adding a quadratic constraint to a model (using Model.addQConstr), rather than by using a `QConstr` constructor.

Quadratic constraint objects have a number of attributes. The full list can be found in the Attributes section of this document. Some constraint attributes can only be queried, while others can also be set. Recall that the Gurobi optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, Model.write on the associated model.

We should point out a few things about quadratic constraint attributes. Consider the `qcrhs` attribute. Its value can be queried using `qconstr.qcrhs`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `qconstr.QCRHS`. It can be set using a standard assignment statement (e.g., `qconstr.qcrhs = 0`). However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `qcpi` attribute), so attempts to assign new values to them will raise an exception.

You can also use QConstr.getAttr/ QConstr.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.QCRHS`).

QConstr.getAttr()

```
getAttr ( attrname )
```

Query the value of a quadratic constraint attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `QConstr` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(qconstr.getAttr(GRB.Attr.QCSense))
print(qconstr.getAttr("qcsense"))
```

QConstr.setAttr()

```
setAttr ( attrname, newvalue )
```

Set the value of a quadratic constraint attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `QConstr` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

Example usage:

```
constr.setAttr(GRB.Attr.QCRHS, 0.0)
constr.setAttr("qcrhs", 0.0)
```

7.9 SOS

Gurobi SOS constraint object. SOS constraints are always associated with a particular model. You create an SOS object by adding an SOS constraint to a model (using Model.addSOS), rather than by using an SOS constructor. Similarly, SOS constraints are removed using the Model.remove method.

An SOS constraint can be of type 1 or 2 (`GRB.SOS_TYPE1` or `GRB.SOS_TYPE2`). A type 1 SOS constraint is a set of variables where at most one variable in the set may take a value other than zero. A type 2 SOS constraint is an ordered set of variables where at most two variables in the set may take non-zero values. If two take non-zero values, they must be contiguous in the ordered set.

SOS constraint objects have one attribute, `IISSOS`, which can be queried with the SOS.getAttr method.

SOS.getAttr()

```
getAttr ( attrname )
```

Query the value of an SOS attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `SOS` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(sos.getAttr(GRB.Attr.IISSOS))
```

SOS.index

```
index
```

This property returns the current index, or order, of the SOS constraint in the underlying model.

Note that the index of an SOS constraint may change after subsequent model modifications.

Return value:

-2: removed, -1: not in model, otherwise: index of the SOS constraint in the model

SOS.setAttr()

```
setAttr ( attrname, newvalue )
```

Set the value of an SOS attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `SOS` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

Example usage:

```
sos.setAttr(GRB.Attr.IISSOSForce, 1)
var.setAttr("IISSOSForce", 0.0)
```

7.10 GenConstr

Gurobi general constraint object. General constraints are always associated with a particular model. You add a general constraint to a model either by using one of the Model.addGenConstrXxx method, or by using Model.addConstr or Model.addConstrs plus a general constraint helper function).

General constraint objects have a number of attributes, which can be queried with the GenConstr.getAttr method. The full list can be found in the Attributes section of this document.

GenConstr.getAttr()

```
getAttr ( attrname )
```

Query the value of a general constraint attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `GenConstr` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(genconstr.getAttr(GRB.Attr.GenConstrType))
print(genconstr.getAttr("GenConstrType"))
```

GenConstr.setAttr()

```
setAttr ( attrname, newvalue )
```

Set the value of a general constraint attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `GenConstr` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

7.11 MGenConstr

Gurobi matrix general constraint object. An `MGenConstr` object is an array-like data structure that represents multiple general constraints (in contrast to a GenConstr object, which represents a single general constraint). It behaves similar to NumPy's `ndarrays`, e.g., it has a shape and can be indexed and sliced. Matrix general constraints are always associated with a particular

model. Currently only the Model.addGenConstrIndicator method produces `MGenConstr` objects when matrix-friendly objects are used as inputs.

General constraint objects have a number of attributes. The full list can be found in the Attributes section of this document. Some general constraint attributes can only be queried, while others can also be set. Recall that the Gurobi Optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

Note that in most cases to query general constraint data, you will need to index into the MGenConstr object and pass individual GenConstr objects to methods such as Model.getGenConstrIndicator to read back general constraint data.

MGenConstr.fromlist()

```
fromlist  ( genconstrlist )
```

Convert a list of general constraints into an MGenConstr object. The shape is inferred from the contents of the list - a list of GenConstr objects produces a 1-D MGenConstr object, a list of lists of GenConstr objects produces a 2-D MGenConstr, etc.

Arguments:

`genconstrlist`: A list of GenConstr objects to populate the returned MGenConstr.

Return value:

MGenConstr object corresponding to the input general constraints.

Example usage:

```
gc0, gc1, gc2, gc3 = model.getGenConstrs()
mgc_1d = MGenConstr.fromlist([gc0, gc1, gc2, gc3])  # 1-D MGenConstr
mgc_2d = MGenConstr.fromlist([[gc0, gc1], [gc2, gc3]])  # 2-D MGenConstr
```

MGenConstr.getAttr()

```
getAttr  ( attrname )
```

Query the value of an attribute for a matrix general constraint. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried. Raises a `GurobiError` if there is a problem with the `MGenConstr` object (e.g., it was removed from the model).

The result is returned as a NumPy `ndarray` with the same shape as the `MGenConstr` object.

Arguments:

`attrname`: The attribute being queried.

Return value:

`ndarray` of current values for the requested attribute.

Example usage:

```
mgc = model.addGenConstrIndicator(z, 1.0, A @ x <= b)
model.computeIIS()
iis = mc.getAttr("IISGenConstr")
```

MGenConstr.setAttr()

```
setAttr ( attrname, newvalue )
```

Set the value of a matrix general constraint attribute.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set. Raises a `GurobiError` if there is a problem with the `MGenConstr` object (e.g., it was removed from the model).

Arguments:

`attrname`: The attribute being modified.

`newvalue`: `ndarray` of desired new values for the attribute. The shape must be the same as the `MGenConstr` object. Alternatively, you can pass a scalar argument, which will automatically be promoted to have the right shape.

Example usage:

```
mgc = model.addGenConstrIndicator(z, 1.0, A @ x <= b)
iis = mc.setAttr("IISGenConstrForce", 1)
model.computeIIS()
```

MGenConstr.tolist()

```
tolist ( )
```

Return the general constraints associated with this matrix general constraint as a list of individual GenConstr objects.

Return value:

List of `GenConstr` objects.

Example usage:

```
mgc = model.addGenConstrIndicator(z, True, A @ x <= b)
genconstrlist = mgc.tolist()
```

7.12 LinExpr

Gurobi linear expression object. A linear expression consists of a constant term, plus a list of coefficient-variable pairs that capture the linear terms. Linear expressions are used to build linear objective and constraints. They are temporary objects that typically have short lifespans.

You generally build linear expressions using overloaded operators. For example, if `x` is a Var object, then `x + 1` is a LinExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x + 2 * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x`, or `expr2 -= expr1`).

The full list of overloaded operators on LinExpr objects is as follows: `+`, `+=`, `-`, `-=`, `*`, `*=`, `/`, and `**` (only for exponent 2). In Python parlance, we've defined the following `LinExpr` functions: `__add__`, `__radd__`, `__iadd__`, `__sub__`, `__rsub__`, `__isub__`, `__neg__`, `__mul__`, `__rmul__`, `__imul__`, `__div__`, and `__pow__`.

We've also overloaded the comparison operators (==, <=, and >=), to make it easier to build constraints from linear expressions.

You can also use add or addTerms to modify expressions. The LinExpr() constructor can also be used to build expressions. Another option is quicksum; it is a more efficient version of the Python `sum` function. Terms can be removed from an expression using remove.

Given all these options for building expressions, you may wonder which is fastest. For small expressions, you won't need to worry about performance differences between them. If you are building lots of very large expressions (100s of terms), you will find that the LinExpr() constructor is generally going to be fastest, followed by the addTerms method, and then the quicksum function.

To add a linear constraint to your model, you generally build one or two linear expression objects (`expr1` and `expr2`) and then use an overloaded comparison operator to build an argument for Model.addConstr. To give a few examples:

$$\text{model.addConstr}(expr1 <= expr2)$$
$$\text{model.addConstr}(expr1 == 1)$$
$$\text{model.addConstr}(2 * x + 3 * y <= 4)$$

Once you add a constraint to your model, subsequent changes to the expression object you used to build the constraint will not change the constraint (you would use Model.chgCoeff for that).

Individual terms in a linear expression can be queried using the getVar, getCoeff, and getConstant methods. You can query the number of terms in the expression using the size method.

Note that a linear expression may contain multiple terms that involve the same variable. These duplicate terms are merged when creating a constraint from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar).

LinExpr()

```
LinExpr  ( arg1=0.0, arg2=None )
```

Linear expression constructor. For linear expressions of moderate size (only a few terms) and for the ease of usage, you should generally use overloaded operators instead of the explicit constructor to build linear expression objects.

This constructor takes multiple forms. You can initialize a linear expression using a constant (`LinExpr(2.0)`), a variable (`LinExpr(x)`), an expression (`LinExpr(2*x)`), a pair of lists containing coefficients and variables, respectively (`LinExpr([1.0, 2.0], [x, y])`), or a list of coefficient-variable tuples (`LinExpr([(1.0, x), (2.0, y), (1.0, z)])`).

Return value:

A linear expression object.

Example usage:

```
expr = LinExpr(2.0)
expr = LinExpr(2*x)
expr = LinExpr([1.0, 2.0], [x, y])
expr = LinExpr([(1.0, x), (2.0, y), (1.0, z)])
```

LinExpr.add()

```
add  ( expr, mult=1.0 )
```

Add one linear expression into another. Upon completion, the invoking linear expression will be equal to the sum of itself and the argument expression.

Arguments:

`expr`: Linear expression to add.

`mult (optional)`: Multiplier for argument expression.

Example usage:

```
e1 = x + y
e1.add(z, 3.0)
```

LinExpr.addConstant()

```
addConstant  ( c )
```

Add a constant into a linear expression.

Arguments:

`c`: Constant to add to expression.

Example usage:

```
expr = x + 2 * y
expr.addConstant(0.1)
```

LinExpr.addTerms()

```
addTerms  ( coeffs, vars )
```

Add new terms into a linear expression.

Arguments:

`coeffs`: Coefficients for new terms; either a list of coefficients or a single coefficient. The two arguments must have the same size.

`vars`: Variables for new terms; either a list of variables or a single variable. The two arguments must have the same size.

Example usage:

```
expr.addTerms(1.0, x)
expr.addTerms([2.0, 3.0], [y, z])
```

LinExpr.clear()

```
clear ( )
```

Set a linear expression to 0.

Example usage:

```
expr.clear()
```

LinExpr.copy()

```
copy ( )
```

Copy a linear expression

Return value:

Copy of input expression.

Example usage:

```
e0 = 2 * x + 3
e1 = e0.copy()
```

LinExpr.getConstant()

```
getConstant ( )
```

Retrieve the constant term from a linear expression.

Return value:

Constant from expression.

Example usage:

```
e = 2 * x + 3
print(e.getConstant())
```

LinExpr.getCoeff()

```
getCoeff ( i )
```

Retrieve the coefficient from a single term of the expression.

Return value:

Coefficient for the term at index `i` in the expression.

Example usage:

```
e = x + 2 * y + 3
print(e.getCoeff(1))
```

LinExpr.getValue()

```
getValue ( )
```

Compute the value of an expression using the current solution.

Return value:

The value of the expression.

Example usage:

```
obj = model.getObjective()
print(obj.getValue())
```

LinExpr.getVar()

```
getVar ( i )
```

Retrieve the variable object from a single term of the expression.

Return value:

Variable for the term at index `i` in the expression.

Example usage:

```
e = x + 2 * y + 3
print(e.getVar(1))
```

LinExpr.remove()

```
remove ( item )
```

Remove a term from a linear expression.

Arguments:

`item`: If `item` is an integer, then the term stored at index `item` of the expression is removed. If `item` is a Var, then all terms that involve `item` are removed.

Example usage:

```
e = x + 2 * y + 3
e.remove(x)
```

LinExpr.size()

```
size ( )
```

Retrieve the number of terms in the linear expression (not including the constant).

Return value:

Number of terms in the expression.

Example usage:

```
e = x + 2 * y + 3
print(e.size())
```

LinExpr.__eq__()

```
__eq__  ( )
```

Overloads the == operator, creating a TempConstr object that captures an equality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x + y == 1)
```

LinExpr.__le__()

```
__le__  ( )
```

Overloads the <= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x + y <= 1)
```

LinExpr.__ge__()

```
__ge__  ( arg )
```

Overloads the >= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x + y >= 1)
```

7.13 QuadExpr

Gurobi quadratic expression object. A quadratic expression consists of a linear expression plus a list of coefficient-variable-variable triples that capture the quadratic terms. Quadratic expressions are used to build quadratic objective functions and quadratic constraints. They are temporary objects that typically have short lifespans.

You generally build quadratic expressions using overloaded operators. For example, if `x` is a Var object, then `x * x` is a QuadExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x *x + 2 * x * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x * x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x * x`, or `expr2 -= expr1`).

The full list of overloaded operators on QuadExpr objects is as follows: `+`, `+=`, `-`, `-=`, `*`, `*=`, and `/`. In Python parlance, we've defined the following `QuadExpr` functions: `__add__`, `__radd__`, `__iadd__`, `__sub__`, `__rsub__`, `__isub__`, `__neg__`, `__mul__`, `__rmul__`, `__imul__`, and `__div__`.

We've also overloaded the comparison operators (==, <=, and >=), to make it easier to build constraints from quadratic expressions.

You can use quicksum to build quadratic expressions; it is a more efficient version of the Python `sum` function. You can also use add or addTerms to modify expressions. Terms can be removed from an expression using remove.

Given all these options for building expressions, you may wonder which is fastest. For small expressions, you won't need to worry about performance differences between them. If you are building lots of very large expressions (100s of terms), you will find that a single call to addTerms is fastest. Next would be a call to quicksum, followed by a series of calls to expr.add(x*x).

To add a quadratic constraint to your model, you generally build one or two quadratic expression objects (`qexpr1` and `qexpr2`) and then use an overloaded comparison operator to build an argument for Model.addConstr. To give a few examples:

$$\text{model.addConstr}(qexpr1 <= qexpr2)$$
$$\text{model.addConstr}(qexpr1 == 1)$$
$$\text{model.addConstr}(2 * x * x + 3 * y * y <= 4)$$

Once you add a constraint to your model, subsequent changes to the expression object you used to build the constraint will have no effect on that constraint.

Individual quadratic terms in a quadratic expression can be queried using the getVar1, getVar2, and getCoeff methods. You can query the number of quadratic terms in the expression using the size method. To query the constant and linear terms associated with a quadratic expression, use getLinExpr to obtain the linear portion of the quadratic expression, and then use the getVar, getCoeff, and getConstant methods on this LinExpr object. Note that a quadratic expression may contain multiple terms that involve the same variable pair. These duplicate terms are merged when creating a constraint from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar1 and getVar2).

QuadExpr()

```
QuadExpr  ( expr = None )
```

Quadratic expression constructor. Note that you should generally use overloaded operators instead of the explicit constructor to build quadratic expression objects.

Arguments:

`expr (optional)`: Initial value of quadratic expression. Can be a `LinExpr` or a `QuadExpr`. If no argument is specified, the initial expression value is 0.

Return value:

A quadratic expression object.

Example usage:

```
expr = QuadExpr()
expr = QuadExpr(2*x)
expr = QuadExpr(x*x + y+y)
```

QuadExpr.add()

```
add  ( expr, mult=1.0 )
```

Add an expression into a quadratic expression. Argument can be either a linear or a quadratic expression. Upon completion, the invoking quadratic expression will be equal to the sum of itself and the argument expression.

Arguments:

`expr`: Linear or quadratic expression to add.

`mult (optional)`: Multiplier for argument expression.

Example usage:

```
expr = x * x + 2 * y * y
expr.add(z * z, 3.0)
```

QuadExpr.addConstant()

```
addConstant  ( c )
```

Add a constant into a quadratic expression.

Arguments:

`c`: Constant to add to expression.

Example usage:

```
expr = x * x + 2 * y * y + z
expr.addConstant(0.1)
```

QuadExpr.addTerms()

```
addTerms  ( coeffs, vars, vars2=None )
```

Add new linear or quadratic terms into a quadratic expression.

Arguments:

`coeffs`: Coefficients for new terms; either a list of coefficients or a single coefficient. The arguments must have the same size.

vars: Variables for new terms; either a list of variables or a single variable. The arguments must have the same size.

vars2 (optional): Variables for new quadratic terms; either a list of variables or a single variable. Only present when you are adding quadratic terms. The arguments must have the same size.

Example usage:

```
expr.addTerms(1.0, x)
expr.addTerms([2.0, 3.0], [y, z])
expr.addTerms([2.0, 3.0], [x, y], [y, z])
```

QuadExpr.clear()

```
clear ( )
```

Set a quadratic expression to 0.

Example usage:

```
expr.clear()
```

QuadExpr.copy()

```
copy ( )
```

Copy a quadratic expression

Return value:

Copy of input expression.

Example usage:

```
e0 = x * x + 2 * y * y + z
e1 = e0.copy()
```

QuadExpr.getCoeff()

```
getCoeff ( i )
```

Retrieve the coefficient from a single term of the expression.

Return value:

Coefficient for the quadratic term at index `i` in the expression.

Example usage:

```
expr = x * x + 2 * y * y + z
print(expr.getCoeff(1))
```

QuadExpr.getLinExpr()

```
getLinExpr ( )
```

A quadratic expression is represented as a linear expression, plus a list of quadratic terms. This method retrieves the linear expression associated with the quadratic expression.

Return value:

Linear expression from quadratic expression.

Example usage:

```
expr = x * x + 2 * y * y + z
le = expr.getLinExpr()
```

QuadExpr.getValue()

```
getValue ( )
```

Compute the value of an expression using the current solution.

Return value:

The value of the expression.

Example usage:

```
obj = model.getObjective()
print(obj.getValue())
```

QuadExpr.getVar1()

```
getVar1 ( i )
```

Retrieve the first variable for a single quadratic term of the quadratic expression.

Return value:

First variable associated with the quadratic term at index `i` in the quadratic expression.

Example usage:

```
expr = x * x + 2 * y * y + z
print(expr.getVar1(1))
```

QuadExpr.getVar2()

```
getVar2 ( i )
```

Retrieve the second variable for a single quadratic term of the quadratic expression.

Return value:

Second variable associated with the quadratic term at index `i` in the quadratic expression.

Example usage:

```
expr = x * x + 2 * y * y + z
print(expr.getVar2(1))
```

QuadExpr.remove()

```
remove ( item )
```

Remove a term from a quadratic expression.

Arguments:

`item`: If `item` is an integer, then the quadratic term stored at index `item` of the expression is removed. If `item` is a Var, then all quadratic terms that involve `item` are removed.

Example usage:

```
expr = x * x + 2 * y * y + z
expr.remove(x)
```

QuadExpr.size()

```
size ( )
```

Retrieve the number of quadratic terms in the expression.

Return value:

Number of quadratic terms in the expression.

Example usage:

```
expr = x * x + 2 * y * y + z
print(expr.size())
```

QuadExpr.__eq__()

```
__eq__ ( )
```

Overloads the == operator, creating a TempConstr object that captures an equality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x*x + y*y == 1)
```

QuadExpr.__le__()

```
__le__ ( )
```

Overloads the <= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x*x + y*y <= 1)
```

QuadExpr.__ge__()

```
__ge__ ( arg )
```

Overloads the >= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x*x + y*y >= 1)
```

7.14 GenExpr

Gurobi general expression object. Objects of this class are created by a set of general constraint helper functions functions. They are temporary objects, meant to be used in conjunction with overloaded operators to build TempConstr objects, which are then passed to addConstr or addConstrs to build general constraints.

To be more specific, the following creates a `GenExpr` object...

```
max_(x, y)
```

The following creates a `TempConstr` object...

```
z == max_(x, y)
```

The following adds a general constraint to a model...

```
model.addConstr(z == max_(x, y))
```

Please refer to the TempConstr documentation for more information on building general constraints.

7.15 MLinExpr

Gurobi linear matrix expression object. A linear matrix expression results from an arithmetic operation with an MVar object. A common example is a matrix-vector product, where the matrix is a NumPy ndarray or a SciPy sparse matrix and the vector is a Gurobi MVar object. Linear matrix expressions are used to build linear objectives and constraints. They are temporary objects that typically have short lifespans.

You generally build linear matrix expressions using overloaded operators, typically by multiplying a 2-D matrix (dense or sparse) by a 1-D MVar object using the Python matrix multiply (`@`) operator (e.g., `expr = A @ x`). You can also promote an `MVar` object to an `MLinExpr` using arithmetic expressions (e.g., `expr = x + 1`). Most arithmetic operations are supported on `MLinExpr` objects, including addition and subtraction (e.g., `expr = A @ x - B @ y`), multiplication by a constant (e.g. `expr = 2 * A @ x`), and point-wise multiplication with an ndarray or a sparse matrix. An `MLinExpr` object containing empty expressions can be created using the zeros method.

An `MLinExpr` object has a `shape` representing its dimensions, a `size` that counts the total number of elements, and an `ndim` that gives the number of dimensions. These properties lean on their counterparts in NumPy's ndarray class.

When working with `MLinExpr` objects, you need to make sure that the operands' shapes are compatible. For matrix multiplication, we follow the rules of Python's matrix multiplication operator: both operands need to have at least one dimension, and their inner dimensions must agree. For more information we refer you to Python's documentation. Other binary operations such as addition and multiplication are straightforward to understand if both operands have the same shape: the operation is applied point wise on the matching indices. For operands that have different shapes, the arithmetic follows NumPy's broadcasting rules. We refer you to the NumPy documentation for more information.

The full list of overloaded operators on MLinExpr objects is as follows: `+`, `+=`, `-`, `-=`, `*`, `*=`, and `@`. In Python parlance, we've defined the following `MLinExpr` functions: `__add__`, `__radd__`, `__iadd__`, `__sub__`, `__rsub__`, `__isub__`, `__neg__`, `__mul__`, `__rmul__`, `__imul__`, `__matmul__`, and `__rmatmul__`.

We've also overloaded the comparison operators (==, <=, and >=), to make it easier to build constraints from linear matrix expressions.

MLinExpr.clear()

```
clear ( )
```

Reset this expression to all zeros.

Example usage:

```
expr = 2 * model.addMVar(3) + 1
expr.clear()  # All three entries are reset to constant 0.0
```

MLinExpr.copy()

```
copy ( )
```

Create a copy of a linear matrix expression.

Return value:

Copy of expression object.

Example usage:

```
orig = 2 * model.addMVar(3) + 1.0
copy = orig.copy()
copy += 2.0  # Leaves 'orig' untouched
```

MLinExpr.getValue()

```
getValue ( )
```

Compute the value of a linear matrix expression using the current solution.

Return value:

Value of expression as an ndarray.

Example usage:

```
expr = A @ x + b
model.addConstr(expr == 0)
model.optimize()
val = expr.getValue()
```

MLinExpr.item()

```
item ( )
```

For an MLinExpr that contains a single element, returns a copy of that element as a LinExpr object. Calling this method on an MLinExpr with more than one element will raise a ValueError.

Return value:

An LinExpr object

Example usage:

```
mle = 2 * model.addMVar((2, 2)) + 1
mle_sub = mle[0, 1]  # A 0-D MLinExpr encapsulating one LinExpr object
mle_le = mle[0, 1].item()  # A copy of the resident LinExpr object
```

MLinExpr.ndim

```
ndim
```

The number of dimensions in this expression.

Return value:

An int

Example usage:

```
expr1 = 2 * model.addMVar((3,)) + 1
print(expr1.ndim)  #  "1"
expr2 =  2 * model.addMVar((1, 3)) + 1
print(expr2.ndim)  #  "2"
```

MLinExpr.shape

```
shape
```

The shape of this expression.

Return value:

A tuple of int

Example usage:

```
expr1 = 2 * model.addMVar((3,)) + 1
print(expr1.shape)  #  "(3,)"
expr2 = 2 * model.addMVar((1, 3)) + 1
print(expr2.shape)  #  "(1, 3)"
```

MLinExpr.size

```
size
```

The total number of elements in this expression.

Return value:

An int

Example usage:

```
expr1 = 2 * model.addMVar((3,)) + 1
print(expr1.size)  #  "3"
expr2 = 2 * model.addMVar((2, 3)) + 1
print(expr2.size)  #  "6"
```

MLinExpr.sum()

```
sum  ( axis=None )
```

Sum the elements of this MLinExpr; returns an MLinExpr object.

Arguments:

axis: An int, or None. Sum along the specified axis. If set to None, summation takes place along all axes of this MLinExpr.

Return value:

An MLinExpr representing the sum.

Example usage:

```
expr = 2 * model.addMVar((2, 2)) - 1
sum_row = expr.sum(axis=0)  # Sum along the rows
sum_col = expr.sum(axis=1)  # Sum along the columns
sum_all = expr.sum()  # Sum all elements, result is 0-D
```

MLinExpr.zeros()

```
zeros  ( shape )
```

Construct an all-zero MLinExpr object of the given shape.

Arguments:

`shape`: An int, or tuple of int. The requested shape.

Return value:

An MLinExpr, initialized to zero.

Example usage:

```
mle = gp.MLinExpr.zeros(3)
x = model.addMVar(3)
mle += 2 * x
```

MLinExpr.__eq__()

```
__eq__  ( )
```

Overloads the == operator, creating a TempConstr object that captures an array of equality constraints. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(A @ x == 1)
```

MLinExpr.__ge__()

```
__ge__  ( arg )
```

Overloads the >= operator, creating a TempConstr object that captures an array of inequality constraints. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(A @ x >= 1)
```

MLinExpr.__getitem__()

```
__getitem__  ( )
```

Index or slice this MLinExpr.

Return value:

An `MLinExpr` object.

Example usage:

```
mle = 2 * m.addMVar((2,2))
col0 = mle[:, 0]  # The first column of mle, 1-D result
elmt = mle[1, 0]  # The element at position (1, 0), 0-D result
```

You can index and slice MLinExpr objects like you would index NumPy's ndarray, and indexing behavior is straightforward to understand if you only *read* from the returned object. When you *write* to the returned object, be aware that some kinds of indexing return NumPy *views* on the

indexed expression (e.g., slices), while others result in copies being returned (e.g., fancy indexing). Here is an example:

Example usage:

```
mle = 2 * m.addMVar(4)
leading_part_1 = mle[:2]
leading_part_2 = mle[[0,1]]
leading_part_1 += 99  # This modifies mle, too
leading_part_2 += 1  # This doesn't modify mle
```

If you are unsure about any of these concepts and want to avoid any risk of accidentally writing back to the indexed object, you should always combine indexing with the `copy` method.

Example usage:

```
expr = 2 * model.addMVar((2,2)) + 1
first_col = expr[:, 0].copy()
first_col =+ 1  # Leaves expr untouched
```

MLinExpr.__le__()

`__le__ ( )`

Overloads the `<=` operator, creating a TempConstr object that captures an array of inequality constraints. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(A @ x <= 1)
```

MLinExpr.__setitem__()

`__setitem__ ( )`

Assign into this MLinExpr.

Example usage:

```
mle = 2 * model.addMVar((2,2))
v = model.addVar()
w = model.addVar()
mle[:] = v + 1  # Overwrite mle with four independent copies of 'v+1'
mle[1, 1] = w  # Overwrite the entry at position (1, 1) of mle with 'w'
```

Note that assignment into an existing MLinExpr always entails multiple data copies and is less efficient than building matrix expressions through operations with MVar objects.

7.16 MQuadExpr

Gurobi quadratic matrix expression object. Quadratic matrix expressions are used to build quadratic objective functions and quadratic constraints. They are temporary objects that typically have short lifespans.

You generally build quadratic matrix expressions using overloaded operators. For example, if `x` is an MVar object and `A` is a 2-D matrix (dense or sparse), then `x @ A @ x` and `x @ x` are both MQuadExpr objects. Arithmetic operations supported on `MQuadExpr` objects are addition, subtraction (e.g., `expr = x @ A @ x - y @ B @ y`), and multiplication by a constant (e.g. `expr = 2 * x @ A @ y`).

An `MQuadExpr` object has a `shape` representing its dimensions, a `size` that counts the total number of elements, and an `ndim` that gives the number of dimensions. These properties lean on their counterparts in NumPy's ndarray class.

When working with `MQuadExpr` objects, you need to make sure that the operands' shapes are compatible. The first step in building an `MQuadExpr` is often to build an `MLinExpr` (e.g., |expr = x @ A @ x|), so we refer you to the discussion of MLinExpr shape rules first. Rules for other operations on `MQuadExpr` objects are generally similar to those for `MLinExpr` objects, and both classes follow NumPy rules, so we refer you to the NumPy documentation for details.

The full list of overloaded operators on MQuadExpr objects is as follows: `+`, `+=`, `-`, `-=`, `*`, and `*=`. In Python parlance, we've defined the following `QuadExpr` functions: `__add__`, `__radd__`, `__iadd__`, `__sub__`, `__rsub__`, `__isub__`, `__neg__`, `__mul__`, `__rmul__`, and `__imul__`.

We've also overloaded the comparison operators (==, <=, and >=), to make it easier to build constraints from quadratic expressions.

MQuadExpr.clear()

```
clear ( )
```

Reset this expression to all zeros.

Example usage:

```
expr = 2 * model.addMVar(3)**2 + 1
expr.clear()  # All three entries are reset to constant 0.0
```

MQuadExpr.copy()

```
copy ( )
```

Create a copy of a quadratic matrix expression.

Return value:

Copy of expression object.

Example usage:

```
orig = x @ Q @ x + p @ x
copy = orig.copy()
copy += 2.0  # Leaves 'orig' untouched
```

MQuadExpr.getValue()

```
getValue ( )
```

Compute the value of a quadratic matrix expression using the current solution.

Return value:

Value of expression as an ndarary.

Example usage:

```
qexpr = x @ A @ x
model.addConstr(qexpr == 0)
model.optimize()
val = qexpr.getValue()
```

MQuadExpr.item()

```
item ( )
```

For an MQuadExpr that contains a single element, returns a copy of that element as a QuadExpr object. Calling this method on an MQuadExpr with more than one element will raise a ValueError.

Return value:

A QuadExpr object

Example usage:

```
mqe = 2 * model.addMVar((2, 2))** + 1
mqe_sub = mqe[0, 1]  # A 0-D MQuadExpr encapsulating one QuadExpr object
mqe_qe = mqe[0, 1].item()  # A copy of the resident QuadExpr object
```

MQuadExpr.ndim

```
ndim
```

The number of dimensions in this expression.

Return value:

An int

Example usage:

```
expr1 = 2 * model.addMVar((3,))**2 + 1
print(expr1.ndim)  #  "1"
expr2 =  2 * model.addMVar((1, 3))**2 + 1
print(expr2.ndim)  #  "2"
```

MQuadExpr.shape

```
shape
```

The shape of this expression.

Return value:

A tuple of int

Example usage:

```
expr1 = 2 * model.addMVar((3,))**2 + 1
print(expr1.shape)  #  "(3,)"
expr2 = 2 * model.addMVar((1, 3))**2 + 1
print(expr2.shape)  #  "(1, 3)"
```

MQuadExpr.size

`size`

The total number of elements in this expression.

Return value:

An int

Example usage:

```
expr1 = 2 * model.addMVar((3,))**2 + 1
print(expr1.size)  #  "3"
expr2 = 2 * model.addMVar((2, 3))**2 + 1
print(expr2.size)  #  "6"
```

MQuadExpr.sum()

`sum  ( axis=None )`

Sum the elements of this MQuadExpr; returns an MQuadExpr object.

Arguments:

`axis`: An int, or None. Sum along the specified axis. If set to None, summation takes place along all axes of this MQuadExpr.

Return value:

An MQuadExpr representing the sum.

Example usage:

```
expr = 2 * model.addMVar((2, 2))**2 - 1
sum_row = expr.sum(axis=0)  # Sum along the rows
sum_col = expr.sum(axis=1)  # Sum along the columns
sum_all = expr.sum()  # Sum all elements, result is 0-D
```

MQuadExpr.zeros()

`zeros  ( shape )`

Construct an all-zero MQuadExpr object of given shape.

Arguments:

`shape`: An int, or tuple of int. The requested shape.

Return value:

An MQuadExpr, initialized to zero.

Example usage:

```
mle = gp.MQuadExpr.zeros(3)
x = model.addMVar(3)
```

```
mle += 2 * x**2
```

MQuadExpr.__eq__()

```
__eq__  ( )
```

Overloads the == operator, creating a TempConstr object that captures an equality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x @ Q @ y == 1)
```

MQuadExpr.__ge__()

```
__ge__  ( arg )
```

Overloads the >= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x @ Q @ y >= 1)
```

MQuadExpr.__getitem__()

```
__getitem__  ( )
```

Index or slice this MQuadExpr.

Return value:

An `MQuadExpr` object.

Example usage:

```
mqe = 2 * m.addMVar((2,2))**2
col0 = mqe[:, 0]  # The first column of mqe, 1-D result
elmt = mqe[1, 0]  # The element at position (1, 0), 0-D result
```

You can index and slice MQuadExpr objects like you would index NumPy's ndarray, and indexing behavior is straightforward to understand if you only *read* from the returned object. When you *write* to the returned object, be aware some kinds of indexing return NumPy *views* on the indexed expression (e.g., slices), while others result in copies being returned (e.g., fancy indexing). Here is an example:

Example usage:

```
mqe = 2 * m.addMVar(4)**2
leading_part_1 = mqe[:2]
leading_part_2 = mqe[[0,1]]
leading_part_1 += 99  # This modifies mqe, too
leading_part_2 += 1  # This doesn't modify mqe
```

If you are unsure about any of these concepts and want to avoid any risk of accidentally writing back to the indexed object, you should always combine indexing with the `copy` method.

Example usage:

```
expr = 2 * model.addMVar((2,2))**2 + 1
first_col = expr[:, 0].copy()
first_col =+ 1  # Leaves expr untouched
```

MQuadExpr.__le__()

```
__le__  (  )
```

Overloads the `<=` operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x @ Q @ y <= 1)
```

MQuadExpr.__setitem__()

```
__setitem__  (  )
```

Assign into this MQuadExpr.

Example usage:

```
mqe = 2 * model.addMVar((2,2))**2
v = model.addVar()
w = model.addVar()
mqe[:] = v + 1  # Overwrite mqe with four independent copies of 'v+1'
mqe[1, 1] = w*w  # Overwrite the entry at position (1, 1) of mqe with 'w**2'
```

Note that assignment into an existing MQuadExpr always entails multiple data copies and is less efficient than building matrix expressions through operations with MVar objects.

7.17 TempConstr

Gurobi temporary constraint object. Objects of this class are created as intermediate results when building constraints using overloaded operators. There are no member functions on this class. Instead, `TempConstr` objects are created by a set of functions on Var, MVar, LinExpr, QuadExpr, MLinExpr, MQuadExpr, and GenExpr objects (e.g., ==, <=, and >=). You will generally never store objects of this class in your own variables.

The `TempConstr` object allows you to create several different types of constraints:

- **Linear Constraint:** an expression of the form `Expr1 sense Expr2`, where `Expr1` and `Expr2` are LinExpr objects, Var objects, or constants, and `sense` is one of `==`, `<=` or `>=`. For example, `x + y <= 1 + z` is a linear constraint, as is `x + y == 5`. Note that `Expr1` and `Expr2` can't both be constants.

- **Ranged Linear Constraint:** an expression of the form `LinExpr == [Const1, Const2]`, where `Const1` and `Const2` are constants and `LinExpr` is a LinExpr object. For example, `x + y == [1, 2]` is a ranged linear constraint.

- **Quadratic Constraint:** an expression of the form `Expr1 sense Expr2`, where `Expr1` and `Expr2` are QuadExpr objects, LinExpr objects, Var objects, or constants, and `sense` is one of `==`, `<=` or `>=`. For example, `x*x + y*y <= 3` is a quadratic constraint, as is `x*x + y*y <= z*z`. Note that one of `Expr1` or `Expr2` must be a QuadExpr (otherwise, the constraint would be linear).

- **Linear Matrix Constraint:** an expression of the form `Expr1 sense Expr2`, where one or both of `Expr1` and `Expr2` are MLinExpr objects and `sense` is one of `==`, `<=` or `>=`. For example, `A @ x <= 1` is a linear matrix constraint, as is `A @ x == B @ y`.

- **Quadratic Matrix Constraint:** an expression of the form `Expr1 sense Expr2`, where one or both of `Expr1` and `Expr2` are MQuadExpr objects and `sense` is one of `==`, `<=` or `>=`. For example, `x @ Q @ y <= 3` is a quadratic constraint, as is `x @ Q @ x <= y @ A @ y`.

- **Absolute Value Constraint:** an expression of the form `x == abs_(y)`, where `x` and `y` must be Var objects.

- **Logical Constraint:** an expression of the form `x == op_(y)`, where `x` is a binary Var object, and `y` is a binary Var, a list of binary Var, or a tupledict of binary Var, and `op_` is either `and_` or `or_` (or the Python-specific variants, `all_` and `any_`).

- **Min or Max Constraint:** an expression of the form `x == op_(y)`, where `x` is a Var object, and `y` is a Var, a list of Var and constants, or a tupledict of Var, and `op_` is one of `min_` or `max_`.

- **Indicator Constraint:** an expression of the form `(x == b) >> (Expr1 sense Expr2)`, where `x` is a binary Var object, `b` is either `0` or `1`; `Expr1` and `Expr2` are LinExpr objects, Var objects, or constants, and `sense` is one of `==`, `<=` or `>=`. Parenthesizing both expressions is required. For example, `(x == 1) >> (y + w <= 5)` is an indicator constraint, indicating that whenever the binary variable `x` takes the value `1` then the linear constraint `y + w <= 5` must hold.

Consider the following examples:

```
model.addConstr(x + y == 1);
model.addConstr(x + y == [1, 2]);
model.addConstr(x*x + y*y <= 1);
model.addConstr(A @ x <= 1);
model.addConstr(x @ A @ x <= 1);
model.addConstr(x == abs_(y));
model.addConstr(x == or_(y, z));
model.addConstr(x == max_(y, z));
model.addConstr((x == 1) >> (y + z <= 5));
```

In each case, the overloaded comparison operator creates an object of type `TempConstr`, which is then immediately passed to method Model.addConstr.

7.18 Column

Gurobi column object. A column consists of a list of coefficient, constraint pairs. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients. They are temporary objects that typically have short lifespans.

You generally build columns using the Column constructor. Terms can be added to an existing column using addTerms. Terms can also be removed from a column using remove.

Individual terms in a column can be queried using the getConstr, and getCoeff methods. You can query the number of terms in the column using the size method.

Column()

```
Column  ( coeffs=None, constrs=None )
```

Column constructor.

Arguments:

coeffs (optional): Lists the coefficients associated with the members of **constrs**.

constrs (optional): Constraint or constraints that participate in expression. If **constrs** is a list, then **coeffs** must contain a list of the same length. If **constrs** is a single constraint, then **coeffs** must be a scalar.

Return value:

An expression object.

Example usage:

```
constrs = model.getConstrs()

c = Column()
c.addTerms(3.0, constrs[0])
model.addVar(vtype=GRB.BINARY, obj=1.0, column=c)

model.addVar(vtype=GRB.INTEGER, column=Column(3.0, constrs[0]))

model.addVar(obj=3.0, column=Column([1.0, 2.0], constrs[1:3]))
```

Column.addTerms()

```
addTerms  ( coeffs, constrs )
```

Add new terms into a column.

Arguments:

coeffs: Coefficients for added constraints; either a list of coefficients or a single coefficient. The two arguments must have the same size.

constrs: Constraints to add to column; either a list of constraints or a single constraint. The two arguments must have the same size.

Example usage:

```
col.addTerms(1.0, x)
col.addTerms([2.0, 3.0], [y, z])
```

Column.clear()

```
clear ( )
```

Remove all terms from a column.

Example usage:

```
col.clear()
```

Column.copy()

```
copy ( )
```

Copy a column.

Return value:

Copy of input column.

Example usage:

```
col0 = Column(1.0, c0)
col1 = col0.copy()
```

Column.getCoeff()

```
getCoeff ( i )
```

Retrieve the coefficient from a single term in the column.

Return value:

Coefficient for the term at index `i` in the column.

Example usage:

```
col = Column([1.0, 2.0], [c0, c1])
print(col.getCoeff(1))
```

Column.getConstr()

```
getConstr ( i )
```

Retrieve the constraint object from a single term in the column.

Return value:

Constraint for the term at index `i` in the column.

Example usage:

```
col = Column([1.0, 2.0], [c0, c1])
print(col.getConstr(1))
```

Column.remove()

```
remove ( item )
```

Remove a term from a column.

Arguments:

`item`: If `item` is an integer, then the term stored at index `item` of the column is removed. If `item` is a Constr, then all terms that involve `item` are removed.

Example usage:

```
col = Column([1.0, 2.0], [c0, c1])
col.remove(c0)
```

Column.size()

```
size ( )
```

Retrieve the number of terms in the column.

Return value:

Number of terms in the column.

Example usage:

```
print(Column([1.0, 2.0], [c0, c1]).size())
```

7.19 Callbacks

Gurobi callback class. A callback is a user function that is called periodically by the Gurobi Optimizer in order to allow the user to query or modify the state of the optimization. More precisely, if you pass a function that takes two arguments (`model` and `where`) as the argument to Model.optimize or Model.computeIIS, your function will be called during the optimization. Your callback function can then call Model.cbGet to query the optimizer for details on the state of the optimization.

Gurobi callbacks can be used both to monitor the progress of the optimization and to modify the behavior of the Gurobi Optimizer. A simple user callback function might call Model.cbGet to produce a custom display, or perhaps to terminate optimization early (using Model.terminate) or to proceed to the next phase of the computation (using Model.cbProceed). More sophisticated MIP callbacks might use Model.cbGetNodeRel or Model.cbGetSolution to retrieve values from the solution to the current node, and then use Model.cbCut or Model.cbLazy to add a constraint to cut off that solution, or Model.cbSetSolution to import a heuristic solution built from that solution. For multi-objective problems, you might use Model.cbStopOneMultiObj to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

The Gurobi callback class provides a set of constants that are used within the user callback function. The first set of constants in this class list the options for the `where` argument to the user callback function. The `where` argument indicates from where in the optimization process the user callback is being called. Options are listed in the Callback Codes section of this document.

The other set of constants in this class list the options for the `what` argument to Model.cbGet. The `what` argument is used by the user callback to indicate what piece of status information it would like to retrieve. The full list of options can be found in the Callback Codes section. As with the `where` argument, you refer to a `what` constant through `GRB.Callback`. For example, the simplex objective value would be requested using `GRB.Callback.SPX_OBJVAL`.

If you would like to pass data to your callback function, you can do so through the Model object. For example, if your program includes the statement `model._value = 1` before the optimization begins, then your callback function can query the value of `model._value`. Note that the name of the user data field must begin with an underscore.

When solving a model using multiple threads, the user callback is only ever called from a single thread, so you don't need to worry about the thread-safety of your callback.

Note that changing parameters from within a callback is not supported, doing so may lead to undefined behavior.

You can look at `callback.py` in the examples directory for details of how to use Gurobi callbacks.

7.20 GurobiError

Gurobi exception object. Upon catching an exception `e`, you can examine `e.errno` (an integer) or `e.message` (a string). A list of possible values for `errno` can be found in the Error Code section. `message` provides additional information on the source of the error.

7.21 Env

Gurobi environment object. Note that environments play a much smaller role in the Python interface than they do in other Gurobi language APIs, mainly because the Python interface has a default environment. Unless you explicitly pass your own environment to routines that require an environment, the default environment will be used.

The primary situations where you will want to use your own environment are:

- When you are using a Gurobi Compute Server and want to choose the server from within your program.
- When you need control over garbage collection of your environment. The Gurobi Python interface maintains a reference to the default environment, so by default it will never be garbage collected. By creating your own environment, you can control exactly when your program releases any licensing tokens or Compute Servers it is using.
- When you are using concurrent environments in one of the concurrent optimizers.

It is good practice to use the `with` keyword when dealing with environment (and model) objects. That way the resources tied to these objects are properly released even if an exception is raised at some point. The following example illustrates two typical use patterns.

Example usage:

```
import gurobipy as gp
with gp.Env("gurobi.log") as env, gp.Model(env=env) as model:
    # Populate model object here...
    model.optimize()

with gp.Env(empty=True) as env:
    env.setParam("ComputeServer", "myserver1:32123")
    env.setParam("ServerPassword", "pass")
    env.start()
    with gp.Model(env=env) as model:
        # Populate model object here...
        model.optimize()
```

Note that you can manually remove the reference to the default environment by calling disposeDefaultEnv. After calling this, and after all models built within the default environment are garbage collected, the default environment will be garbage collected as well. A new default environment will be created automatically if you call a routine that needs one.

Env()

```
Env  ( logfilename="", empty=False, params=None )
```

Env constructor. You will generally want to use the default environment in Gurobi Python programs. The exception is when you want explicit control over environment garbage collection. By creating your own environment object and always passing it to methods that take an environment as input (read or the Model constructor), you will avoid creating the default environment. Once

every model created using an Env object is garbage collected, and once the Env object itself is no longer referenced, the garbage collector will reclaim the environment and release all associated resources.

If the environment is not empty, This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`logfilename`: Name of the log file for this environment. Pass an empty string if you don't want a log file.

`empty`: Indicates whether the environment should be empty. You should use `empty=True` if you want to set parameters before actually starting the environment. This can be useful if you want to connect to a Compute Server, a Token Server, the Gurobi Instant Cloud, a Cluster Manager or use a WLS license. See the Environment Section for more details.

`params`: A dict containing Gurobi parameter/value pairs that should be set already upon environment creation. Any server related parameters can be set through this dict, too.

Return value:

New environment object.

Example usage:

```
env = Env("gurobi.log")
model = read("misc07.mps", env)
model.optimize()
```

Example usage:

```
p = {"ComputeServer": "localhost:33322",
     "ServerPassword": "pass",
     "TimeLimit": 120.0}
with Env(params=p) as env, read('misc07.mps', env=env) as model:
    model.optimize()
```

Env.close()

```
close ( )
```

Free all resources associated with this `Env` object. This method is a synonym for dispose. Users should close all models created in this environment before closing this `Env` object. After this method is called, this `Env` object must no longer be used.

Example usage:

```
env = Env()
```

```
model = read("misc07.mps", env)
model.optimize()
model.close()
env.close()
```

Env.dispose()

```
dispose  ( )
```

Free all resources associated with this Env object. This method is a synonym for close.

Users should dispose of all models created in this environment before disposing of this Env object.

After this method is called, this Env object must no longer be used.

Example usage:

```
env = Env()
model = read("misc07.mps", env)
model.optimize()
model.dispose()
env.dispose()
```

Env.getParam()

```
getParam  ( paramname )
```

Get the current value of a given parameter.

Arguments:

paramname: String containing the name of parameter that you would like to query. Note that case is ignored.

Return value:

Current value of the parameter in this environment

Env.resetParams()

```
resetParams  ( )
```

Reset the values of all parameters to their default values.

Example usage:

```
env.resetParams()
```

Env.setParam()

```
setParam  ( paramname, newvalue )
```

Set the value of a parameter to a new value.

Arguments:

paramname: String containing the name of parameter that you would like to modify. The name can include '*' and '?' wildcards. If more than one parameter matches, a GurobiEr-

ror is raised with the matching names listed, and no parameters are modified. Note that case is ignored.

newvalue: Desired new value for parameter. Can be 'default', which indicates that the parameter should be reset to its default value.

Example usage:

```
env.setParam("Cuts", 2)
env.setParam("Heu*", 0.5)
env.setParam("*Interval", 10)
```

Env.start()

```
start  ( )
```

Start an empty environment. If the environment has already been started, this method will do nothing. If the call fails, the environment will have the same state as it had before the call to this method.

This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter). After that, it will apply all parameter changes specified by the user prior to this call. Note that this might overwrite parameters set in the license file, or in the `gurobi.env` file, if present.

After all these changes are performed, the code will actually activate the environment, and make it ready to work with models.

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Example usage:

```
env = Env(empty=True)
env.setParam('ComputeServer', 'server.mydomain.com:61000')
env.setParam('ServerPassword', 'mypassword')
env.start()
```

Env.writeParams()

```
writeParams  ( filename )
```

Write all modified parameters to a file. The file is written in PRM format.

Example usage:

```
env.setParam("Heu*", 0.5)
env.writeParams("params.prm")  # file will contain changed parameter
system("cat params.prm")
```

7.22 Batch

Gurobi batch object. Batch optimization is a feature available with the Gurobi Cluster Manager. It allows a client program to build an optimization model, submit it to a Compute Server cluster (through a Cluster Manager), and later check on the status of the model and retrieve its solution. For more information, please refer to the Batch Optimization section.

Commonly used methods on batch objects include update (refresh attributes from the Cluster Manager), abort (abort execution of a batch request), retry (retry optimization for an interrupted or failed batch), discard (remove the batch request and all related information from the Cluster Manager), and getJSONSolution (query solution information for the batch request).

These methods are built on top of calls to the Cluster Manager REST API. They are meant to simplify such calls, but note that you always have the option of calling the REST API directly.

Batch objects have four attributes:

- BatchID: Unique ID for the batch request.
- BatchStatus: Last batch status.
- BatchErrorCode: Last error code.
- BatchErrorMessage: Last error message.

You can access their values as you would for other attributes: `batch.BatchStatus`, `batch.BatchID`, etc. Note that all Batch attributes are locally cached, and are only updated when you create a client-side batch object or when you explicitly update this cache, which can done by calling update.

Batch()

```
Batch  ( batchID, env )
```

Given a `BatchID`, as returned by optimizeBatch, and a Gurobi environment that can connect to the appropriate Cluster Manager (i.e., one where parameters CSManager, UserName, and ServerPassword have been set appropriately), this function returns a Batch object. With it, you can query the current status of the associated batch request and, once the batch request has been processed, you can query its solution. Please refer to the Batch Optimization section for details and examples.

Arguments:

`batchID`: ID of the batch request for which you want to access status and other information.
`env`: The environment in which the new batch object should be created.

Return value:

New batch object.

Example usage:

```
batch = gp.Batch(batchID, env)

# Automatically disposed with context manager
with gp.Batch(batchID, env) as batch:
  pass
```

Batch.abort()

```
abort ( )
```

This method instructs the Cluster Manager to abort the processing of this batch request, changing its status to ABORTED. Please refer to the Batch Status Codes section for further details.

Example usage:

```
starttime = time.time()
while batch.BatchStatus == GRB.BATCH_SUBMITTED:
    # Abort this batch if it is taking too long
    curtime = time.time()
    if curtime - starttime > maxwaittime:
        batch.abort()
        break
```

Batch.discard()

```
discard ( )
```

This method instructs the Cluster Manager to remove all information related to the batch request in question, including the stored solution if available. Further queries for the associated batch request will fail with error code GRB_ERROR_DATA_NOT_AVAILABLE. Use this function with care, as the removed information can not be recovered later on.

Example usage:

```
# Remove batch request from manager
batch.discard()
```

Batch.dispose()

```
dispose ( )
```

Free all resources associated with this Batch object. After this method is called, this Batch object must no longer be used.

Example usage:

```
batch.dispose()
```

Batch.getJSONSolution()

```
getJSONSolution ( )
```

This method retrieves the solution of a completed batch request from a Cluster Manager. The solution is returned as a JSON solution string. For this call to succeed, the status of the batch request must be COMPLETED. Please refer to the Batch Status Codes section for further details.

Example usage:

```
print("JSON solution:")
# Get JSON solution as string, create dict from it
sol = json.loads(batch.getJSONSolution())
```

Batch.retry()

```
retry ( )
```

This method instructs the Cluster Manager to retry optimization of a failed or aborted batch request, changing its status to `SUBMITTED`. Please refer to the Batch Status Codes section for further details.

Example usage:

```
starttime = time.time()
while batch.BatchStatus == GRB.BATCH_SUBMITTED:
    # Abort this batch if it is taking too long
    curtime = time.time()
    if curtime - starttime > maxwaittime:
        batch.abort()
        break

    # Wait for two seconds
    time.sleep(2)

    # Update the resident attribute cache of the Batch object with the
    # latest values from the cluster manager.
    batch.update()

    # If the batch failed, we retry it
    if batch.BatchStatus == GRB.BATCH_FAILED:
        batch.retry()
```

Batch.update()

```
update ( )
```

All Batch attribute values are cached locally, so queries return the value received during the last communication with the Cluster Manager. This method refreshes the values of all attributes with the values currently available in the Cluster Manager (which involves network communication).

Example usage:

```
# Update the resident attribute cache of the Batch object with the
# latest values from the cluster manager.
batch.update()
```

Batch.writeJSONSolution()

```
writeJSONSolution ( filename )
```

This method returns the stored solution of a completed batch request from a Cluster Manager. The solution is returned in a gzip-compressed JSON file. The file name you provide must end with a `.json.gz` extension. The JSON format is described in the JSON solution format section. Note that for this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details.

Arguments:

`filename`: Name of file where the solution should be stored (in JSON format).

Example usage:

```
# Write the full JSON solution string to a file
batch.writeJSONSolution("batch-sol.json.gz")
```

7.23 GRB

Class for Python constants. Classes GRB.Attr and GRB.Param are used to get or set Gurobi attributes and parameters, respectively.

Constants

The following list contains a set of constants that are used by the Gurobi Python interface. You would refer to them using a `GRB.` prefix (e.g., `GRB.OPTIMAL`).

```
# Status codes

  LOADED          = 1
  OPTIMAL         = 2
  INFEASIBLE      = 3
  INF_OR_UNBD     = 4
  UNBOUNDED       = 5
  CUTOFF          = 6
  ITERATION_LIMIT = 7
  NODE_LIMIT      = 8
  TIME_LIMIT      = 9
  SOLUTION_LIMIT  = 10
  INTERRUPTED     = 11
  NUMERIC         = 12
  SUBOPTIMAL      = 13
  INPROGRESS      = 14
  USER_OBJ_LIMIT  = 15
  WORK_LIMIT      = 16
  MEM_LIMIT       = 17

# Batch status codes

  BATCH_CREATED   = 1
  BATCH_SUBMITTED = 2
  BATCH_ABORTED   = 3
  BATCH_FAILED    = 4
  BATCH_COMPLETED = 5

# Constraint senses

  LESS_EQUAL    = '<'
  GREATER_EQUAL = '>'
  EQUAL         = '='

# Variable types

  CONTINUOUS = 'C'
  BINARY     = 'B'
  INTEGER    = 'I'
  SEMICONT   = 'S'
  SEMIINT    = 'N'

# Objective sense

  MINIMIZE = 1
  MAXIMIZE = -1
```

```
# SOS types

  SOS_TYPE1 = 1
  SOS_TYPE2 = 2

# General constraint types

  GENCONSTR_MAX       = 0
  GENCONSTR_MIN       = 1
  GENCONSTR_ABS       = 2
  GENCONSTR_AND       = 3
  GENCONSTR_OR        = 4
  GENCONSTR_NORM      = 5
  GENCONSTR_INDICATOR = 6
  GENCONSTR_PWL       = 7
  GENCONSTR_POLY      = 8
  GENCONSTR_EXP       = 9
  GENCONSTR_EXPA      = 10
  GENCONSTR_LOG       = 11
  GENCONSTR_LOGA      = 12
  GENCONSTR_POW       = 13
  GENCONSTR_SIN       = 14
  GENCONSTR_COS       = 15
  GENCONSTR_TAN       = 16
  GENCONSTR_LOGISTIC  = 17

# Basis status

  BASIC          = 0
  NONBASIC_LOWER = -1
  NONBASIC_UPPER = -2
  SUPERBASIC     = -3

# Numeric constants

  INFINITY  = 1e100
  UNDEFINED = 1e101
  MAXINT    = 2000000000

# Limits

  MAX_NAMELEN    = 255
  MAX_STRLEN     = 512
  MAX_TAGLEN     = 10240
  MAX_CONCURRENT = 64

# Other constants

  DEFAULT_CS_PORT = 61000

# Version number

  VERSION_MAJOR     = 11
  VERSION_MINOR     = 0
  VERSION_TECHNICAL = 0
```

```
# Errors

  ERROR_OUT_OF_MEMORY            = 10001
  ERROR_NULL_ARGUMENT            = 10002
  ERROR_INVALID_ARGUMENT         = 10003
  ERROR_UNKNOWN_ATTRIBUTE        = 10004
  ERROR_DATA_NOT_AVAILABLE       = 10005
  ERROR_INDEX_OUT_OF_RANGE       = 10006
  ERROR_UNKNOWN_PARAMETER        = 10007
  ERROR_VALUE_OUT_OF_RANGE       = 10008
  ERROR_NO_LICENSE               = 10009
  ERROR_SIZE_LIMIT_EXCEEDED      = 10010
  ERROR_CALLBACK                 = 10011
  ERROR_FILE_READ                = 10012
  ERROR_FILE_WRITE               = 10013
  ERROR_NUMERIC                  = 10014
  ERROR_IIS_NOT_INFEASIBLE       = 10015
  ERROR_NOT_FOR_MIP              = 10016
  ERROR_OPTIMIZATION_IN_PROGRESS = 10017
  ERROR_DUPLICATES               = 10018
  ERROR_NODEFILE                 = 10019
  ERROR_Q_NOT_PSD                = 10020
  ERROR_QCP_EQUALITY_CONSTRAINT  = 10021
  ERROR_NETWORK                  = 10022
  ERROR_JOB_REJECTED             = 10023
  ERROR_NOT_SUPPORTED            = 10024
  ERROR_EXCEED_2B_NONZEROS       = 10025
  ERROR_INVALID_PIECEWISE_OBJ    = 10026
  ERROR_UPDATEMODE_CHANGE        = 10027
  ERROR_CLOUD                    = 10028
  ERROR_MODEL_MODIFICATION       = 10029
  ERROR_CSWORKER                 = 10030
  ERROR_TUNE_MODEL_TYPES         = 10031
```

GRB.Attr

The constants defined in this class are used to get or set attributes (through Model.getAttr or Model.setAttr, for example). Please refer to the Attributes section to see a list of all attributes and their functions. You refer to an attribute using a `GRB.Attr` prefix (e.g., `GRB.Attr.Obj`). Note that these constants are simply strings, so wherever you might use this constant, you also have the option of using the string directly (e.g, `'obj'` rather than `GRB.Attr.Obj`).

GRB.Param

The constants defined in this class are used to get or set parameters Model.getParamInfo or Model.setParam. Please refer to the Parameters section to see a list of all parameters and their functions. You refer to a parameter using a `GRB.Param` prefix (e.g., `GRB.Param.displayInterval`). Note that these constants are simply strings, so wherever you might use this constant, you also have the option of using the string directly (e.g, `'displayInterval'` rather than `GRB.Param.displayInterval`).

7.24 tuplelist

Gurobi tuple list. This is a sub-class of the Python `list` class that is designed to efficiently support a usage pattern that is quite common when building optimization models. In particular, if a `tuplelist` is populated with a list of tuples, the select function on this class efficiently selects tuples whose values match specified values in specified tuple fields. To give an example, the statement `l.select(1, '*', 5)` would select all member tuples whose first field is equal to '1' and whose third field is equal to '5'. The `'*'` character is used as a wildcard to indicate that any value is acceptable in that field.

You generally build `tuplelist` objects in the same way you would build standard Python lists. For example, you can use the `+=` operator to append a new list of items to an existing `tuplelist`, or the `+` operator to concatenate a pair of `tuplelist` objects. You can also call the `append`, `extend`, `insert`, `pop`, and `remove` functions.

To access the members of a `tuplelist`, you also use standard list functions. For example, `l[0]` returns the first member of a `tuplelist`, while `l[0:10]` returns a `tuplelist` containing the first ten members. You can also use `len(l)` to query the length of a list.

Note that `tuplelist` objects build and maintain a set of internal data structures to support efficient `select` operations. If you wish to reclaim the storage associated with these data structures, you can call the clean function.

A `tuplelist` is designed to store tuples containing scalar values (`int`, `float`, `string`, ...). It may produce unpredictable results with other Python objects, such as tuples of tuples. Thus, you can store `(1, 2.0, 'abc')` in a `tuplelist`, but you shouldn't store `((1, 2.0), 'abc')`.

tuplelist()

```
tuplelist  ( list )
```

tuplelist constructor.

Arguments:

`list`: Initial list of member tuples.

Return value:

A tuplelist object.

Example usage:

```
l = tuplelist([(1,2), (1,3), (2,4)])
l = tuplelist([('A', 'B', 'C'), ('A', 'C', 'D')])
```

tuplelist.select()

```
select  ( pattern )
```

Returns a `tuplelist` containing all member tuples that match the specified pattern. The pattern requires one argument for each field in the member tuple. A scalar argument must match the corresponding field exactly. A list argument matches if any list member matches the corresponding field. A `'*'` argument matches any value in the corresponding field.

Arguments:

`pattern`: Pattern to match for a member tuple.

Example usage:

```
l.select(1, 3, '*', 6)
l.select([1, 2], 3, '*', 6)
l.select('A', '*', 'C')
```

tuplelist.clean()

```
clean  ( )
```

Discards internal data structure associated with a `tuplelist` object. Note that calling this routine won't affect the contents of the `tuplelist`. It only affects the memory used and the performance of later calls to select.

Example usage:

```
l.clean()
```

tuplelist.__contains__()

```
__contains__  ( val )
```

Provides efficient support for the Python `in` operator.

Example usage:

```
if (1,2) in l:
  print("Tuple (1,2) is in tuplelist l")
```

7.25 tupledict

Gurobi tuple dict. This is a sub-class of the Python `dict` class that is designed to efficiently support a usage pattern that is quite common when building optimization models. In particular, a `tupledict` is a Python `dict` where the keys represent variable indices, and the values are typically Gurobi Var objects. Objects of this class make it easier to build linear expressions on sets of Gurobi variables, using tuplelist.select() syntax and semantics.

You typically build a `tupledict` by calling Model.addVars. Once you've created a `tupledict` `d`, you can use `d.sum()` to create a linear expression that captures the sum of the variables in the `tupledict`. You can also use a command like `d.sum(1, '*', 5)` to create a sum over a subset of the variables in `d`. Assuming the keys for the `tupledict` are tuples containing three fields, this statement would create a linear expression that captures the sum over all variables in `d` whose keys contain a 1 in the first field of the tuple and a 5 in the third field (the `'*'` character is a wildcard that indicates that any value is acceptable in that field). You can also use `d.prod(coeff)` to create a linear expression where the coefficients are pulled from the dictionary `coeff`. For example, if `d(1,2,5)` contains variable `x` and `coeff(1,2,5)` is 2.0, then the resulting expression would include term $2.0 * x$.

To access the members of a `tupledict`, you can use standard dict indexing. For example, `d[1,2]` returns the value associated with tuple `(1,2)`.

Note that a `tupledict` key must be a tuple of scalar values (`int`, `float`, `string`, ...). Thus, you can use `(1, 2.0, 'abc')` as a key, but you can't use `((1, 2.0), 'abc')`.

Note that `tupledict` objects build and maintain a set of internal data structures to support efficient `select` operations. If you wish to reclaim the storage associated with these data structures, you can call the clean function.

tupledict()

```
tupledict  ( args, kwargs )
```

`tupledict` constructor. Arguments are identical to those of a Python `dict` constructor.

Note that you will typically use Model.addVars to build a `tupledict`.

Arguments:

`args`: Positional arguments.

`kwargs`: Named arguments.

Return value:

A `tupledict` object.

Example usage:

```
d = tupledict([((1,2), 'onetwo'), ((1,3), 'onethree'), ((2,3), 'twothree')])
print(d[1,2]) # prints 'onetwo'
```

tupledict.select()

```
select  ( pattern )
```

Returns a `list` containing the values associated with keys that match the specified tuple pattern. The pattern should provide one value for each field in the key tuple. A `'*'` value indicates that any value is accepted in that field.

Without arguments, this method returns a list of all values in the `tupledict`.

Arguments:

`pattern`: Pattern to match for a key tuple.

Example usage:

```
d = tupledict([((1,2), 'onetwo'), ((1,3), 'onethree'), ((2,3), 'twothree')])
print(d.select())       # prints ['onetwo', 'onethree', 'twothree']
print(d.select(1, '*')) # prints ['onetwo', 'onethree']
print(d.select('*', 3)) # prints ['onethree', 'twothree']
print(d.select(1, 3))   # prints ['onethree']
```

tupledict.sum()

```
sum  ( pattern )
```

Returns the sum of the values associated with keys that match the specified pattern. If the values are Gurobi `Var` objects, the result is a `LinExpr`. The pattern should provide one value for each field in the key tuple. A `'*'` value indicates that any value is accepted in that field.

Without arguments, this method returns the sum of all values in the `tupledict`.

Arguments:

`pattern`: Pattern to match for a key tuple.

Example usage:

```
x = m.addVars([(1,2), (1,3), (2,3)])
expr = x.sum()       # LinExpr: x[1,2] + x[1,3] + x[2,3]
expr = x.sum(1, '*') # LinExpr: x[1,2] + x[1,3]
expr = x.sum('*', 3) # LinExpr: x[1,3] + x[2,3]
expr = x.sum(1, 3)   # LinExpr: x[1,3]
```

tupledict.prod()

```
prod  ( coeff, pattern )
```

Returns a linear expression that contains one term for each tuple that is present in both the `tupledict` and the `coeff` argument; `coeff` should be a Python `dict` object that maps tuples to coefficient values. For example, `x.prod(coeff)` would contain term `2.0*var` if `x[1,2] = var` and `coeff[1,2] = 2.0`.

Arguments:

`coeff`: Python `dict` that maps tuples to coefficients.

`pattern`: Pattern to match for a key tuple.

Example usage:

```
x = m.addVars([(1,2), (1,3), (2,3)])
coeff = dict([((1,2), 2.0), ((1,3), 2.1), ((2,3), 3.3)])
expr = x.prod(coeff) # LinExpr: 2.0 x[1,2] + 2.1 x[1,3] + 3.3 x[2,3]
expr = x.prod(coeff, '*', 3) # LinExpr: 2.1 x[1,3] + 3.3 x[2,3]
```

tupledict.clean()

```
clean ( )
```

Discards internal data structure associated with a `tupledict` object. Note that calling this routine won't affect the contents of the `tupledict`. It only affects the memory used and the performance of later calls to select.

Example usage:

```
d.clean()
```

7.26 General Constraint Helper Functions

Gurobi general constraint helper functions - used in conjunction with overloaded operators and Model.addConstr or Model.addConstrs to build general constraints.

abs_()

```
abs_  ( argvar )
```

Used to set a decision variable equal to the absolute value of another decision variable.

Arguments:

argvar (Var): The variable whose absolute value will be taken.

Example usage:

```
m.addConstr(y == abs_(x))
```

Return value:

Returns a GenExpr object.

and_()

```
and_  ( *args )
```

Used to set a binary decision variable equal to the logical AND of a list of other binary decision variables.

Note that the Gurobi Python interface includes an equivalent `all_()` function.

Arguments:

***args (Var, or list of Var, or tupledict of Var values)**: The variables over which the AND concatenation will be taken.

Example usage:

```
m.addConstr(z == and_(x, y))
m.addConstr(z == and_([x, y]))
```

Return value:

Returns a GenExpr object.

max_()

```
max_  ( *args, constant=None )
```

Used to set a decision variable equal to the maximum of a list of decision variables and, if desired, a constant.

Arguments:

***args (Var, or list of Var, or tupledict of Var values)**: The variables over which the MAX will be taken.

constant (float, optional): The constant value to include among the arguments of the MAX operation.

Example usage:

```
m.addConstr(z == max_(x, y, constant=3))
m.addConstr(z == max_([x, y], constant=3))
```

Return value:

Returns a GenExpr object.

min_()

```
min_  ( *args, constant=None )
```

Used to set a decision variable equal to the minimum of a list of decision variables and, if desired, a constant.

Arguments:

`*args (Var, or list of Var, or tupledict of Var values)`: The variables over which the MIN will be taken.

`constant (float, optional)`: The constant value to include among the arguments of the MIN operation.

Example usage:

```
m.addConstr(z == min_(x, y, constant=3))
m.addConstr(z == min_([x, y], constant=3))
```

Return value:

Returns a GenExpr object.

or_()

```
or_  ( *args )
```

Used to set a binary decision variable equal to the logical OR of a list of other binary decision variables.

Note that the Gurobi Python interface includes an equivalent `any_()` function.

Arguments:

`*args (Var, or list of Var, or tupledict of Var values)`: The variables over which the OR concatenation will be taken.

Example usage:

```
m.addConstr(z == or_(x, y))
m.addConstr(z == or_([x, y]))
```

Return value:

Returns a GenExpr object.

norm()

```
norm  ( vars, which )
```

Used to set a decision variable equal to the norm of other decision variables.

Arguments:

vars (list of Var, or tupledict of Var values, or 1-dim MVar): The variables over which the NORM will be taken. Note that this may not contain duplicates.

which (float): Which norm to use. Options are 0, 1, 2, and any value greater than or equal to GRB.INFINITY.

Example usage:

```
x = m.addVars(3)
nx = m.addVar()
m.addConstr(nx == norm(x, 1.0))
```

Return value:

Returns a GenExpr object.

7.27 Matrix-Friendly API Functions

Implements a small set of functions equivalent to numpy that operate on the array-like matrix-friendly API classes MVar, MLinExpr, MQuadExpr, MConstr, MQConstr, and MGenConstr.

hstack()

```
hstack ( tup )
```

Stack arrays in sequence horizontally (column wise). This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis.

This method has the same behavior as numpy.hstack, except that it always returns a gurobipy matrix-friendly API object of the appropriate type.

Arguments:

`tup`: A sequence of matrix-friendly API objects

Return value:

Returns a matrix-friendly API object. Note that the return type depends on the types of the input objects.

Example usage:

```
X = model.addMVar((k, n))
Y = model.addMVar((k, m))
XY = gp.hstack((X, Y))     # (k, n+m) MVar
```

vstack()

```
vstack ( tup )
```

Stack arrays in sequence vertically (row wise). This is equivalent to concatenation along the first axis after 1-D arrays of shape `(N,)` have been reshaped to `(1,N)`.

This method has the same behavior as numpy.vstack, except that it always returns a gurobipy matrix-friendly API object of the appropriate type.

Arguments:

`tup`: A sequence of matrix-friendly API objects

Return value:

Returns a matrix-friendly API object. Note that the return type depends on the types of the input objects.

Example usage:

```
X = model.addMVar((n, k))
Y = model.addMVar((m, k))
XY = gp.vstack((X, Y))     # (n+m, k) MVar
```

concatenate()

```
concatenate ( tup, axis )
```

Join a sequence of arrays along an existing axis.

This method has the same behavior as numpy.concatenate, except that it always returns a gurobipy matrix-friendly API object of the appropriate type.

Arguments:

tup: A sequence of matrix-friendly API objects

axis: The axis along which the arrays will be joined. If axis is None, arrays are flattened before use. Default is 0.

Return value:

Returns a matrix-friendly API object. Note that the return type depends on the types of the input objects.

Example usage:

```
X = model.addMVar((k, n))
Y = model.addMVar((k, m))
XY = gp.concatenate((X, Y), axis=1)  # (k, n+m) MVar
```

MATLAB API Overview

This section documents the Gurobi MATLAB® interface. For those of you who are not familiar with MATLAB, it is an environment for doing numerical computing. Please visit the MATLAB web site for more information. This manual begins with a quick overview of the methods provided by our MATLAB API. It then continues with a `comprehensive presentation` of all of the available methods, their arguments, and their return values.

If you are new to the Gurobi Optimizer, we suggest that you start with the Getting Started Knowledge Base article or the Example Tour of the reference manual. These documents provide concrete examples of how to use the methods described here.

The MATLAB Optimization Toolbox provides its own interface for building optimization models (starting with version 2017b). Gurobi supports this interface as well. We'll discuss this aspect in the problem-based modeling section; consult also the `linprog`, `intlinprog`, `opttoolbox_lp`, and `opttoolbox_mip1` examples in the Gurobi distribution for illustrations of how to pass models built using this interface to Gurobi.

A quick note for new users: the convention in math programming is that variables are non-negative unless specified otherwise. You'll need to explicitly set lower bounds if you want variables to be able to take negative values.

Models

Our Gurobi MATLAB interface enables you to express problems of the following form:

minimize	$x^T Q x + c^T x + \text{alpha}$	
subject to	$Ax = b$	(linear constraints)
	$\ell \leq x \leq u$	(bound constraints)
	some x_j integral	(integrality constraints)
	$x^T Q c\, x + q^T x \leq \text{beta}$	(quadratic constraints)
	some x_i in SOS	(special ordered set constraints)
	min, max, abs, or, ...	(general constraints)

Models are stored as `struct` variables, each consisting of multiple *fields*. The fields capture the different model components listed above. Many of these model components are optional. For example, integrality constraints may be omitted.

An optimization model may be loaded from a file (using the `gurobi_read` function), or it can be built by populating the appropriate fields of a model variable (using standard MATLAB constructs). We will discuss the details of how models are represented in the `model` argument section.

We often refer to the *class* of an optimization model. At the highest level, a model can be continuous or discrete, depending on whether the modeling elements present in the model require discrete decisions to be made. Among continuous models...

- A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*.
- If the objective is quadratic, the model is a *Quadratic Program (QP)*.
- If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We sometimes refer to a few special cases of QCP: QCPs with convex constraints, QCPs with non-convex constraints, *bilinear programs*, and *Second-Order Cone Programs (SOCP)*.
- If any of the constraints are non-linear (chosen from among the available general constraints), the model is a *Non-Linear Program (NLP)*.

A model that contains any integer variables, semi-continuous variables, semi-integer variables, Special Ordered Set (SOS) constraints, or general constraints, is discrete, and is referred to as a *Mixed Integer Program (MIP)*. The special cases of MIP, which are the discrete versions of the continuous models types we've already described, are...

- *Mixed Integer Linear Programs (MILP)*
- *Mixed Integer Quadratic Programs (MIQP)*
- *Mixed Integer Quadratically-Constrained Programs (MIQCP)*
- *Mixed Integer Second-Order Cone Programs (MISOCP)*
- *Mixed Integer Non-Linear Programs (MINLP)*

The Gurobi Optimizer handles all of these model classes. Note that the boundaries between them aren't as clear as one might like, because we are often able to transform a model from one class to a simpler class.

Solving a Model

Once you have built a model, you can call `gurobi` to compute a solution. By default, `gurobi` will use the concurrent optimizer to solve LP models, the barrier algorithm to solve QP models and QCP models with convex constraints, and the branch-and-cut algorithm to solve mixed integer models. The solution is returned as a `struct` variable. We will discuss the details of how optimization results are represented when we discuss the `gurobi` function.
Here is a simple example of a likely sequence of commands in the MATLAB API:

```
model = gurobi_read('examples/data/stein9.mps');
result = gurobi(model);
```

Multiple Solutions and Multiple Objectives

By default, the Gurobi Optimizer assumes that your goal is to find one proven optimal solution to a model with a single objective function. Gurobi provides features that allow you to relax either of these assumptions. You should refer to the section on Solution Pools for information on how to request more than one solution, or the section on Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call `gurobi_iis` to compute an Irreducible Inconsistent Subsystem (IIS). This method can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. We will discuss the details of how IIS results are represented in the `gurobi_iis` function documentation.

To attempt to repair an infeasibility, call `gurobi_feasrelax` to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Managing Parameters

The Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization.

Each Gurobi parameter has a default value. Desired parameter changes are passed in a `struct` variable. The name of each field within this struct must be the name of a Gurobi parameter, and the associated value should be the desired value of that parameter. You can find a complete list of the available Gurobi parameters in the reference manual. We will provide additional details on changing parameter settings in the `params` argument section.

Monitoring Progress

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging behavior. If you would like to direct output to a file as well as to the screen, set the `LogFile` parameter to the name of your desired log file. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter. A detailed description of the Gurobi log file can be found in the Logging section of the reference manual.

Error Handling

If unsuccessful, the methods of the Gurobi MATLAB interface will display an error code and an error message. A list of possible error codes can be found in the Error Code section in the reference manual.

Environments and license parameters

By default, the various Gurobi functions will look for a valid license file and create a local Gurobi environment. This environment exists for as long as the corresponding MATLAB API function is running, and is released upon completion.

Another option is to provide licensing parameters through an optional `params` argument (also through a `struct`). This argument allows you to solve the given problem on a Gurobi Compute Server, on Gurobi Instant Cloud, or using a Gurobi Cluster Manager. We will discuss this topic further in the `params` argument section.

Gurobi will check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

8.1 Common Arguments

Most common arguments in the Gurobi MATLAB interface are MATLAB `struct` variables, each containing multiple `fields`. Several of these fields are optional. Note that you refer to a field of a struct variable by adding a period to the end of the variable name, followed by the name of the field. For example, `model.A` refers to field `A` of variable `model`.

The model argument

Model variables store optimization problems (as described in the `problem` statement).

Models can be built in a number of ways. You can populate the appropriate fields of the `model` struct using standard MATLAB routines. You can also read a model from a file, using `gurobi_read`. A few API functions (`gurobi_feasrelax` and `gurobi_relax`) also return models.

Note that all vector fields within the `model` variable must be dense vectors except for the linear part of the quadratic constraints and INDICATOR general constraints, all matrix fields must be sparse matrices, and all strings, names, etc. must be `char` arrays.

The following is an enumeration of all of the fields of the `model` argument that Gurobi will take into account when optimizing the model:

Commonly used fields

`A`: The linear constraint matrix.

`obj (optional)`: The linear objective vector (the `c` vector in the `problem` statement). When present, you must specify one value for each column of `A`. When absent, each variable has a default objective coefficient of 0.

`sense (optional)`: The senses of the linear constraints. Allowed values are `=`, `<`, or `>`. You must specify one value for each row of `A`, or a single value to specify that all constraints have the same sense. When absent, all senses default to `<`.

`rhs (optional)`: The right-hand side vector for the linear constraints (b in the `problem` statement). You must specify one value for each row of `A`. When absent, the right-hand side vector defaults to the zero vector.

`lb (optional)`: The lower bound vector. When present, you must specify one value for each column of `A`. When absent, each variable has a default lower bound of 0.

`ub (optional)`: The upper bound vector. When present, you must specify one value for each column of `A`. When absent, the variables have infinite upper bounds.

`vtype (optional)`: The variable types. This vector is used to capture variable integrality constraints. Allowed values are `C` (continuous), `B` (binary), `I` (integer), `S` (semi-continuous), or `N` (semi-integer). Binary variables must be either 0 or 1. Integer variables can take any integer value between the specified lower and upper bounds. Semi-continuous variables can take any value between the specified lower and upper bounds, or a value of zero. Semi-integer variables can take any integer value between the specified lower and upper bounds, or a value of zero. When present, you must specify one value for each column of `A`, or a single value to specify that all variables have the same type. When absent, each variable is treated as being

continuous. Refer to the variable section of the reference manual for more information on variable types.

`modelsense (optional)`: The optimization sense. Allowed values are `min` (minimize) or `max` (maximize). When absent, the default optimization sense is minimization.

`modelname (optional)`: The name of the model. The name appears in the Gurobi log, and when writing a model to a file.

`objcon (optional)`: The constant offset in the objective function (alpha in the `problem` statement).

`varnames (optional)`: The variable names vector. A cell array. When present, each element of this vector defines the name of a variable. You must specify a name for each column of `A`.

`constrnames (optional)`: The constraint names vector. A cell array. When present, each element of the vector defines the name of a constraint. You must specify a name for each row of `A`.

Quadratic objective and constraint fields

`Q (optional)`: The quadratic objective matrix. When present, `Q` must be a square matrix whose row and column counts are equal to the number of columns in `A`.

`quadcon (optional)`: The quadratic constraints. A struct array. When present, each element in `quadcon` defines a single quadratic constraint: $x^T Qc\, x + q^T x \leq$ beta.

The `Qc` matrix must be a square matrix whose row and column counts are equal to the number of columns of `A`. There are two options to store the matrix: (i) in `model.quadcon(i).Qc` as a sparse matrix; (ii) through three dense vectors `model.quadcon(i).Qrow`, `model.quadcon(i).Qcol`, and `model.quadcon(i).Qval` specifying the matrix in triple format, with row indices, column indices, and values, respectively.

The optional `q` vector defines the linear terms in the constraint. It can be a dense vector specifying a value for each column of `A` or a sparse vector (sparse n-by-1 matrix). It is stored in `model.quadcon(i).q`.

The scalar `beta` is stored in `model.quadcon(i).rhs`. It defines the right-hand side value for the constraint.

The optional `sense` string defines the sense of the quadratic constraint. Allowed values are `<`, `=` or `>`. If not present, the default sense is `<`. It is stored in `model.quadcon(i).sense`.

The optional `name` string defines the name of the quadratic constraint. It is stored in `model.quadcon(i).name`.

SOS constraint fields

`sos (optional)`: The Special Ordered Set (SOS) constraints. A struct array. When present, each entry in `sos` defines a single SOS constraint. A SOS constraint can be of type 1 or 2. The type of SOS constraint i is specified via `model.sos(i).type`. A type 1 SOS constraint is a set of variables where at most one variable in the set may take a value other than zero. A type 2 SOS constraint is an ordered set of variables where at most two variables in the set may take non-zero values. If two take non-zeros values, they must be contiguous in the

ordered set. The members of an SOS constraint are specified by placing their indices in vector `model.sos(i).index`. Weights associated with SOS members are provided in vector `model.sos(i).weight`. Please refer to SOS Constraints section in the reference manual for details on SOS constraints.

Multi-objective fields

`multiobj (optional)`: Multi-objective specification for the model. A struct array. When present, each entry in `multiobj` defines a single objective of a multi-objective problem. Please refer to the Multiple Objectives section in the reference manual for more details on multi-objective optimization. Each objective i may have the following fields:

`objn`: Specified via `model.multiobj(i).objn`. This is the i-th objective vector.

`objcon (optional)`: Specified via `model.multiobj(i).objcon`. If provided, this is the i-th objective constant. The default value is 0.

`priority (optional)`: Specified via `model.multiobj(i).priority`. If provided, this value is the *hierarchical* priority for this objective. The default value is 0.

`weight (optional)`: Specified via `model.multiobj(i).weight`. If provided, this value is the multiplier used when aggregating objectives. The default value is 1.0.

`reltol (optional)`: Specified via `model.multiobj(i).reltol`. If provided, this value specifies the relative objective degradation when doing hierarchical multi-objective optimization. The default value is 0.

`abstol (optional)`: Specified via `model.multiobj(i).abstol`. If provided, this value specifies the absolute objective degradation when doing hierarchical multi-objective optimization. The default value is 0.

`name (optional)`: Specified via `model.multiobj(i).name`. If provided, this string specifies the name of the i-th objective function.

Note that when multiple objectives are present, the `result.objval` field that is returned in the result of an optimization call will be a vector of the same length as `model.multiobj`.

A multi-objective model can't have other objectives. Thus, combining `model.multiobj` with any of `model.obj`, `model.objcon`, `model.pwlobj`, or `model.Q` is an error.

Computing an IIS

When computing an Irreducible Inconsistent Subsystem (IIS) for an infeasible model, additional model attributes for variable bounds, linear constraints, quadratic constraints and general constraints may be set in order to indicate whether the corresponding entity should be explicitly included or excluded from the IIS:

`iislbforce (optional)`: array of length equal to the number of variables. The value of `model.iislbforce(i)` specifies the IIS force attribute for the lower bound of the i-th variable.

`iisubforce (optional)`: array of length equal to the number of variables. The value of `model.iisubforce(i)` specifies the IIS force attribute for the upper bound of the i-th variable.

`iisconstrforce (optional)`: array of length equal to the number of constraints. The value of `model.iisconstrforce(i)` specifies the IIS force attribute for the i-th constraint.

`iisqconstrforce (optional)`: array of length equal to the number of quadratic constraints. The value of `model.iisqconstrforce(i)` specifies the IIS force attribute for the i-th quadratic constraint.

`iisgenconstrforce (optional)`: array of length equal to the number of general constraints. The value of `model.iisgenconstrforce(i)` specifies the IIS force attribute for the i-th general constraint.

Possible values for all five attribute arraysfrom above are: -1 to let the algorithm decide, 0 to exclude the corresponding entity from the IIS, and 1 to always include the corresponding entity in the IIS.

Note that setting this attribute to 0 may make the resulting subsystem feasible (or consistent), which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

General constraint fields

The struct arrays described below are used to add *general constraints* to a model.

Mathematical programming has traditionally defined a set of fundamental constraint types: variable bound constraints, linear constraints, quadratic constraints, integrality constraints, and SOS constraints. These are typically treated directly by the underlying solver (although not always), and are fundamental to the overall algorithm.

Gurobi accepts a number of additional constraint types, which we collectively refer to as *general (function) constraints*. These are typically *not* treated directly by the solver. Rather, they are transformed by presolve into constraints (and variables) chosen from among the fundamental types listed above. In some cases, the resulting constraint or constraints are mathematically equivalent to the original; in others, they are approximations. If such constraints appear in your model, but if you prefer to reformulate them yourself using fundamental constraint types instead, you can certainly do so. However, note that Gurobi can sometimes exploit information contained in the other constraints in the model to build a more efficient formulation than what you might create.

The additional constraint types that fall under this *general constraint* umbrella are:

- MAX (genconmax): set a decision variable equal to the maximum value from among a set of decision variables
- MIN (genconmin): set a decision variable equal to the minimum value from among a set of decision variables
- ABS (genconabs): set a decision variable equal to the absolute value of some other decision variable
- AND (genconand): set a binary variable equal to one if and only if all of a set of binary decision variables are equal to one

- OR (genconor): set a binary variable equal to one if and only if at least one variable out of a set of binary decision variables is equal to one
- NORM (genconnorm): set a decision variable equal to the p-norm of a vector of decision variables
- INDICATOR (genconind): whenever a given binary variable takes a certain value, then the given linear constraint must be satisfied
- Piecewise-linear constraints (genconpwl): set a variable equal to the piecewise-linear function defined by a set of points using some other variable
- Polynomial (genconpoly): set a variable equal to the polynomial function defined by some other variable
- Natural exponential (genconexp): set a variable equal to the natural exponential function by some other variable
- Exponential (genconexpa): set a variable equal to the exponential function by some other variable
- Natural logarithm (genconlog): set a variable equal to the natural logarithmic function by some other variable
- Logarithm (genconloga): set a variable equal to the logarithmic function by some other variable
- Logistic (genconlogistic) set a variable equal to the logistic function by some other variable
- Power (genconpow): set a variable equal to the power function by some other variable
- SIN (genconsin): set a variable equal to the sine function by some other variable
- COS (genconcos): set a variable equal to the cosine function by some other variable
- TAN (gencontan): set a variable equal to the tangent function by some other variable

Please refer to General Constraints section in the reference manual for additional details on general constraints.

`genconmax (optional)`: A struct array. When present, each entry in `genconmax` defines a MAX general constraint of the form

$$x[\text{resvar}] = \max\{\text{con}, x[j] : j \in \text{vars}\}$$

Each entry may have the following fields:

`resvar`: Specified via `model.genconmax(i).resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model.genconmax(i).vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`con (optional)`: Specified via `model.genconmax(i).con`. When present, specifies the constant on the left-hand side. Default value is $-\infty$.

`name (optional)`: Specified via `model.genconmax(i).name`. When present, specifies the name of the i-th MAX general constraint.

`genconmin (optional)`: A struct array. When present, each entry in `genconmax` defines a MIN general constraint of the form

$$x[\text{resvar}] = \min\{\text{con}, x[j] : j \in \text{vars}\}$$

Each entry may have the following fields:

`resvar`: Specified via `model.genconmin(i).resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model.genconmin(i).vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`con (optional)`: Specified via `model.genconmin(i).con`. When present, specifies the constant on the left-hand side. Default value is ∞.

`name (optional)`: Specified via `model.genconmin(i).name`. When present, specifies the name of the i-th MIN general constraint.

`genconabs (optional)`: A struct array. When present, each entry in `genconmax` defines an ABS general constraint of the form

$$x[\text{resvar}] = |x[\text{argvar}]|$$

Each entry may have the following fields:

`resvar`: Specified via `model.genconabs(i).resvar`. Index of the variable in the left-hand side of the constraint.

`argvar`: Specified via `model.genconabs(i).argvar`. Index of the variable in the right-hand side of the constraint.

`name (optional)`: Specified via `model.genconabs(i).name`. When present, specifies the name of the i-th ABS general constraint.

`genconand (optional)`: A struct array. When present, each entry in `genconand` defines an AND general constraint of the form

$$x[\text{resvar}] = \text{and}\{x[i] : i \in \text{vars}\}$$

Each entry may have the following fields:

`resvar`: Specified via `model.genconand(i).resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model.genconand(i).vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`name (optional)`: Specified via `model.genconand(i).name`. When present, specifies the name of the i-th AND general constraint.

`genconor (optional)`: A struct array. When present, each entry in `genconor` defines an OR general constraint of the form

$$x[\text{resvar}] = \text{or}\{x[i] : i \in \text{vars}\}$$

Each entry may have the following fields:

`resvar`: Specified via `model.genconor(i).resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model.genconor(i).vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`name (optional)`: Specified via `model.genconor(i).name`. When present, specifies the name of the i-th OR general constraint.

`genconnorm (optional)`: A struct array. When present, each entry in `genconnorm` defines a NORM general constraint of the form

$$x[\text{resvar}] = \text{norm}(x[i] : i \in \text{vars}, \text{which})$$

Each entry may have the following fields:

`resvar`: Specified via `model.genconnorm(i).resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model.genconnorm(i).vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`which`: Specified via `model.genconnorm(i).which`. Specifies which p-norm to use. Possible values are 0, 1, 2 and ∞.

`name (optional)`: Specified via `model.genconnorm(i).name`. When present, specifies the name of the i-th NORM general constraint.

`genconind (optional)`: A struct array. When present, each entry in `genconind` defines an INDICATOR general constraint of the form

$$x[\text{binvar}] = \text{binval} \Rightarrow \sum (x[j] \cdot \text{a}[j]) \ \text{ sense rhs}$$

This constraint states that when the binary variable x[binvar] takes the value `binval` then the linear constraint $\sum (x[\text{vars}[j]] \cdot \text{val}[j])$ sense rhs must hold. Note that `sense` is one of `=`, `<`, or `>` for equality ($=$), less than or equal ($\leq$) or greater than or equal ($\geq$) constraints. Each entry may have the following fields:

`binvar`: Specified via `model.genconind(i).binvar`. Index of the implicating binary variable.

`binval`: Specified via `model.genconind(i).binval`. Value for the binary variable that forces the following linear constraint to be satisfied. It can be either 0 or 1.

`a`: Specified via `model.genconind(i).a`. Vector of coefficients of variables participating in the implied linear constraint. You can specify a value for `a` for each column of `A` (dense vector) or pass a sparse vector (sparse n-by-1 matrix).

`sense`: Specified via `model.genconind(i).sense`. Sense of the implied linear constraint. Must be one of `=`, `<`, or `>`.

`rhs`: Specified via `model.genconind(i).rhs`. Right-hand side value of the implied linear constraint.

`name (optional)`: Specified via `model.genconind(i).name`. When present, specifies the name of the i-th INDICATOR general constraint.

`genconpwl (optional)`: A struct array. When present, each entry in `genconpwl` defines a piecewise-linear constraint of the form

$$x[\text{yvar}] = f(x[\text{xvar}])$$

The breakpoints for f are provided as arguments. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined

Each entry may have the following fields:

`xvar`: Specified via `model.genconpwl(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconpwl(i).yvar`. Index of the variable in the left-hand side of the constraint.

`xpts`: Specified via `model.genconpwl(i).xpts`. Specifies the x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

`ypts`: Specified via `model.genconpwl(i).ypts`. Specifies the y values for the points that define the piecewise-linear function.

`name (optional)`: Specified via `model.genconpwl(i).name`. When present, specifies the name of the i-th piecewise-linear general constraint.

`genconpoly (optional)`: A struct array. When present, each entry in `genconpoly` defines a polynomial function constraint of the form

$$x[\text{yvar}] = p_0 x[\text{xvar}]^d + p_1 x[\text{xvar}]^{d-1} + ... + p_{d-1} x[\text{xvar}] + p_d$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconpoly(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconpoly(i).yvar`. Index of the variable in the left-hand side of the constraint.

`p`: Specified via `model.genconpoly(i).p`. Specifies the coefficients for the polynomial function (starting with the coefficient for the highest power). If x^d is the highest power term, a dense vector of length $d+1$ is returned.

`name (optional)`: Specified via `model.genconpoly(i).name`. When present, specifies the name of the i-th polynomial function constraint.

`funcpieces (optional)`: Specified via `model.genconpoly(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th polynomial function constraint.

`funcpiecelength (optional)`: Specified via `model.genconpoly(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th polynomial function constraint.

`funcpieceerror (optional)`: Specified via `model.genconpoly(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th polynomial function constraint.

`funcpieceratio (optional)`: Specified via `model.genconpoly(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th polynomial function constraint.

`funcnonlinear (optional)`: Specified via `model.genconpoly(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th polynomial function constraint.

`genconexp (optional)`: A struct array. When present, each entry in `genconexp` defines the natural exponential function constraint of the form

$$x[\text{yvar}] = \exp(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconexp(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconexp(i).yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model.genconexp(i).name`. When present, specifies the name of the i-th natural exponential function constraint.

`funcpieces (optional)`: Specified via `model.genconexp(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th natural exponential function constraint.

`funcpiecelength (optional)`: Specified via `model.genconexp(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th natural exponential function constraint.

`funcpieceerror (optional)`: Specified via `model.genconexp(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th natural exponential function constraint.

`funcpieceratio (optional)`: Specified via `model.genconexp(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th natural exponential function constraint.

`funcnonlinear (optional)`: Specified via `model.genconexp(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th natural exponential function constraint.

`genconexpa (optional)`: A struct array. When present, each entry in `genconexpa` defines an exponential function constraint of the form

$$x[\text{yvar}] = \text{a}^{x[\text{xvar}]}$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconexpa(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconexpa(i).yvar`. Index of the variable in the left-hand side of the constraint.

`a`: Specified via `model.genconexpa(i).a`. Specifies the base of the exponential function $a > 0$.

`name (optional)`: Specified via `model.genconexpa(i).name`. When present, specifies the name of the i-th exponential function constraint.

`funcpieces (optional)`: Specified via `model.genconexpa(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th exponential function constraint.

`funcpiecelength (optional)`: Specified via `model.genconexpa(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th exponential function constraint.

`funcpieceerror (optional)`: Specified via `model.genconexpa(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th exponential function constraint.

`funcpieceratio (optional)`: Specified via `model.genconexpa(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th exponential function constraint.

`funcnonlinear (optional)`: Specified via `model.genconexpa(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th exponential function constraint.

`genconlog (optional)`: A struct array. When present, each entry in `genconlog` defines the natural logarithmic function constraint of the form

$$x[\text{yvar}] = \log(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconlog(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconlog(i).yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model.genconlog(i).name`. When present, specifies the name of the i-th natural logarithmic function constraint.

`funcpieces (optional)`: Specified via `model.genconlog(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th natural logarithmic function constraint.

`funcpiecelength (optional)`: Specified via `model.genconlog(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th natural logarithmic function constraint.

`funcpieceerror (optional)`: Specified via `model.genconlog(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th natural logarithmic function constraint.

`funcpieceratio (optional)`: Specified via `model.genconlog(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th natural logarithmic function constraint.

`funcnonlinear (optional)`: Specified via `model.genconlog(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th natural logarithmic function constraint.

`genconloga (optional)`: A struct array. When present, each entry in `genconloga` defines a logarithmic function constraint of the form

$$x[\text{yvar}] = \log(x[\text{xvar}]) \setminus \log(a)$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconloga(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconloga(i).yvar`. Index of the variable in the left-hand side of the constraint.

`a`: Specified via `model.genconloga(i).a`. Specifies the base of the logarithmic function $a > 0$.

`name (optional)`: Specified via `model.genconloga(i).name`. When present, specifies the name of the i-th logarithmic function constraint.

`funcpieces (optional)`: Specified via `model.genconloga(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th logarithmic function constraint.

`funcpiecelength (optional)`: Specified via `model.genconloga(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th logarithmic function constraint.

`funcpieceerror (optional)`: Specified via `model.genconloga(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th logarithmic function constraint.

`funcpieceratio (optional)`: Specified via `model.genconloga(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th logarithmic function constraint.

`funcnonlinear (optional)`: Specified via `model.genconloga(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th logarithmic function constraint.

`genconlogistic (optional)`: A struct array. When present, each entry in `genconlog` defines the logistic function constraint of the form

$$x[\text{yvar}] = 1/(1 + \exp(-x[\text{xvar}]))$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconlogistic(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconlogistic(i).yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model.genconlogistic(i).name`. When present, specifies the name of the i-th logistic function constraint.

`funcpieces (optional)`: Specified via `model.genconlogistic(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th logistic function constraint.

`funcpiecelength (optional)`: Specified via `model.genconlogistic(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th logistic function constraint.

`funcpieceerror (optional)`: Specified via `model.genconlogistic(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th logistic function constraint.

`funcpieceratio (optional)`: Specified via `model.genconlogistic(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th logistic function constraint.

`funcnonlinear (optional)`: Specified via `model.genconlogistic(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th logistic function constraint.

`genconpow (optional)`: A struct array. When present, each entry in `genconpow` defines a power function constraint of the form

$$x[\text{yvar}] = x[\text{xvar}]^{\text{a}}$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconpow(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconpow(i).yvar`. Index of the variable in the left-hand side of the constraint.

`a`: Specified via `model.genconpow(i).a`. Specifies the exponent of the power function.

`name (optional)`: Specified via `model.genconpow(i).name`. When present, specifies the name of the i-th power function constraint.

`funcpieces (optional)`: Specified via `model.genconpow(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th power function constraint.

`funcpiecelength (optional)`: Specified via `model.genconpow(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th power function constraint.

`funcpieceerror (optional)`: Specified via `model.genconpow(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th power function constraint.

`funcpieceratio (optional)`: Specified via `model.genconpow(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th power function constraint.

`funcnonlinear (optional)`: Specified via `model.genconpow(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th power function constraint.

`genconsin (optional)`: A struct array. When present, each entry in `genconsin` defines the sine function constraint of the form

$$x[\text{yvar}] = \sin(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconsin(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconsin(i).yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model.genconsin(i).name`. When present, specifies the name of the i-th sine function constraint.

`funcpieces (optional)`: Specified via `model.genconsin(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th sine function constraint.

`funcpiecelength (optional)`: Specified via `model.genconsin(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th sine function constraint.

`funcpieceerror (optional)`: Specified via `model.genconsin(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th sine function constraint.

`funcpieceratio (optional)`: Specified via `model.genconsin(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th sine function constraint.

`funcnonlinear (optional)`: Specified via `model.genconsin(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th sine function constraint.

`genconcos (optional)`: A struct array. When present, each entry in `genconcos` defines the cosine function constraint of the form

$$x[\text{yvar}] = \cos(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.genconcos(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.genconcos(i).yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model.genconcos(i).name`. When present, specifies the name of the i-th cosine function constraint.

`funcpieces (optional)`: Specified via `model.genconcos(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th cosine function constraint.

`funcpiecelength (optional)`: Specified via `model.genconcos(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th cosine function constraint.

`funcpieceerror (optional)`: Specified via `model.genconcos(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th cosine function constraint.

`funcpieceratio (optional)`: Specified via `model.genconcos(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th cosine function constraint.

`funcnonlinear (optional)`: Specified via `model.genconcos(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th cosine function constraint.

`gencontan (optional)`: A struct array. When present, each entry in `gencontan` defines the tangent function constraint of the form

$$x[\text{yvar}] = \tan(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following fields:

`xvar`: Specified via `model.gencontan(i).xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model.gencontan(i).yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model.gencontan(i).name`. When present, specifies the name of the i-th tangent function constraint.

`funcpieces (optional)`: Specified via `model.gencontan(i).funcpieces`. When present, specifies the FuncPieces attribute of the i-th tangent function constraint.

`funcpiecelength (optional)`: Specified via `model.gencontan(i).funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th tangent function constraint.

`funcpieceerror (optional)`: Specified via `model.gencontan(i).funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th tangent function constraint.

`funcpieceratio (optional)`: Specified via `model.gencontan(i).funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th tangent function constraint.

`funcnonlinear (optional)`: Specified via `model.gencontan(i).funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th tangent function constraint.

Advanced fields

`pwlobj (optional)`: The piecewise-linear objective functions. A struct array. When present, each entry in `pwlobj` defines a piecewise-linear objective function for a single variable. The index of the variable whose objective function is being defined is stored in `model.pwlobj(i).var`. The x values for the points that define the piecewise-linear function are stored in `model.pwlobj(i).x`. The values in the x vector must be in non-decreasing order. The y values for the points that define the piecewise-linear function are stored in `model.pwlobj(i).y`.

`vbasis (optional)`: The variable basis status vector. Used to provide an advanced starting point for the simplex algorithm. You would generally never concern yourself with the contents of this vector, but would instead simply pass it from the result of a previous optimization run to the input of a subsequent run. When present, you must specify one value for each column of `A`.

`cbasis (optional)`: The constraint basis status vector. Used to provide an advanced starting point for the simplex algorithm. Consult the `vbasis` description for details. When present, you must specify one value for each row of `A`.

`varhintval (optional)`: A set of user hints. If you know that a variable is likely to take a particular value in high quality solutions of a MIP model, you can provide that value as a hint. You can also (optionally) provide information about your level of confidence in a hint with the `varhintpri` field. If present, you must specify one value for each column of `A`. Use a value of `nan` for variables where no such hint is known. For more details, please refer to the Attribute section in the reference manual.

`varhintpri (optional)`: Priorities on user hints. After providing variable hints through the `varhintval` struct, you can optionally also provide hint priorities to give an indication of your level of confidence in your hints. If present, you must specify a value for each column of `A`. For more details, please refer to the Attribute section in the reference manual.

`branchpriority (optional)`: Variable branching priority. If present, the value of this attribute is used as the primary criteria for selecting a fractional variable for branching during the MIP search. Variables with larger values always take priority over those with smaller values. Ties are broken using the standard branch variable selection criteria. If present, you must specify one value for each column of `A`.

`pstart (optional)`: The current simplex start vector. If you set `pstart` values for every variable in the model and `dstart` values for every constraint, then simplex will use those values to compute a warm start basis. For more details, please refer to the Attribute section in the reference manual.

`dstart (optional)`: The current simplex start vector. If you set `dstart` values for every linear constraint in the model and `pstart` values for every variable, then simplex will use those values to compute a warm start basis. For more details, please refer to the Attribute section in the reference manual.

`lazy (optional)`: Determines whether a linear constraint is treated as a *lazy constraint*. If present, you must specify one value for each row of `A`. For more details, please refer to the Attribute section in the reference manual.

`start (optional)`: The MIP start vector. The MIP solver will attempt to build an initial solution from this vector. When present, you must specify a start value for each variable. Note that you can set the start value for a variable to `nan`, which instructs the MIP solver to try to fill in a value for that variable.

`partition (optional)`: The MIP variable partition number, which is used by the MIP solution improvement heuristic. If present, you must specify one value for each variable of `A`. For more details, please refer to the Attribute section in the reference manual.

If any of the mandatory components listed above are missing, the `gurobi()` function will return an error.

Below is an example that demonstrates the construction of a simple optimization model:

```
model.A = sparse([1 2 3; 1 1 0]);
model.obj = [1 1 1];
model.modelsense = 'max';
model.rhs = [4; 1];
model.sense = '<>'
```

The params argument

As mentioned previously, the Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization.

Parameter changes are specified using a `struct` variable having multiple `fields`, which is passed as an argument to the appropriate Gurobi function (e.g., `gurobi`). The name of each field must be the name of a Gurobi parameter, and the associated value should be the desired value of that parameter. You can find a complete list of the available Gurobi parameters in the reference manual.

To create a struct that would set the Gurobi `Method` parameter to 2 and the `ResultFile` parameter to `model.mps`, you would do the following:

```
params.Method = 2;
params.ResultFile = 'model.mps';
```

We should say a bit more about the `ResultFile` parameter. If this parameter is set, the optimization model that is eventually passed to Gurobi will also be output to the specified file. The filename suffix should be one of `.mps`, `.lp`, `.rew`, `.rlp`, `.dua`, or `.dlp`, to indicate the desired file format (see the file format section in the reference manual for details on Gurobi file formats).

The params struct can also be used to set license specific parameters, that define the computational environment to be used. We will discuss the two most common use cases next, and refer again to the collection of all available parameters in the reference manual.

Using a Compute Server License

Gurobi Compute Server allows you to offload optimization jobs to a remote server. Servers are organized into clusters. By providing the name of any node within the cluster, your job will automatically be sent to the least heavily loaded node in the cluster. If all nodes are at capacity, your job will be placed in a queue, and will proceed once capacity becomes available. You can find additional information about Gurobi Compute Server in the Gurobi Remote Services Reference Manual. The most commonly used parameters are the following.

`ComputeServer`: A Compute Server. You can refer to the server using its name or its IP address. If you are using a non-default port, the server name should be followed by the port number (e.g., `server1:61000`).

`ServerPassword (optional)`: User password on the Compute Server cluster. Obtain this from your Compute Server administrator.

`CSPriority (optional)`: The priority of the job. Priorities must be between -100 and 100, with a default value of 0 (by convention). Higher priority jobs are chosen from the server job queue before lower priority jobs. A job with priority 100 runs immediately, bypassing the job queue and ignoring the job limit on the server. You should exercise caution with priority 100 jobs, since they can severely overload a server, which can cause jobs to fail, and in extreme cases can cause the server to crash.

`CSRouter (optional)`: The router for the Compute Server cluster. A router can be used to improve the robustness of a Compute Server deployment. You can refer to the router using either its name or its IP address. A typical Remote Services deployment won't use a router, so you typically won't need to set this.

`CSTLSinsecure (optional)`: Indicates whether to use insecure mode in the TLS (Transport Layer Security). Leave this at its default value of 0 unless your server administrator tells you otherwise.

Here is an example of how to use a `params` argument to connect to a Compute Server:

```
params.ComputeServer = 'server1.mycompany.com:61000';
params.CSPriority = 5;
```

Using a Gurobi Instant Cloud License

Gurobi Instant Cloud allows you to offload optimization jobs to a Gurobi Compute Server on the cloud. If an appropriate machine is already running, the job will run on that machine. It will automatically launch a new machine otherwise. Note that launching a new machine can take a few minutes. You can find additional information about the Gurobi Instant Cloud service in the reference manual. The most commonly used parameters are the following.

`CloudAccessID`: The access ID for your Gurobi Instant Cloud license. This can be retrieved from the Gurobi Instant Cloud website. When used in combination with your `CloudSecretKey`, this allows you to launch Instant Cloud instances and submit jobs to them.

`CloudSecretKey`: The secret key for your Gurobi Instant Cloud license. This can be retrieved from the Gurobi Instant Cloud website. When used in combination with your `CloudAccessID`, this allows you to launch Instant Cloud instances and submit jobs to them. Note that you should keep your secret key private.

`CloudPool (optional)`: The machine pool. Machine pools allow you to create fixed configurations on the Instant Cloud website (capturing things like type of machine, geographic region, etc.), and then launch and share machines from client programs without having to restate configuration information each time you launch a machine. If not provided, your job will be launched in the default pool associated with your cloud license.

`CSPriority (optional)`: The priority of the job. Priorities must be between -100 and 100, with a default value of 0 (by convention). Higher priority jobs are chosen from the server job queue

before lower priority jobs. A job with priority 100 runs immediately, bypassing the job queue and ignoring the job limit on the server. You should exercise caution with priority 100 jobs, since they can severely overload a server, which can cause jobs to fail, and in extreme cases can cause the server to crash.

Here is an example of how to use a `params` argument to launch a Gurobi Instant Cloud instance:

```
params.CloudAccessID = '3d1ecef9-dfad-eff4-b3fa';
params.CloudSecretKey = 'ae6L23alJe3+fas';
```

8.2 Solving a Model

gurobi()

```
gurobi  ( model )
gurobi  ( model, params )
```

This function optimizes the given model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion it will return a `struct` variable containing solution information.

Please consult Variables and Constraints section in the reference manual for a discussion of some of the practical issues associated with solving a precisely defined mathematical model using finite-precision floating-point arithmetic.

Arguments:

`model`: The model `struct` must contain a valid Gurobi model. See the `model` argument section for more information.

`params`: The params `struct`, when provided, contains a list of modified Gurobi parameters. See the `params` argument section for more information.

Example usage:

```
result = gurobi(model, params);
if strcmp(result.status, 'OPTIMAL');
  fprintf('Optimal objective: %e\n', result.objval);
  disp(result.x)
else
  fprintf('Optimization returned status: %s\n', result.status);
end
```

Return value:

The optimization result

The `gurobi` function returns a `struct`, with the various results of the optimization stored in its fields. The specific results that are available depend on the type of model that was solved, the parameters used, and the status of the optimization. The following is a list of fields that might be available in the returned result. We will discuss the circumstances under which each will be available after presenting the list.

Model fields

`status`: The status of the optimization, returned as a string. The desired result is `OPTIMAL`, which indicates that an optimal solution to the model was found. Other status are possible, for example if the model has no feasible solution or if you set a Gurobi parameter that leads to early solver termination. See the Status Code section for further information on the Gurobi status codes.

`objval`: The objective value of the computed solution. Note that for multi-objective models `result.objval` will be a vector, where `result.objval(i)` stores the value for `model.multiobj(i)`.

`objbound`: Best available bound on solution (lower bound for minimization, upper bound for maximization).

`objboundc`: The best unrounded bound on the optimal objective. In contrast to `objbound`, this attribute does not take advantage of objective integrality information to round to a tighter bound. For example, if the objective is known to take an integral value and the current best bound is 1.5, `objbound` will return 2.0 while `objboundc` will return 1.5.

`mipgap`: Current relative MIP optimality gap; computed as $|ObjBound - ObjVal|/|ObjVal|$ (where $ObjBound$ and $ObjVal$ are the MIP objective bound and incumbent solution objective, respectively). Returns `GRB_INFINITY` when an incumbent solution has not yet been found, when no objective bound is available, or when the current incumbent objective is 0. This is only available for mixed-integer problems.

`runtime`: The elapsed wall-clock time (in seconds) for the optimization.

`work`: The work (in work units) spent on the optimization. As opposed to the runtime in seconds, the work is deterministic. This means that on the same hardware and with the same parameter and attribute settings, solving the same model twice will lead to exactly the same amount of work in each of the two solves. One work unit corresponds very roughly to one second, but this greatly depends on the hardware on which Gurobi is running and on the model that has been solved.

`itercount`: Number of simplex iterations performed.

`baritercount`: Number of barrier iterations performed.

`nodecount`: Number of branch-and-cut nodes explored.

`maxvio`: Value of the maximal (unscaled) violation of the returned solution.

`farkasproof`: Magnitude of infeasibility violation in Farkas infeasibility proof. Only available if the model was found to be infeasible. Please refer to Attribute section in the reference manual for details.

Variable fields

`x`: The computed solution. This vector contains one entry for each column of `A`.

`rc`: Variable reduced costs for the computed solution. This vector contains one entry for each column of `A`.

`vbasis`: Variable basis status values for the computed optimal basis. You generally should not concern yourself with the contents of this vector. If you wish to use an advanced start later, you would simply copy the `vbasis` and `cbasis` fields into the corresponding fields for the next model. This vector contains one entry for each column of `A`.

`unbdray`: Unbounded ray. Provides a vector that, when added to any feasible solution, yields a new solution that is also feasible but improves the objective. Only available if the model is found to be unbounded. This vector contains one entry for each column of `A`.

Linear constraint fields

`slack`: The constraint slack for the computed solution. This vector contains one entry for each row of `A`.

`pi`: Dual values for the computed solution (also known as *shadow prices*). This vector contains one entry for each row of `A`.

`cbasis`: Constraint basis status values for the computed optimal basis. This vector contains one entry for each row of `A`.

`farkasdual`: Farkas infeasibility proof. Only available if the model was found to be infeasible. Please refer to Attribute section in the reference manual for details.

Quadratic constraint fields

`qcslack`: The quadratic constraint slack in the current solution. This vector contains one entry for each quadratic constraint.

`qcpi`: The dual values associated with the quadratic constraints. This vector contains one entry for each quadratic constraint.

Solution Pool fields

`pool`: When multiple solutions are found during the optimization call, these solutions are returned in this field. A struct array. When present, each struct has the following fields:

- `objval`: Stores the objective value of the i-th solution in `result.pool(i).objval`. Note that when the model is a multi-objective model, instead of a single value, `result.pool(i).objval(j)` stores the value of the j-th objective function for the i-th solution.
- `xn`: Stores the i-th solution in `result.pool(i).xn`. This vector contains one entry for each column of `A`.

Note that to query the number of solutions stored, you can query the length of `result.pool`.

`poolobjbound`: For single-objective MIP optimization problems, this value gives a bound on the best possible objective of an undiscovered solution. The difference between this value and `objbound` is that the former gives an objective bound for undiscovered solutions, while the latter gives a bound for any solution.

What is Available When

The `status` field will be present in all cases. It indicates whether Gurobi was able to find a proven optimal solution to the model. In cases where a solution to the model was found, optimal or otherwise, the `objval` and `x` fields will be present.

For linear and quadratic programs, if a solution is available, then the `pi` and `rc` fields will also be present. For models with quadratic constraints, if the parameter `qcpdual` is set to 1, the field `qcpi` will be present. If the final solution is a *basic* solution (computed by simplex), then `vbasis` and `cbasis` will be present. If the model is an unbounded linear program and the InfUnbdInfo parameter is set to 1, the field `unbdray` will be present. Finally, if the model is an infeasible linear

program and the InfUnbdInfo parameter is set to 1, the fields `farkasdual` and `farkasproof` will be set.

For mixed integer problems, no dual information (i.e. `pi`, `slack`, `rc`, `vbasis`, `cbasis`, `qcslack`, `qcpi`, `ubdray` or `farkasdual`) is ever available. When multiple solutions are found, the `pool` and `poolobjbound` fields will be present. Depending on the `status` field value, the fields `nodecount`, `objbound`, `objbundc` and `mipgap` will be available.

For continuous and mixed-integer models, under normal execution, the fields `runtime`, `work`, `itercount` and `baritercount` will be available.

gurobi_iis()

```
gurobi_iis  ( model )
gurobi_iis  ( model, params )
```

Compute an Irreducible Inconsistent Subsystem (IIS).

An IIS is a subset of the constraints and variable bounds with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the smallest one; there may exist others with fewer constraints or bounds.

You can obtain information about the outcome of the IIS computation from the returned IIS result (described below). Note that this method can be used to compute IISs for both continuous and MIP models.

Arguments:

`model`: The model `struct` must contain a valid Gurobi model. See the `model` argument section for more information.

`params`: The params `struct`, when provided, contains a list of modified Gurobi parameters. See the `params` argument section for more information.

Example usage:

```
model = gurobi_read('examples/data/klein1.mps');
iis = gurobi_iis(model);
```

Return value:

The `gurobi_iis()` function returns a `struct`, with various results stored in its fields. The specific results that are available depend on the type of model.

The returned `struct` will always contain the following `fields`:

`minimal`: A logical scalar that indicates whether the computed IIS is minimal. It will normally be true, but it may be false if the IIS computation was stopped early (due to a time limit or a user interrupt).

`Arows`: A logical vector that indicates whether a linear constraint appears in the computed IIS.

`lb`: A logical vector that indicates whether a lower bound appears in the computed IIS.

`ub`: A logical vector that indicates whether a upper bound appears in the computed IIS.

If your model contains general constraints, the returned `struct` will also contain the following `fields`:

`genconmax`: A logical vector that indicates whether a general MAX constraint appears in the computed IIS.

`genconmin`: A logical vector that indicates whether a general MIN constraint appears in the computed IIS.

`genconand`: A logical vector that indicates whether a general AND constraint appears in the computed IIS.

`genconor`: A logical vector that indicates whether a general OR constraint appears in the computed IIS.

`genconabs`: A logical vector that indicates whether a general ABS constraint appears in the computed IIS.

`genconind`: A logical vector that indicates whether a general INDICATOR constraint appears in the computed IIS.

`genconpwl`: A logical vector that indicates whether a general piecewise-linear function constraint appears in the computed IIS.

`genconpoly`: A logical vector that indicates whether a polynomial function constraint appears in the computed IIS.

`genconexp`: A logical vector that indicates whether a natural exponential function constraint appears in the computed IIS.

`genconexpa`: A logical vector that indicates whether a exponential function constraint appears in the computed IIS.

`genconlog`: A logical vector that indicates whether a natural logarithmic function constraint appears in the computed IIS.

`genconloga`: A logical vector that indicates whether a logarithmic function constraint appears in the computed IIS.

`genconlogistic`: A logical vector that indicates whether a logistic function constraint appears in the computed IIS.

`genconpow`: A logical vector that indicates whether a power function constraint appears in the computed IIS.

`genconsin`: A logical vector that indicates whether a SIN function constraint appears in the computed IIS.

genconcos: A logical vector that indicates whether a COS function constraint appears in the computed IIS.

gencontan: A logical vector that indicates whether a TAN function constraint appears in the computed IIS.

If your model contains SOS constraints, the returned struct will also contain the following field:

sos: A logical vector that indicates whether an SOS constraint appears in the computed IIS

If your model contains quadratic constraints, the returned struct will also contain the following field:

quadcon: A logical vector that indicates whether a quadratic constraint appears in the computed IIS.

gurobi_feasrelax()

```
gurobi_feasrelax  ( model, relaxobjtype, minrelax, penalties )
gurobi_feasrelax  ( model, relaxobjtype, minrelax, penalties, params )
```

This function computes a feasibility relaxation for the input model argument. The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. You must provide a penalty to associate with relaxing each individual bound or constraint (through the penalties argument). These penalties are interpreted in different ways, depending on the value of the relaxobjtype argument.

Arguments:

model: The model struct must contain a valid Gurobi model. See the model argument section for more information.

relaxobjtype: The approach used to impose penalties on violations.

If you specify relaxobjtype=0, the objective for the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations.

If you specify relaxobjtype=1, the objective for the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations.

If you specify relaxobjtype=2, the objective for the feasibility relaxation is to minimize the weighted count of bound and constraint violations.

In all cases, the weights are taken from penalties.lb, penalties.ub and penalties.rhs. You can provide the special penalty value Inf to indicate that the corresponding bound or constraint cannot be relaxed.

minrelax: The minrelax argument is a boolean that controls the type of feasibility relaxation that is created. If minrelax=false, optimizing the returned model gives a solution that minimizes the cost of the violation. If minrelax=true, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that gurobi_feasrelax must solve an optimization problem to find the minimum possible relaxation when minrelax=true, which can be quite expensive.

penalties: The penalties argument is a struct array, having the following optional fields (default: all Inf):

lb Penalty for violating each lower bound.

ub Penalty for violating each upper bound.

rhs Penalty for violating each constraint.

To give an example, if a constraint with penalties.rhs value p is violated by 2.0, it would contribute 2*p to the feasibility relaxation objective for relaxobjtype=0, 2*2*p for relaxobjtype=1, and p for relaxobjtype=2.

params: The params struct, when provided, contains a list of modified Gurobi parameters. See the params argument section for more information.

Return value:

A struct containing two fields:

result.model, a struct variable, as described in the model argument section.

result.feasobj, a scalar. If minrelax==true this is the relaxation problem objective value, 0.0 otherwise.

Example usage:

```
model = gurobi_read('stein9.mps');
penalties.lb = ones(length(model.lb),1);
penalties.ub = ones(length(model.ub),1);
penalties.rhs = ones(length(model.rhs),1);
feasrelaxresult = gurobi_feasrelax(model, 0, false, penalties);
```

gurobi_relax()

```
gurobi_relax  ( model )
gurobi_relax  ( model, params )
```

Create the relaxation of a MIP model. Transforms integer variables into continuous variables, and removes SOS and general constraints.

Arguments:

model: The model struct must contain a valid Gurobi model. See the model argument section for more information.

params: The params struct, when provided, contains a list of modified Gurobi parameters. See the params argument section for more information.

Return value:

A model struct variable, as described in the model parameter section.

Example usage:

```
model = gurobi_read('stein9.mps');
relaxed = gurobi_relax(model);
```

8.3 Input-Output

gurobi_read()

```
gurobi_read  ( filename )
gurobi_read  ( filename, params )
```

Reads a model from a file.

Arguments:

filename: Name of the file to read. Note that the type of the file is encoded in the file name suffix. The filename suffix should be one of .mps, .rew, .lp, .rlp, .dua, .dlp, .ilp, or .opb (see the file formats section for details on Gurobi file formats). The files can be compressed, so additional suffixes of .gz, .bz2, .zip, or .7z are accepted.

params: The params **struct**, when provided, contains a list of modified Gurobi parameters. See the params argument section for more information.

Return value:

A **model** struct variable, as described in the model section.

Example usage:

```
model = gurobi_read('stein9.mps');
result = gurobi(model);
```

gurobi_write()

```
gurobi_write  ( model, filename )
gurobi_write  ( model, filename, params )
```

Writes a model to a file.

Arguments:

model: The model **struct** must contain a valid Gurobi model. See the model argument section for more information.

filename: Name of the file to write. Note that the type of the file is encoded in the file name suffix. The filename suffix should be one of .mps, .rew, .lp, .rlp, .dua, .dlp, or .ilp, to indicate the desired file format (see the file formats section for details on Gurobi file formats). The files can be compressed, so additional suffixes of .gz, .bz2, .zip, or .7z are accepted.

params: The params **struct**, when provided, contains a list of modified Gurobi parameters. See the params argument section for more information.

Example usage:

```
model.A          = sparse([1 2 3; 1 1 0]);
model.obj        = [1 1 2];
model.modelsense = 'max';
model.rhs        = [4; 1];
model.sense      = '<>';
gurobi_write(model, 'mymodel.mps');
gurobi_write(model, 'mymodel.lp');
gurobi_write(model, 'mymodel.mps.bz2');
```

8.4 Using Gurobi within MATLAB's Problem-Based Optimization

Starting with release R2017b, the MATLAB Optimization Toolbox offers an alternative way to formulate optimization problems, coined "Problem-Based Optimization". In this section we'll explain how this modeling technique can be used in combination with the Gurobi solver.

The problem-based modeling approach uses an object-oriented paradigm for the components of an optimization problem; the optimization problem itself, the decision variables, and the linear constraints are represented by objects. Their creation and modification is effected through methods. The complete documentation for problem-based optimization is part of the Optimization Toolbox; we will only walk through a simple example. For this it is important that your MATLAB path contains Gurobi's example directory, which can be set as follows:

```
addpath(fullfile(<path_to_Gurobi>, <architecture>, 'examples', 'matlab'));
```

The first step is to create an optimization problem:

```
prob = optimproblem('ObjectiveSense','maximize');
```

The variable `prob` now refers to an optimization problem object, which we have specified to be a maximization problem. Next we create three non-negative optimization variables: `x`, `y` and `z`:

```
x = optimvar('x', 'LowerBound', 0);
y = optimvar('y', 'LowerBound', 0);
z = optimvar('z', 'LowerBound', 0);
```

With these variables at hand, we now build linear expressions in order to set an objective function, and to add two linear constraints to `prob`:

```
prob.Objective = x + 2 * y + 3 * z;
prob.Constraints.cons1 = x + y <= 1;
prob.Constraints.cons2 = y + z <= 1;
```

Finally we create an options object that guides `prob`'s solution method to the linear program solver function `linprog`, and call the `solve` method.

```
options = optimoptions('linprog');
sol = solve(prob, options);
```

Since the `examples` directory of the Gurobi installation has been added to the path in the very first step above, a bit of magic happens at this stage: The directory contains a file `linprog.m`, so that the invocation of the `solve` method ends up calling this latter function instead of the built-in function `linprog` of MATLAB's Optimization Toolbox. The following output from Gurobi will be shown on the console:

```
Gurobi Optimizer version 11.0.0 build v11.0.0rc0 (linux64)
Optimize a model with 2 rows, 3 columns and 4 nonzeros
Model fingerprint: 0x3a4c68c2
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 3e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 2 rows and 3 columns
Presolve time: 0.03s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -4.0000000e+00   0.000000e+00   0.000000e+00      0s
```

```
Solved in 0 iterations and 0.05 seconds
Optimal objective -4.000000000e+00
```

The example we just discussed can be found in the `examples` directory in the file `opttoolbox_lp.m`. The example `opttoolbox_mip1.m` shows an analogous problem formulation with integer variables, that uses the function `intlinprog.m`, also found in the Gurobi examples directory, as a surrogate for MATLAB's built-in counterpart.

The modeling constructs provided by the Optimization Toolbox do not cover all the features of Gurobi, e.g., SOS, semi-continuous variables and general constraints to name a few. Moreover not all Gurobi parameters have equivalent counterparts in the option objects for `linprog` and `intlinprog`. In order to use such features, Gurobi's own Matlab API should be used.

8.5 Setting up the Gurobi MATLAB interface

In order to use our MATLAB interface, you'll need to use the MATLAB function `gurobi_setup` to tell MATLAB where to find the Gurobi `mex` file. This file is stored in the `<installdir>/matlab` directory of your Gurobi installation. For example, if you installed the 64-bit Windows version of Gurobi 11.0 in the default location, you should run

```
 >> cd c:/Users/jones/gurobi1100/win64/matlab
>> gurobi_setup
```

The `gurobi_setup` function adjusts your MATLAB path to include the `<installdir>/matlab` directory. If you want to avoid typing this command every time you start MATLAB, follow the instructions issued by `gurobi_setup` to permanently adjust your path.

The MATLAB examples provided with the Gurobi distribution are included in the `<installdir>/examples/matlab` directory. To run these examples you need to change to this directory. For example, if you are running the 64-bit Windows version of Gurobi, you would say:

```
 >> cd c:/Users/jones/gurobi1100/win64/examples/matlab
>> mip1
```

If the Gurobi package was successfully installed, you should see the following output:

```
          status: 'OPTIMAL'
     versioninfo: [1x1 struct]
         runtime: 2.9397e-04
          objval: 3
               x: [3x1 double]
           slack: [2x1 double]
    poolobjbound: 3
            pool: [1x2 struct]
          mipgap: 0
        objbound: 3
       objboundc: 3
       itercount: 0
    baritercount: 0
       nodecount: 0

x 1
y 0
z 1
Obj: 3.000000e+00
```

R API Overview

This section documents the Gurobi R interface. For those of you who are not familiar with R, it is a free language for statistical computing. Please visit the R web site for more information. This manual begins with a quick overview of the methods provided by our R API. It then continues with a `comprehensive presentation` of all of the available methods, their arguments, and their return values.

If you are new to the Gurobi Optimizer, we suggest that you start with the Getting Started Knowledge Base article or the Example Tour of the reference manual. These documents provide concrete examples of how to use the methods described here.

A quick note for new users: the convention in math programming is that variables are non-negative unless specified otherwise. You'll need to explicitly set lower bounds if you want variables to be able to take negative values.

Models

Our Gurobi R interface enables you to express problems of the following form:

minimize	$x^T Q x + c^T x +$ alpha	
subject to	$Ax = b$	(linear constraints)
	$\ell \leq x \leq u$	(bound constraints)
	some x_j integral	(integrality constraints)
	$x^T Q c\, x + q^T x \leq$ beta	(quadratic constraints)
	some x_i in SOS	(special ordered set constraints)
	min, max, abs, or, ...	(general constraints)

Models are stored as `list` variables, each consisting of multiple *named components*. The named components capture the different model components listed above. Many of these model components are optional. For example, integrality constraints may be omitted.

An optimization model may be loaded from a file (using the `gurobi_read` function), or it can be built by populating the appropriate named components of a model variable (using standard R constructs). We will discuss the details of how models are represented in the `model` argument section.

We often refer to the *class* of an optimization model. At the highest level, a model can be continuous or discrete, depending on whether the modeling elements present in the model require discrete decisions to be made. Among continuous models...

- A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*.
- If the objective is quadratic, the model is a *Quadratic Program (QP)*.

- If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We sometimes refer to a few special cases of QCP: QCPs with convex constraints, QCPs with non-convex constraints, *bilinear programs*, and *Second-Order Cone Programs (SOCP)*.
- If any of the constraints are non-linear (chosen from among the available general constraints), the model is a *Non-Linear Program (NLP)*.

A model that contains any integer variables, semi-continuous variables, semi-integer variables, Special Ordered Set (SOS) constraints, or general constraints, is discrete, and is referred to as a *Mixed Integer Program (MIP)*. The special cases of MIP, which are the discrete versions of the continuous models types we've already described, are...

- *Mixed Integer Linear Programs (MILP)*
- *Mixed Integer Quadratic Programs (MIQP)*
- *Mixed Integer Quadratically-Constrained Programs (MIQCP)*
- *Mixed Integer Second-Order Cone Programs (MISOCP)*
- *Mixed Integer Non-Linear Programs (MINLP)*

The Gurobi Optimizer handles all of these model classes. Note that the boundaries between them aren't as clear as one might like, because we are often able to transform a model from one class to a simpler class.

Solving a Model

Once you have built a model, you can call `gurobi` to compute a solution. By default, `gurobi` will use the concurrent optimizer to solve LP models, the barrier algorithm to solve QP models and QCP models with convex constraints, and the branch-and-cut algorithm to solve mixed integer models. The solution is returned as a `list` variable. We will discuss the details of how optimization results are represented when we discuss the `gurobi` function.
Here is a simple example of a likely sequence of commands in the R API:

```
model <- gurobi_read('examples/data/stein9.mps')
result <- gurobi(model)
```

Multiple Solutions and Multiple Objectives

By default, the Gurobi Optimizer assumes that your goal is to find one proven optimal solution to a model with a single objective function. Gurobi provides features that allow you to relax either of these assumptions. You should refer to the section on Solution Pools for information on how to request more than one solution, or the section on Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call `gurobi_iis` to compute an Irreducible

Inconsistent Subsystem (IIS). This method can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. We will discuss the details of how IIS results are represented in the `gurobi_iis` function documentation.

To attempt to repair an infeasibility, call `gurobi_feasrelax` to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Managing Parameters

The Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization.

Each Gurobi parameter has a default value. Desired parameter changes are passed in a `list` variable. The name of each named component within this list must be the name of a Gurobi parameter, and the associated value should be the desired value of that parameter. You can find a complete list of the available Gurobi parameters in the reference manual. We will provide additional details on changing parameter settings in the `params` argument section.

Monitoring Progress

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging behavior. If you would like to direct output to a file as well as to the screen, set the `LogFile` parameter to the name of your desired log file. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter. A detailed description of the Gurobi log file can be found in the Logging section of the reference manual.

Error Handling

If unsuccessful, the methods of the Gurobi R interface will display an error code and an error message. A list of possible error codes can be found in the Error Code section in the reference manual.

Environments and license parameters

By default, the various Gurobi functions will look for a valid license file and create a local Gurobi environment. This environment exists for as long as the corresponding R API function is running, and is released upon completion.

Another option is to provide licensing parameters through an optional `params` argument (also through a `list`). This argument allows you to solve the given problem on a Gurobi Compute Server, on Gurobi Instant Cloud, or using a Gurobi Cluster Manager. We will discuss this topic further in the `params` argument section.

Gurobi will check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

9.1 Common Arguments

Most common arguments in the Gurobi R interface are R `list` variables, each containing multiple `named components`. Several of these named components are optional. Note that you refer to a named component of a list variable by adding a dollar sign to the end of the variable name, followed by the name of the named component. For example, `model$A` refers to named component `A` of variable `model`.

The model argument

Model variables store optimization problems (as described in the `problem` statement).

Models can be built in a number of ways. You can populate the appropriate named components of the `model` list using standard R routines. You can also read a model from a file, using `gurobi_read`. A few API functions (`gurobi_feasrelax` and `gurobi_relax`) also return models.

Note that all matrix named components within the `model` variable can be dense or sparse. Sparse matrices should be built using either `sparseMatrix` from the `Matrix` package, or `simple_triplet_matrix` from the `slam` package.

The following is an enumeration of all of the named components of the `model` argument that Gurobi will take into account when optimizing the model:

Commonly used named components

`A`: The linear constraint matrix.

`obj (optional)`: The linear objective vector (the `c` vector in the `problem` statement). When present, you must specify one value for each column of `A`. When absent, each variable has a default objective coefficient of 0.

`sense (optional)`: The senses of the linear constraints. Allowed values are `=`, `<`, or `>`. You must specify one value for each row of `A`, or a single value to specify that all constraints have the same sense. When absent, all senses default to `<`.

`rhs (optional)`: The right-hand side vector for the linear constraints (b in the `problem` statement). You must specify one value for each row of `A`. When absent, the right-hand side vector defaults to the zero vector.

`lb (optional)`: The lower bound vector. When present, you must specify one value for each column of `A`. When absent, each variable has a default lower bound of 0.

`ub (optional)`: The upper bound vector. When present, you must specify one value for each column of `A`. When absent, the variables have infinite upper bounds.

`vtype (optional)`: The variable types. This vector is used to capture variable integrality constraints. Allowed values are `C` (continuous), `B` (binary), `I` (integer), `S` (semi-continuous), or `N` (semi-integer). Binary variables must be either 0 or 1. Integer variables can take any integer value between the specified lower and upper bounds. Semi-continuous variables can take any value between the specified lower and upper bounds, or a value of zero. Semi-integer variables can take any integer value between the specified lower and upper bounds, or a value of zero. When present, you must specify one value for each column of `A`, or a single value to specify that all variables have the same type. When absent, each variable is treated as being

continuous. Refer to the variable section of the reference manual for more information on variable types.

`modelsense (optional)`: The optimization sense. Allowed values are `min` (minimize) or `max` (maximize). When absent, the default optimization sense is minimization.

`modelname (optional)`: The name of the model. The name appears in the Gurobi log, and when writing a model to a file.

`objcon (optional)`: The constant offset in the objective function (alpha in the `problem` statement).

`varnames (optional)`: The variable names vector. A character vector. When present, each element of this vector defines the name of a variable. You must specify a name for each column of `A`.

`constrnames (optional)`: The constraint names vector. A character vector. When present, each element of the vector defines the name of a constraint. You must specify a name for each row of `A`.

Quadratic objective and constraint named components

`Q (optional)`: The quadratic objective matrix. When present, `Q` must be a square matrix whose row and column counts are equal to the number of columns in `A`.

`quadcon (optional)`: The quadratic constraints. A list of lists. When present, each element in `quadcon` defines a single quadratic constraint: $x^T Qc\, x + q^T x \leq$ beta.

The `Qc` matrix must be a square matrix whose row and column counts are equal to the number of columns of `A`. It is stored in `model$quadcon[[i]]$Qc`.

The optional `q` vector defines the linear terms in the constraint. It can be a dense vector specifying a value for each column of `A` or a sparse vector (should be built using `sparseVector` from the `Matrix` package). It is stored in `model$quadcon[[i]]$q`.

The scalar `beta` is stored in `model$quadcon[[i]]$rhs`. It defines the right-hand side value for the constraint.

The optional `sense` string defines the sense of the quadratic constraint. Allowed values are `<`, `=` or `>`. If not present, the default sense is `<`. It is stored in `model$quadcon[[i]]$sense`.

The optional `name` string defines the name of the quadratic constraint. It is stored in `model$quadcon[[i]]$name`.

SOS constraint named components

`sos (optional)`: The Special Ordered Set (SOS) constraints. A list of lists. When present, each entry in `sos` defines a single SOS constraint. A SOS constraint can be of type 1 or 2. The type of SOS constraint i is specified via `model$sos[[i]]$type`. A type 1 SOS constraint is a set of variables where at most one variable in the set may take a value other than zero. A type 2 SOS constraint is an ordered set of variables where at most two variables in the set may take non-zero values. If two take non-zeros values, they must be contiguous in the ordered set. The members of an SOS constraint are specified by placing their indices in

vector `model$sos[[i]]$index`. Weights associated with SOS members are provided in vector `model$sos[[i]]$weight`. Please refer to SOS Constraints section in the reference manual for details on SOS constraints.

Multi-objective named components

`multiobj (optional)`: Multi-objective specification for the model. A list of lists. When present, each entry in `multiobj` defines a single objective of a multi-objective problem. Please refer to the Multiple Objectives section in the reference manual for more details on multi-objective optimization. Each objective i may have the following named components:

`objn`: Specified via `model$multiobj[[i]]$objn`. This is the i-th objective vector.

`objcon (optional)`: Specified via `model$multiobj[[i]]$objcon`. If provided, this is the i-th objective constant. The default value is 0.

`priority (optional)`: Specified via `model$multiobj[[i]]$priority`. If provided, this value is the *hierarchical* priority for this objective. The default value is 0.

`weight (optional)`: Specified via `model$multiobj[[i]]$weight`. If provided, this value is the multiplier used when aggregating objectives. The default value is 1.0.

`reltol (optional)`: Specified via `model$multiobj[[i]]$reltol`. If provided, this value specifies the relative objective degradation when doing hierarchical multi-objective optimization. The default value is 0.

`abstol (optional)`: Specified via `model$multiobj[[i]]$abstol`. If provided, this value specifies the absolute objective degradation when doing hierarchical multi-objective optimization. The default value is 0.

`name (optional)`: Specified via `model$multiobj[[i]]$name`. If provided, this string specifies the name of the i-th objective function.

Note that when multiple objectives are present, the `result$objval` named component that is returned in the result of an optimization call will be a vector of the same length as `model$multiobj`.

A multi-objective model can't have other objectives. Thus, combining `model$multiobj` with any of `model$obj`, `model$objcon`, `model$pwlobj`, or `model$Q` is an error.

Computing an IIS

When computing an Irreducible Inconsistent Subsystem (IIS) for an infeasible model, additional model attributes for variable bounds, linear constraints, quadratic constraints and general constraints may be set in order to indicate whether the corresponding entity should be explicitly included or excluded from the IIS:

`iislbforce (optional)`: list of length equal to the number of variables. The value of `model$iislbforce[[i]]` specifies the IIS force attribute for the lower bound of the i-th variable.

`iisubforce (optional)`: list of length equal to the number of variables. The value of `model$iisubforce[[i]]` specifies the IIS force attribute for the upper bound of the i-th variable.

`iisconstrforce (optional)`: list of length equal to the number of constraints. The value of `model$iisconstrforce[[i]]` specifies the IIS force attribute for the i-th constraint.

`iisqconstrforce (optional)`: list of length equal to the number of quadratic constraints. The value of `model$iisqconstrforce[[i]]` specifies the IIS force attribute for the i-th quadratic constraint.

`iisgenconstrforce (optional)`: list of length equal to the number of general constraints. The value of `model$iisgenconstrforce[[i]]` specifies the IIS force attribute for the i-th general constraint.

Possible values for all five attribute of listsfrom above are: -1 to let the algorithm decide, 0 to exclude the corresponding entity from the IIS, and 1 to always include the corresponding entity in the IIS.

Note that setting this attribute to 0 may make the resulting subsystem feasible (or consistent), which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

General constraint named components

The list of lists described below are used to add *general constraints* to a model.

Mathematical programming has traditionally defined a set of fundamental constraint types: variable bound constraints, linear constraints, quadratic constraints, integrality constraints, and SOS constraints. These are typically treated directly by the underlying solver (although not always), and are fundamental to the overall algorithm.

Gurobi accepts a number of additional constraint types, which we collectively refer to as *general (function) constraints*. These are typically *not* treated directly by the solver. Rather, they are transformed by presolve into constraints (and variables) chosen from among the fundamental types listed above. In some cases, the resulting constraint or constraints are mathematically equivalent to the original; in others, they are approximations. If such constraints appear in your model, but if you prefer to reformulate them yourself using fundamental constraint types instead, you can certainly do so. However, note that Gurobi can sometimes exploit information contained in the other constraints in the model to build a more efficient formulation than what you might create.

The additional constraint types that fall under this *general constraint* umbrella are:

- MAX (genconmax): set a decision variable equal to the maximum value from among a set of decision variables
- MIN (genconmin): set a decision variable equal to the minimum value from among a set of decision variables
- ABS (genconabs): set a decision variable equal to the absolute value of some other decision variable
- AND (genconand): set a binary variable equal to one if and only if all of a set of binary decision variables are equal to one

- OR (genconor): set a binary variable equal to one if and only if at least one variable out of a set of binary decision variables is equal to one
- NORM (genconnorm): set a decision variable equal to the p-norm of a vector of decision variables
- INDICATOR (genconind): whenever a given binary variable takes a certain value, then the given linear constraint must be satisfied
- Piecewise-linear constraints (genconpwl): set a variable equal to the piecewise-linear function defined by a set of points using some other variable
- Polynomial (genconpoly): set a variable equal to the polynomial function defined by some other variable
- Natural exponential (genconexp): set a variable equal to the natural exponential function by some other variable
- Exponential (genconexpa): set a variable equal to the exponential function by some other variable
- Natural logarithm (genconlog): set a variable equal to the natural logarithmic function by some other variable
- Logarithm (genconloga): set a variable equal to the logarithmic function by some other variable
- Logistic (genconlogistic) set a variable equal to the logistic function by some other variable
- Power (genconpow): set a variable equal to the power function by some other variable
- SIN (genconsin): set a variable equal to the sine function by some other variable
- COS (genconcos): set a variable equal to the cosine function by some other variable
- TAN (gencontan): set a variable equal to the tangent function by some other variable

Please refer to General Constraints section in the reference manual for additional details on general constraints.

`genconmax (optional)`: A list of lists. When present, each entry in `genconmax` defines a MAX general constraint of the form

$$x[\text{resvar}] = \max\{\text{con}, x[j] : j \in \text{vars}\}$$

Each entry may have the following named components:

`resvar`: Specified via `model$genconmax[[i]]$resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model$genconmax[[i]]$vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`con (optional)`: Specified via `model$genconmax[[i]]$con`. When present, specifies the constant on the left-hand side. Default value is $-\infty$.

`name (optional)`: Specified via `model$genconmax[[i]]$name`. When present, specifies the name of the i-th MAX general constraint.

`genconmin (optional)`: A list of lists. When present, each entry in `genconmax` defines a MIN general constraint of the form

$$x[\text{resvar}] = \min\{\text{con}, x[j] : j \in \text{vars}\}$$

Each entry may have the following named components:

`resvar`: Specified via `model$genconmin[[i]]$resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model$genconmin[[i]]$vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`con (optional)`: Specified via `model$genconmin[[i]]$con`. When present, specifies the constant on the left-hand side. Default value is ∞.

`name (optional)`: Specified via `model$genconmin[[i]]$name`. When present, specifies the name of the i-th MIN general constraint.

`genconabs (optional)`: A list of lists. When present, each entry in `genconmax` defines an ABS general constraint of the form

$$x[\text{resvar}] = |x[\text{argvar}]|$$

Each entry may have the following named components:

`resvar`: Specified via `model$genconabs[[i]]$resvar`. Index of the variable in the left-hand side of the constraint.

`argvar`: Specified via `model$genconabs[[i]]$argvar`. Index of the variable in the right-hand side of the constraint.

`name (optional)`: Specified via `model$genconabs[[i]]$name`. When present, specifies the name of the i-th ABS general constraint.

`genconand (optional)`: A list of lists. When present, each entry in `genconand` defines an AND general constraint of the form

$$x[\text{resvar}] = \text{and}\{x[i] : i \in \text{vars}\}$$

Each entry may have the following named components:

`resvar`: Specified via `model$genconand[[i]]$resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model$genconand[[i]]$vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`name (optional)`: Specified via `model$genconand[[i]]$name`. When present, specifies the name of the i-th AND general constraint.

`genconor (optional)`: A list of lists. When present, each entry in `genconor` defines an OR general constraint of the form

$$x[\text{resvar}] = \text{or}\{x[i] : i \in \text{vars}\}$$

Each entry may have the following named components:

`resvar`: Specified via `model$genconor[[i]]$resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model$genconor[[i]]$vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`name (optional)`: Specified via `model$genconor[[i]]$name`. When present, specifies the name of the i-th OR general constraint.

`genconnorm (optional)`: A list of lists. When present, each entry in `genconnorm` defines a NORM general constraint of the form

$$x[\text{resvar}] = \text{norm}(x[i] : i \in \text{vars}, \text{which})$$

Each entry may have the following named components:

`resvar`: Specified via `model$genconnorm[[i]]$resvar`. Index of the variable in the left-hand side of the constraint.

`vars`: Specified via `model$genconnorm[[i]]$vars`, it is a vector of indices of variables in the right-hand side of the constraint.

`which`: Specified via `model$genconnorm[[i]]$which`. Specifies which p-norm to use. Possible values are 0, 1, 2 and ∞.

`name (optional)`: Specified via `model$genconnorm[[i]]$name`. When present, specifies the name of the i-th NORM general constraint.

`genconind (optional)`: A list of lists. When present, each entry in `genconind` defines an INDICATOR general constraint of the form

$$x[\text{binvar}] = \text{binval} \Rightarrow \sum (x[j] \cdot \text{a}[j]) \ \text{ sense rhs}$$

This constraint states that when the binary variable $x[\text{binvar}]$ takes the value `binval` then the linear constraint $\sum (x[\text{vars}[j]] \cdot \text{val}[j])$ sense rhs must hold. Note that `sense` is one of `=`, `<`, or `>` for equality ($=$), less than or equal ($\leq$) or greater than or equal ($\geq$) constraints. Each entry may have the following named components:

`binvar`: Specified via `model$genconind[[i]]$binvar`. Index of the implicating binary variable.

`binval`: Specified via `model$genconind[[i]]$binval`. Value for the binary variable that forces the following linear constraint to be satisfied. It can be either 0 or 1.

`a`: Specified via `model$genconind[[i]]$a`. Vector of coefficients of variables participating in the implied linear constraint. You can specify a value for `a` for each column of `A` (dense vector) or pass a sparse vector (should be built using `sparseVector` from the `Matrix` package).

`sense`: Specified via `model$genconind[[i]]$sense`. Sense of the implied linear constraint. Must be one of `=`, `<`, or `>`.

`rhs`: Specified via `model$genconind[[i]]$rhs`. Right-hand side value of the implied linear constraint.

`name (optional)`: Specified via `model$genconind[[i]]$name`. When present, specifies the name of the i-th INDICATOR general constraint.

`genconpwl (optional)`: A list of lists. When present, each entry in `genconpwl` defines a piecewise-linear constraint of the form

$$x[\text{yvar}] = f(x[\text{xvar}])$$

The breakpoints for f are provided as arguments. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined

Each entry may have the following named components:

`xvar`: Specified via `model$genconpwl[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconpwl[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`xpts`: Specified via `model$genconpwl[[i]]$xpts`. Specifies the x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

`ypts`: Specified via `model$genconpwl[[i]]$ypts`. Specifies the y values for the points that define the piecewise-linear function.

`name (optional)`: Specified via `model$genconpwl[[i]]$name`. When present, specifies the name of the i-th piecewise-linear general constraint.

`genconpoly (optional)`: A list of lists. When present, each entry in `genconpoly` defines a polynomial function constraint of the form

$$x[\text{yvar}] = p_0 x[\text{xvar}]^d + p_1 x[\text{xvar}]^{d-1} + ... + p_{d-1} x[\text{xvar}] + p_d$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconpoly[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconpoly[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`p`: Specified via `model$genconpoly[[i]]$p`. Specifies the coefficients for the polynomial function (starting with the coefficient for the highest power). If x^d is the highest power term, a dense vector of length $d + 1$ is returned.

`name (optional)`: Specified via `model$genconpoly[[i]]$name`. When present, specifies the name of the i-th polynomial function constraint.

`funcpieces (optional)`: Specified via `model$genconpoly[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th polynomial function constraint.

`funcpiecelength (optional)`: Specified via `model$genconpoly[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th polynomial function constraint.

`funcpieceerror (optional)`: Specified via `model$genconpoly[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th polynomial function constraint.

`funcpieceratio (optional)`: Specified via `model$genconpoly[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th polynomial function constraint.

`funcnonlinear (optional)`: Specified via `model$genconpoly[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th polynomial function constraint.

`genconexp (optional)`: A list of lists. When present, each entry in `genconexp` defines the natural exponential function constraint of the form

$$x[\text{yvar}] = \exp(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconexp[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconexp[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model$genconexp[[i]]$name`. When present, specifies the name of the i-th natural exponential function constraint.

`funcpieces (optional)`: Specified via `model$genconexp[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th natural exponential function constraint.

`funcpiecelength (optional)`: Specified via `model$genconexp[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th natural exponential function constraint.

`funcpieceerror (optional)`: Specified via `model$genconexp[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th natural exponential function constraint.

`funcpieceratio (optional)`: Specified via `model$genconexp[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th natural exponential function constraint.

`funcnonlinear (optional)`: Specified via `model$genconexp[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th natural exponential function constraint.

`genconexpa (optional)`: A list of lists. When present, each entry in `genconexpa` defines an exponential function constraint of the form

$$x[\text{yvar}] = \text{a}^{x[\text{xvar}]}$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconexpa[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconexpa[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`a`: Specified via `model$genconexpa[[i]]$a`. Specifies the base of the exponential function $a > 0$.

`name (optional)`: Specified via `model$genconexpa[[i]]$name`. When present, specifies the name of the i-th exponential function constraint.

`funcpieces (optional)`: Specified via `model$genconexpa[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th exponential function constraint.

`funcpiecelength (optional)`: Specified via `model$genconexpa[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th exponential function constraint.

`funcpieceerror (optional)`: Specified via `model$genconexpa[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th exponential function constraint.

`funcpieceratio (optional)`: Specified via `model$genconexpa[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th exponential function constraint.

`funcnonlinear (optional)`: Specified via `model$genconexpa[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th exponential function constraint.

`genconlog (optional)`: A list of lists. When present, each entry in `genconlog` defines the natural logarithmic function constraint of the form

$$x[\text{yvar}] = \log(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconlog[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconlog[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model$genconlog[[i]]$name`. When present, specifies the name of the i-th natural logarithmic function constraint.

`funcpieces (optional)`: Specified via `model$genconlog[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th natural logarithmic function constraint.

`funcpiecelength (optional)`: Specified via `model$genconlog[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th natural logarithmic function constraint.

`funcpieceerror (optional)`: Specified via `model$genconlog[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th natural logarithmic function constraint.

`funcpieceratio (optional)`: Specified via `model$genconlog[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th natural logarithmic function constraint.

`funcnonlinear (optional)`: Specified via `model$genconlog[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th natural logarithmic function constraint.

`genconloga (optional)`: A list of lists. When present, each entry in `genconloga` defines a logarithmic function constraint of the form

$$x[\text{yvar}] = \log(x[\text{xvar}]) \setminus \log(a)$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconloga[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconloga[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`a`: Specified via `model$genconloga[[i]]$a`. Specifies the base of the logarithmic function $a > 0$.

`name (optional)`: Specified via `model$genconloga[[i]]$name`. When present, specifies the name of the i-th logarithmic function constraint.

`funcpieces (optional)`: Specified via `model$genconloga[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th logarithmic function constraint.

`funcpiecelength (optional)`: Specified via `model$genconloga[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th logarithmic function constraint.

`funcpieceerror (optional)`: Specified via `model$genconloga[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th logarithmic function constraint.

`funcpieceratio (optional)`: Specified via `model$genconloga[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th logarithmic function constraint.

`funcnonlinear (optional)`: Specified via `model$genconloga[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th logarithmic function constraint.

`genconlogistic (optional)`: A list of lists. When present, each entry in `genconlog` defines the logistic function constraint of the form

$$x[\text{yvar}] = 1/(1 + \exp(-x[\text{xvar}]))$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconlogistic[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconlogistic[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model$genconlogistic[[i]]$name`. When present, specifies the name of the i-th logistic function constraint.

`funcpieces (optional)`: Specified via `model$genconlogistic[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th logistic function constraint.

`funcpiecelength (optional)`: Specified via `model$genconlogistic[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th logistic function constraint.

`funcpieceerror (optional)`: Specified via `model$genconlogistic[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th logistic function constraint.

`funcpieceratio (optional)`: Specified via `model$genconlogistic[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th logistic function constraint.

`funcnonlinear (optional)`: Specified via `model$genconlogistic[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th logistic function constraint.

`genconpow (optional)`: A list of lists. When present, each entry in `genconpow` defines a power function constraint of the form

$$x[\text{yvar}] = x[\text{xvar}]^{\text{a}}$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconpow[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconpow[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`a`: Specified via `model$genconpow[[i]]$a`. Specifies the exponent of the power function.

`name (optional)`: Specified via `model$genconpow[[i]]$name`. When present, specifies the name of the i-th power function constraint.

`funcpieces (optional)`: Specified via `model$genconpow[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th power function constraint.

`funcpiecelength (optional)`: Specified via `model$genconpow[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th power function constraint.

`funcpieceerror (optional)`: Specified via `model$genconpow[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th power function constraint.

`funcpieceratio (optional)`: Specified via `model$genconpow[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th power function constraint.

`funcnonlinear (optional)`: Specified via `model$genconpow[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th power function constraint.

`genconsin (optional)`: A list of lists. When present, each entry in `genconsin` defines the sine function constraint of the form

$$x[\text{yvar}] = \sin(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconsin[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconsin[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model$genconsin[[i]]$name`. When present, specifies the name of the i-th sine function constraint.

`funcpieces (optional)`: Specified via `model$genconsin[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th sine function constraint.

`funcpiecelength (optional)`: Specified via `model$genconsin[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th sine function constraint.

`funcpieceerror (optional)`: Specified via `model$genconsin[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th sine function constraint.

`funcpieceratio (optional)`: Specified via `model$genconsin[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th sine function constraint.

`funcnonlinear (optional)`: Specified via `model$genconsin[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th sine function constraint.

`genconcos (optional)`: A list of lists. When present, each entry in `genconcos` defines the cosine function constraint of the form

$$x[\text{yvar}] = \cos(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$genconcos[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$genconcos[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model$genconcos[[i]]$name`. When present, specifies the name of the i-th cosine function constraint.

`funcpieces (optional)`: Specified via `model$genconcos[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th cosine function constraint.

`funcpiecelength (optional)`: Specified via `model$genconcos[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th cosine function constraint.

`funcpieceerror (optional)`: Specified via `model$genconcos[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th cosine function constraint.

`funcpieceratio (optional)`: Specified via `model$genconcos[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th cosine function constraint.

`funcnonlinear (optional)`: Specified via `model$genconcos[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th cosine function constraint.

`gencontan (optional)`: A list of lists. When present, each entry in `gencontan` defines the tangent function constraint of the form

$$x[\text{yvar}] = \tan(x[\text{xvar}])$$

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPieceLength, and FuncPieceRatio. Alternatively, the function can be treated as a nonlinear constraint by setting the attribute FuncNonlinear. For details, consult the General Constraint discussion.

Each entry may have the following named components:

`xvar`: Specified via `model$gencontan[[i]]$xvar`. Index of the variable in the right-hand side of the constraint.

`yvar`: Specified via `model$gencontan[[i]]$yvar`. Index of the variable in the left-hand side of the constraint.

`name (optional)`: Specified via `model$gencontan[[i]]$name`. When present, specifies the name of the i-th tangent function constraint.

`funcpieces (optional)`: Specified via `model$gencontan[[i]]$funcpieces`. When present, specifies the FuncPieces attribute of the i-th tangent function constraint.

`funcpiecelength (optional)`: Specified via `model$gencontan[[i]]$funcpiecelength`. When present, specifies the FuncPieceLength attribute of the i-th tangent function constraint.

`funcpieceerror (optional)`: Specified via `model$gencontan[[i]]$funcpieceerror`. When present, specifies the FuncPieceError attribute of the i-th tangent function constraint.

`funcpieceratio (optional)`: Specified via `model$gencontan[[i]]$funcpieceratio`. When present, specifies the FuncPieceRatio attribute of the i-th tangent function constraint.

`funcnonlinear (optional)`: Specified via `model$gencontan[[i]]$funcnonlinear`. When present, specifies the FuncNonlinear attribute of the i-th tangent function constraint.

Advanced named components

`pwlobj (optional)`: The piecewise-linear objective functions. A list of lists. When present, each entry in `pwlobj` defines a piecewise-linear objective function for a single variable. The index of the variable whose objective function is being defined is stored in `model$pwlobj[[i]]$var`. The x values for the points that define the piecewise-linear function are stored in `model$pwlobj[[i]]$x`. The values in the x vector must be in non-decreasing order. The y values for the points that define the piecewise-linear function are stored in `model$pwlobj[[i]]$y`.

`vbasis (optional)`: The variable basis status vector. Used to provide an advanced starting point for the simplex algorithm. You would generally never concern yourself with the contents of this vector, but would instead simply pass it from the result of a previous optimization run to the input of a subsequent run. When present, you must specify one value for each column of `A`.

`cbasis (optional)`: The constraint basis status vector. Used to provide an advanced starting point for the simplex algorithm. Consult the `vbasis` description for details. When present, you must specify one value for each row of `A`.

`varhintval (optional)`: A set of user hints. If you know that a variable is likely to take a particular value in high quality solutions of a MIP model, you can provide that value as a hint. You can also (optionally) provide information about your level of confidence in a hint with the `varhintpri` named component. If present, you must specify one value for each column of `A`. Use a value of `NA` for variables where no such hint is known. For more details, please refer to the Attribute section in the reference manual.

`varhintpri (optional)`: Priorities on user hints. After providing variable hints through the `varhintval` list, you can optionally also provide hint priorities to give an indication of your level of confidence in your hints. If present, you must specify a value for each column of `A`. For more details, please refer to the Attribute section in the reference manual.

`branchpriority (optional)`: Variable branching priority. If present, the value of this attribute is used as the primary criteria for selecting a fractional variable for branching during the MIP search. Variables with larger values always take priority over those with smaller values. Ties are broken using the standard branch variable selection criteria. If present, you must specify one value for each column of `A`.

`pstart (optional)`: The current simplex start vector. If you set `pstart` values for every variable in the model and `dstart` values for every constraint, then simplex will use those values to compute a warm start basis. For more details, please refer to the Attribute section in the reference manual.

`dstart (optional)`: The current simplex start vector. If you set `dstart` values for every linear constraint in the model and `pstart` values for every variable, then simplex will use those values to compute a warm start basis. For more details, please refer to the Attribute section in the reference manual.

`lazy (optional)`: Determines whether a linear constraint is treated as a *lazy constraint*. If present, you must specify one value for each row of `A`. For more details, please refer to the Attribute section in the reference manual.

`start (optional)`: The MIP start vector. The MIP solver will attempt to build an initial solution from this vector. When present, you must specify a start value for each variable. Note that you can set the start value for a variable to `NA`, which instructs the MIP solver to try to fill in a value for that variable.

`partition (optional)`: The MIP variable partition number, which is used by the MIP solution improvement heuristic. If present, you must specify one value for each variable of `A`. For more details, please refer to the Attribute section in the reference manual.

If any of the mandatory components listed above are missing, the `gurobi()` function will return an error.

Below is an example that demonstrates the construction of a simple optimization model:

```
model <- list()
model$A <- matrix(c(1,2,3,1,1,0), nrow=2, byrow=T)
model$obj <- c(1,1,1)
model$modelsense <- 'max'
model$rhs <- c(4,1)
model$sense <- c('<', '>')
```

You can also build `A` as a sparse matrix, using either `sparseMatrix` or `simple_triplet_matrix`:

```
model$A <- spMatrix(2, 3, c(1, 1, 1, 2, 2), c(1, 2, 3, 1, 2), c(1, 2, 3, 1, 1))
model$A <- simple_triplet_matrix(c(1, 1, 1, 2, 2), c(1, 2, 3, 1, 2), c(1, 2, 3, 1,
1))
```

Note that the Gurobi R interface allows you to specify a scalar value for most of the array-valued components. The specified value will be expanded to an array of the appropriate size, with each component of the array equal to the scalar (e.g., `model$obj <- 1` would be equivalent to `model$obj <- c(1,1,1)` in the example).

The params argument

As mentioned previously, the Gurobi Optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization.

Parameter changes are specified using a `list` variable having multiple `named components`, which is passed as an argument to the appropriate Gurobi function (e.g., `gurobi`). The name of each named component must be the name of a Gurobi parameter, and the associated value should be the desired value of that parameter. You can find a complete list of the available Gurobi parameters in the reference manual.

To create a list that would set the Gurobi `Method` parameter to 2 and the `ResultFile` parameter to `model.mps`, you would do the following:

```
params <- list()
params$Method <- 2
```

```
params$ResultFile <- 'model.mps'
```

We should say a bit more about the `ResultFile` parameter. If this parameter is set, the optimization model that is eventually passed to Gurobi will also be output to the specified file. The filename suffix should be one of `.mps`, `.lp`, `.rew`, `.rlp`, `.dua`, or `.dlp`, to indicate the desired file format (see the file format section in the reference manual for details on Gurobi file formats).

The params struct can also be used to set license specific parameters, that define the computational environment to be used. We will discuss the two most common use cases next, and refer again to the collection of all available parameters in the reference manual.

Using a Compute Server License

Gurobi Compute Server allows you to offload optimization jobs to a remote server. Servers are organized into clusters. By providing the name of any node within the cluster, your job will automatically be sent to the least heavily loaded node in the cluster. If all nodes are at capacity, your job will be placed in a queue, and will proceed once capacity becomes available. You can find additional information about Gurobi Compute Server in the Gurobi Remote Services Reference Manual. The most commonly used parameters are the following.

`ComputeServer`: A Compute Server. You can refer to the server using its name or its IP address. If you are using a non-default port, the server name should be followed by the port number (e.g., `server1:61000`).

`ServerPassword (optional)`: User password on the Compute Server cluster. Obtain this from your Compute Server administrator.

`CSPriority (optional)`: The priority of the job. Priorities must be between -100 and 100, with a default value of 0 (by convention). Higher priority jobs are chosen from the server job queue before lower priority jobs. A job with priority 100 runs immediately, bypassing the job queue and ignoring the job limit on the server. You should exercise caution with priority 100 jobs, since they can severely overload a server, which can cause jobs to fail, and in extreme cases can cause the server to crash.

`CSRouter (optional)`: The router for the Compute Server cluster. A router can be used to improve the robustness of a Compute Server deployment. You can refer to the router using either its name or its IP address. A typical Remote Services deployment won't use a router, so you typically won't need to set this.

`CSTLSinsecure (optional)`: Indicates whether to use insecure mode in the TLS (Transport Layer Security). Leave this at its default value of 0 unless your server administrator tells you otherwise.

Here is an example of how to use a `params` argument to connect to a Compute Server:

```
params <- list()
params$ComputeServer <- 'server1.mycompany.com:61000'
params$CSPriority <- 5
```

Using a Gurobi Instant Cloud License

Gurobi Instant Cloud allows you to offload optimization jobs to a Gurobi Compute Server on the cloud. If an appropriate machine is already running, the job will run on that machine. It will automatically launch a new machine otherwise. Note that launching a new machine can take a few minutes. You can find additional information about the Gurobi Instant Cloud service in the reference manual. The most commonly used parameters are the following.

`CloudAccessID`: The access ID for your Gurobi Instant Cloud license. This can be retrieved from the Gurobi Instant Cloud website. When used in combination with your `CloudSecretKey`, this allows you to launch Instant Cloud instances and submit jobs to them.

`CloudSecretKey`: The secret key for your Gurobi Instant Cloud license. This can be retrieved from the Gurobi Instant Cloud website. When used in combination with your `CloudAccessID`, this allows you to launch Instant Cloud instances and submit jobs to them. Note that you should keep your secret key private.

`CloudPool (optional)`: The machine pool. Machine pools allow you to create fixed configurations on the Instant Cloud website (capturing things like type of machine, geographic region, etc.), and then launch and share machines from client programs without having to restate configuration information each time you launch a machine. If not provided, your job will be launched in the default pool associated with your cloud license.

`CSPriority (optional)`: The priority of the job. Priorities must be between -100 and 100, with a default value of 0 (by convention). Higher priority jobs are chosen from the server job queue before lower priority jobs. A job with priority 100 runs immediately, bypassing the job queue and ignoring the job limit on the server. You should exercise caution with priority 100 jobs, since they can severely overload a server, which can cause jobs to fail, and in extreme cases can cause the server to crash.

Here is an example of how to use a `params` argument to launch a Gurobi Instant Cloud instance:

```
params <- list()
params$CloudAccessID <- '3d1ecef9-dfad-eff4-b3fa'
params$CloudSecretKey <- 'ae6L23alJe3+fas'
```

9.2 Solving a Model

gurobi()

```
gurobi  ( model, params=NULL )
```

This function optimizes the given model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion it will return a `list` variable containing solution information.

Please consult Variables and Constraints section in the reference manual for a discussion of some of the practical issues associated with solving a precisely defined mathematical model using finite-precision floating-point arithmetic.

Arguments:

model: The model list must contain a valid Gurobi model. See the model argument section for more information.

params: The params list, when provided, contains a list of modified Gurobi parameters. See the params argument section for more information.

Example usage:

```
result <- gurobi(model, params)
if (result$status == 'OPTIMAL') {
  print(result$objval)
  print(result$x)
} else {
  cat('Optimization returned status:', formatC(result$status), '\n')
}
```

Return value:

The optimization result

The gurobi function returns a list, with the various results of the optimization stored in its named components. The specific results that are available depend on the type of model that was solved, the parameters used, and the status of the optimization. The following is a list of named components that might be available in the returned result. We will discuss the circumstances under which each will be available after presenting the list.

Model named components

status: The status of the optimization, returned as a string. The desired result is OPTIMAL, which indicates that an optimal solution to the model was found. Other status are possible, for example if the model has no feasible solution or if you set a Gurobi parameter that leads to early solver termination. See the Status Code section for further information on the Gurobi status codes.

objval: The objective value of the computed solution. Note that for multi-objective models result$objval will be a vector, where result$objval[[i]] stores the value for model$multiobj[[i]].

objbound: Best available bound on solution (lower bound for minimization, upper bound for maximization).

objboundc: The best unrounded bound on the optimal objective. In contrast to objbound, this attribute does not take advantage of objective integrality information to round to a tighter bound. For example, if the objective is known to take an integral value and the current best bound is 1.5, objbound will return 2.0 while objboundc will return 1.5.

mipgap: Current relative MIP optimality gap; computed as $|ObjBound - ObjVal|/|ObjVal|$ (where $ObjBound$ and $ObjVal$ are the MIP objective bound and incumbent solution objective, respectively). Returns GRB_INFINITY when an incumbent solution has not yet been found,

when no objective bound is available, or when the current incumbent objective is 0. This is only available for mixed-integer problems.

`runtime`: The elapsed wall-clock time (in seconds) for the optimization.

`work`: The work (in work units) spent on the optimization. As opposed to the runtime in seconds, the work is deterministic. This means that on the same hardware and with the same parameter and attribute settings, solving the same model twice will lead to exactly the same amount of work in each of the two solves. One work unit corresponds very roughly to one second, but this greatly depends on the hardware on which Gurobi is running and on the model that has been solved.

`itercount`: Number of simplex iterations performed.

`baritercount`: Number of barrier iterations performed.

`nodecount`: Number of branch-and-cut nodes explored.

`maxvio`: Value of the maximal (unscaled) violation of the returned solution.

`farkasproof`: Magnitude of infeasibility violation in Farkas infeasibility proof. Only available if the model was found to be infeasible. Please refer to Attribute section in the reference manual for details.

Variable named components

`x`: The computed solution. This vector contains one entry for each column of `A`.

`rc`: Variable reduced costs for the computed solution. This vector contains one entry for each column of `A`.

`vbasis`: Variable basis status values for the computed optimal basis. You generally should not concern yourself with the contents of this vector. If you wish to use an advanced start later, you would simply copy the `vbasis` and `cbasis` named components into the corresponding named components for the next model. This vector contains one entry for each column of `A`.

`unbdray`: Unbounded ray. Provides a vector that, when added to any feasible solution, yields a new solution that is also feasible but improves the objective. Only available if the model is found to be unbounded. This vector contains one entry for each column of `A`.

Linear constraint named components

`slack`: The constraint slack for the computed solution. This vector contains one entry for each row of `A`.

`pi`: Dual values for the computed solution (also known as *shadow prices*). This vector contains one entry for each row of `A`.

`cbasis`: Constraint basis status values for the computed optimal basis. This vector contains one entry for each row of `A`.

`farkasdual`: Farkas infeasibility proof. Only available if the model was found to be infeasible. Please refer to Attribute section in the reference manual for details.

Quadratic constraint named components

`qcslack`: The quadratic constraint slack in the current solution. This vector contains one entry for each quadratic constraint.

`qcpi`: The dual values associated with the quadratic constraints. This vector contains one entry for each quadratic constraint.

Solution Pool named components

`pool`: When multiple solutions are found during the optimization call, these solutions are returned in this named component. A list of lists. When present, each list has the following named components:

`objval`: Stores the objective value of the i-th solution in `result$pool[[i]]$objval`. Note that when the model is a multi-objective model, instead of a single value, `result$pool[[i]]$objval[j]` stores the value of the j-th objective function for the i-th solution.

`xn`: Stores the i-th solution in `result$pool[[i]]$xn`. This vector contains one entry for each column of `A`.

Note that to query the number of solutions stored, you can query the length of `result$pool`.

`poolobjbound`: For single-objective MIP optimization problems, this value gives a bound on the best possible objective of an undiscovered solution. The difference between this value and `objbound` is that the former gives an objective bound for undiscovered solutions, while the latter gives a bound for any solution.

What is Available When

The `status` named component will be present in all cases. It indicates whether Gurobi was able to find a proven optimal solution to the model. In cases where a solution to the model was found, optimal or otherwise, the `objval` and `x` named components will be present.

For linear and quadratic programs, if a solution is available, then the `pi` and `rc` named components will also be present. For models with quadratic constraints, if the parameter `qcpdual` is set to 1, the named component `qcpi` will be present. If the final solution is a *basic* solution (computed by simplex), then `vbasis` and `cbasis` will be present. If the model is an unbounded linear program and the InfUnbdInfo parameter is set to 1, the named component `unbdray` will be present. Finally, if the model is an infeasible linear program and the InfUnbdInfo parameter is set to 1, the named components `farkasdual` and `farkasproof` will be set.

For mixed integer problems, no dual information (i.e. `pi`, `slack`, `rc`, `vbasis`, `cbasis`, `qcslack`, `qcpi`, `ubdray` or `farkasdual`) is ever available. When multiple solutions are found, the `pool` and `poolobjbound` named components will be present. Depending on the `status` named component value, the named components `nodecount`, `objbound`, `objbundc` and `mipgap` will be available.

For continuous and mixed-integer models, under normal execution, the named components `runtime`, `work`, `itercount` and `baritercount` will be available.

gurobi_iis()

```
gurobi_iis  ( model, params=NULL )
```

Compute an Irreducible Inconsistent Subsystem (IIS).
An IIS is a subset of the constraints and variable bounds with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the smallest one; there may exist others with fewer constraints or bounds.

You can obtain information about the outcome of the IIS computation from the returned IIS result (described below). Note that this method can be used to compute IISs for both continuous and MIP models.

Arguments:

`model`: The model `list` must contain a valid Gurobi model. See the `model` argument section for more information.

`params`: The params `list`, when provided, contains a list of modified Gurobi parameters. See the `params` argument section for more information.

Example usage:

```
model <- gurobi_read('examples/data/klein1.mps')
iis <- gurobi_iis(model)
```

Return value:

The `gurobi_iis()` function returns a `list`, with various results stored in its named components. The specific results that are available depend on the type of model.

The returned `list` will always contain the following `named components`:

`minimal`: A logical scalar that indicates whether the computed IIS is minimal. It will normally be true, but it may be false if the IIS computation was stopped early (due to a time limit or a user interrupt).

`Arows`: A logical vector that indicates whether a linear constraint appears in the computed IIS.

`lb`: A logical vector that indicates whether a lower bound appears in the computed IIS.

`ub`: A logical vector that indicates whether a upper bound appears in the computed IIS.

If your model contains general constraints, the returned `list` will also contain the following `named components`:

`genconmax`: A logical vector that indicates whether a general MAX constraint appears in the computed IIS.

`genconmin`: A logical vector that indicates whether a general MIN constraint appears in the computed IIS.

`genconand`: A logical vector that indicates whether a general AND constraint appears in the computed IIS.

`genconor`: A logical vector that indicates whether a general OR constraint appears in the computed IIS.

`genconabs`: A logical vector that indicates whether a general ABS constraint appears in the computed IIS.

`genconind`: A logical vector that indicates whether a general INDICATOR constraint appears in the computed IIS.

`genconpwl`: A logical vector that indicates whether a general piecewise-linear function constraint appears in the computed IIS.

`genconpoly`: A logical vector that indicates whether a polynomial function constraint appears in the computed IIS.

`genconexp`: A logical vector that indicates whether a natural exponential function constraint appears in the computed IIS.

`genconexpa`: A logical vector that indicates whether a exponential function constraint appears in the computed IIS.

`genconlog`: A logical vector that indicates whether a natural logarithmic function constraint appears in the computed IIS.

`genconloga`: A logical vector that indicates whether a logarithmic function constraint appears in the computed IIS.

`genconlogistic`: A logical vector that indicates whether a logistic function constraint appears in the computed IIS.

`genconpow`: A logical vector that indicates whether a power function constraint appears in the computed IIS.

`genconsin`: A logical vector that indicates whether a SIN function constraint appears in the computed IIS.

`genconcos`: A logical vector that indicates whether a COS function constraint appears in the computed IIS.

`gencontan`: A logical vector that indicates whether a TAN function constraint appears in the computed IIS.

If your model contains SOS constraints, the returned `list` will also contain the following `named component`:

`sos`: A logical vector that indicates whether an SOS constraint appears in the computed IIS

If your model contains quadratic constraints, the returned `list` will also contain the following `named component`:

`quadcon`: A logical vector that indicates whether a quadratic constraint appears in the computed IIS.

gurobi_feasrelax()

`gurobi_feasrelax ( model, relaxobjtype, minrelax, penalties, params=NULL )`

This function computes a feasibility relaxation for the input `model` argument. The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. You must provide a penalty to associate with relaxing each individual bound or constraint (through the `penalties` argument). These penalties are interpreted in different ways, depending on the value of the `relaxobjtype` argument.

Arguments:

`model`: The model `list` must contain a valid Gurobi model. See the `model` argument section for more information.

`relaxobjtype`: The approach used to impose penalties on violations.

If you specify `relaxobjtype=0`, the objective for the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations.

If you specify `relaxobjtype=1`, the objective for the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations.

If you specify `relaxobjtype=2`, the objective for the feasibility relaxation is to minimize the weighted count of bound and constraint violations.

In all cases, the weights are taken from `penalties$lb`, `penalties$ub` and `penalties$rhs`. You can provide the special penalty value `Inf` to indicate that the corresponding bound or constraint cannot be relaxed.

`minrelax`: The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=FALSE`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=TRUE`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `gurobi_feasrelax` must solve an optimization problem to find the minimum possible relaxation when `minrelax=TRUE`, which can be quite expensive.

`penalties`: The `penalties` argument is a list of lists, having the following optional named components (default: all `Inf`):

`lb` Penalty for violating each lower bound.

`ub` Penalty for violating each upper bound.

`rhs` Penalty for violating each constraint.

To give an example, if a constraint with `penalties.rhs` value `p` is violated by 2.0, it would contribute `2*p` to the feasibility relaxation objective for `relaxobjtype=0`, `2*2*p` for `relaxobjtype=1`, and `p` for `relaxobjtype=2`.

`params`: The params `list`, when provided, contains a list of modified Gurobi parameters. See the `params` argument section for more information.

Return value:

A `list` containing two named components:
`result$model`, a list variable, as described in the `model` argument section.
`result$feasobj`, a scalar. If `minrelax==TRUE` this is the relaxation problem objective value, 0.0 otherwise.

Example usage:

```
penalties <- list()
model <- gurobi_read('stein9.mps')
penalties$lb <- rep(1,length(model$lb))
penalties$ub <- rep(1,length(model$ub))
penalties$rhs <- rep(1,length(model$rhs))
feasrelaxresult <- gurobi_feasrelax(model, 0, FALSE, penalties)
```

gurobi_relax()

```
gurobi_relax  ( model, params=NULL )
```

Create the relaxation of a MIP model. Transforms integer variables into continuous variables, and removes SOS and general constraints.

Arguments:

`model`: The model `list` must contain a valid Gurobi model. See the `model` argument section for more information.

`params`: The params `list`, when provided, contains a list of modified Gurobi parameters. See the `params` argument section for more information.

Return value:

A `model` list variable, as described in the `model` parameter section.

Example usage:

```
model <- gurobi_read('stein9.mps')
relaxed <- gurobi_relax(model)
```

9.3 Input-Output

gurobi_read()

```
gurobi_read  ( filename, params=NULL )
```

Reads a model from a file.

Arguments:

`filename`: Name of the file to read. Note that the type of the file is encoded in the file name suffix. The filename suffix should be one of .mps, .rew, .lp, .rlp, .dua, .dlp, .ilp, or .opb (see the file formats section for details on Gurobi file formats). The files can be compressed, so additional suffixes of .gz, .bz2, .zip, or .7z are accepted.

params: The params `list`, when provided, contains a list of modified Gurobi parameters. See the params argument section for more information.

Return value:

A `model` list variable, as described in the model section.

Example usage:

```
model <- gurobi_read('stein9.mps')
result <- gurobi(model)
```

gurobi_write()

```
gurobi_write  ( model, filename, params=NULL )
```

Writes a model to a file.

Arguments:

model: The model `list` must contain a valid Gurobi model. See the model argument section for more information.

filename: Name of the file to write. Note that the type of the file is encoded in the file name suffix. The filename suffix should be one of .mps, .rew, .lp, .rlp, .dua, .dlp, or .ilp, to indicate the desired file format (see the file formats section for details on Gurobi file formats). The files can be compressed, so additional suffixes of .gz, .bz2, .zip, or .7z are accepted.

params: The params `list`, when provided, contains a list of modified Gurobi parameters. See the params argument section for more information.

Example usage:

```
model            <- list()
model$A          <- matrix(c(1,2,3,1,1,0), nrow=2, byrow=T)
model$obj        <- c(1,1,2)
model$modelsense <- 'max'
model$rhs        <- c(4,1)
model$sense      <- c('<', '>')
gurobi_write(model, 'mymodel.mps');
gurobi_write(model, 'mymodel.lp');
gurobi_write(model, 'mymodel.mps.bz2');
```

9.4 Installing the R package

To use our R interface, you'll need to install the Gurobi package in your local R installation. The R command for doing this is:

```
install.packages('<R-package-file>', repos=NULL)
```

The Gurobi R package file can be found in the `<installdir>/R` directory of your Gurobi installation (the default `<installdir>` for Gurobi 11.0.0 is `/opt/gurobi1100/linux64` for Linux, `c:\gurobi1100\win64` for 64-bit Windows, and `/Library/gurobi1100/macos_universal2` for Mac). You should browse the `<installdir>/R` directory to find the exact name of the file for

your platform (the Linux package is in file `gurobi_11.0-0_R_4.3.2.tar.gz`, the Windows package is in file `gurobi_11.0-0.zip`, and the Mac package is in file `gurobi_11.0-0_R_4.3.2.tgz`).

Depending on your local R environment you might need to install the R package `slam`. To do this, you should issue the following command within R:

```
install.packages('slam')
```

You will need to be careful to make sure that the R binary and the Gurobi package you install both use the same instruction set. For example, if you are using the 64-bit version of R, you'll need to install the 64-bit version of Gurobi, and the 64-bit Gurobi R package. This is particularly important on Windows systems, where the error messages that result from instruction set mismatches can be quite cryptic.

To run one of the R examples provided with the Gurobi distribution, you can use the `source` command in R. For example, if you are running R from the Gurobi R examples directory, you can say:

```
> source('mip.R')
```

If the Gurobi package was successfully installed, you should see the following output:

```
[1] 'Solution:'
[1] 3
[1] 1 0 1
```

The lowest-level building blocks for Gurobi models are variables, constraints, and objectives. While each has a clean mathematical definition, linear and integer programming aren't performed in exact arithmetic, so computed results can sometimes deviate from these clean definitions. This section discusses the use of and restrictions on these basic building blocks.

10.1 Variables

Decision variables capture the results of the optimization. In a feasible solution, the computed values for the decision variables satisfy all of the model constraints. Some of these constraints are associated with individual variables (e.g., variable bounds), while others capture relationships between variables. We'll first consider the different types of decision variables that can be added to a Gurobi model, and the implicit and explicit constraints associated with these variable types.

Before starting, we should point out one important thing about the variables in a mathematical programming model: their computed solution values will only satisfy bounds *to tolerances*, meaning that a variable may violate its stated bounds. Mathematical programming is fundamentally built on top of linear algebra and in particular on the numerical solution of systems of linear equations. These linear systems are solved using finite-precision arithmetic, which means that small errors are unavoidable. For some models, large errors are unavoidable too; we'll return to that topic later in this section.

The available variables types are continuous, general integer, binary, semi-continuous, and semi-integer.

Continuous Variables

The simplest and least constrained of the available variable types is the continuous variable. This variable can take any value between its lower and upper bound. In mathematical programming, the convention is that variables are non-negative unless stated otherwise, so if you don't explicitly provide bounds for a variable, you should assume that the lower bound is 0 and the upper bound is infinite.

The Gurobi APIs provides a symbolic constant to allow you to indicate that a bound is infinite (`GRB_INFINITY` in C and C++, `GRB.INFINITY` in C#, Java, and Python). A variable can have an infinite upper bound, an infinite lower bound (negative infinity), or both. A variable with infinite upper and lower bounds is referred to as a *free variable*. Any bound larger than 1e30 is treated as infinite.

As noted earlier, variables may violate their bounds by tolerances. In the case of variable bounds, the relevant tolerance value is the FeasibilityTol. You can reduce the value of this tolerance parameter, but due to numerical errors it may not be possible to achieve your desired accuracy.

General Integer Variables

General integer variables are more constrained than continuous variables. In addition to respecting the specified lower and upper bounds, integer variables also take integral values.

Due to the limitations of finite-precision arithmetic, integer variables will often take values that aren't exactly integral. The magnitude of the allowed integrality violation is controlled by the IntFeasTol parameter. You can tighten this parameter to reduce the magnitude of these integrality violations, but the cost of solving the optimization problem may increase significantly as a result.

The fact that modern computers represent integer values using 32-bit values places some restrictions on the range of an integral variable. Specifically, the largest and smallest bounds that can be placed on an integer variable are +/- 2,000,000,000. Furthermore, integer variables with infinite bounds actually have these values as their implicit bounds. A solution is not considered feasible unless all integer variables take values that satisfy these bounds.

Binary Variables

Binary variables are the most constrained variable type that can be added to your model. A binary variable takes a value of either 0 or 1.

Again, due to the limitations of finite-precision arithmetic, binary variables will often take values that aren't exactly integral. The magnitude of the allowed integrality violation is controlled by the IntFeasTol parameter.

Semi-Continuous and Semi-Integer Variables

You can also add semi-continuous or semi-integer variables to your model. A semi-continuous variable has the property that it takes a value of 0, or a value between the specified lower and upper bounds. A semi-integer variable adds the additional restriction that the variable also take an integral value.

Again, these variables may violate these restrictions up to tolerances. In this case, the relevant tolerance is IntFeasTol (even for semi-continuous variables).

10.2 Constraints

A constraint in Gurobi captures a restriction on the values that a set of variables may take. The simplest example is a linear constraint, which states that a linear expression on a set of variables take a value that is either less-than-or-equal, greater-than-or-equal, or equal to another linear expression. More complicated constraints are also supported, including quadratic constraints (e.g., $x^2 + y^2 <= z^2$), logical constraints (e.g., logical AND on binary variables, if-then, etc.), and a few non-linear functions (e.g., $y = sin(x)$).

We now consider a few more details about linear, SOS, quadratic (both convex and non-convex), and general constraints. General constraints are the catch-all we use for the constraint types that don't fit in the other categories.

Recall that Gurobi works in finite-precision arithmetic, so constraints are only satisfied *to tolerances*. Tolerances can be tightened to reduce such violations, but there are limits to how small the violations can be -- errors are inherent in floating-point arithmetic. This point will be reiterated in several places in this section.

Linear Constraints

A linear constraint allows you to restrict the value of a linear expression. For example, you may require that any feasible solution satisfies the constraint $3x + 4y \leq 5z$. Note that the matrix-oriented Gurobi APIs (C, MATLAB, and R) require the right-hand side of a linear constraint to be

a constant, while the object-oriented APIs (C++, Java, .NET, and Python) allow arbitrary linear expressions on both sides of the comparator.

The computed solution should satisfy the stated constraint to within FeasibilityTol (although it may not in cases of numerical ill-conditioning -- we'll discuss this shortly).

Gurobi supports a limited set of comparators. Specifically, you can constrain an expression to be less-than-or-equal, greater-than-or-equal, or equal another. We do not support strict less-than, strict greater-than, or not-equal comparators. While these other comparators may seem appropriate for mathematical programming, we exclude them to avoid potential confusion related to numerical tolerances. Consider a simple example of a strict inequality constraint on a pair of continuous variables: $x > y$. How large would $x - y$ need to be in order to satisfy the constraint? Rather than trying to embed a subtle and potentially confusing strategy for handling such constraints into the solver, we've chosen not to support them instead.

SOS Constraints

A Special-Ordered Set, or SOS constraint, is a highly specialized constraint that places restrictions on the values that variables in a given list can take. There are two types of SOS constraints. In an SOS constraint of type 1 (an SOS1 constraint), at most one variable in the specified list is allowed to take a non-zero value. In an SOS constraint of type 2 (an SOS2 constraint), at most two variables in the specified, ordered list are allowed to take a non-zero value, and those non-zero variables must be contiguous in the list. The variables in an SOS constraint can be continuous, integer, or binary.

Again, tolerances play an important role in SOS constraints. Specifically, variables that take values less than IntFeasTol (in absolute value) are considered to be zero for the purposes of determining whether an SOS constraint is satisfied.

An SOS constraint is described using a list of variables and a list of corresponding weights. While the weights have historically had intuitive meanings associated with them, we simply use them to order the list of variables. The weights should be unique. This is especially important for an SOS2 constraint, which relies on the notion of *contiguous* variables. Since the variables in the SOS are ordered by weight, contiguity becomes ambiguous when multiple variables have the same weight.

It is often more efficient to capture SOS structure using linear constraints rather than SOS constraints. The optimizer will often perform this reformulation automatically. This is controlled with four parameters: PreSOS1BigM, PreSOS1Encoding, PreSOS2BigM and PreSOS2Encoding.

The reformulation adds constraints of the form $x \leq Mb$, where x is the variable that participates in the SOS constraint, b is a binary variable, and M is an upper bound on the value of variable x. Large values of M can lead to numerical issues. The two parameters PreSOS1BigM and PreSOS2BigM control the maximum value of M that can be introduced by this reformulation. SOS constraints that would require a larger value aren't converted. Setting one of these parameters to 0 disables the corresponding reformulation.

Additionally, there are several known integer formulations for SOS1 and SOS2 constraints. These reformulations differ in the number of variables that they introduce to the problem, in the complexity of the resulting LP relaxations, and in their properties in terms of branching and cutting planes. The two parameters PreSOS1Encoding and PreSOS2Encoding control the choice of the reformulation performed.

Quadratic Constraints

A quadratic constraint allows you to restrict the value of a quadratic expression. For example, you may require that any feasible solution satisfy the constraint $3x^2 + 4y^2 + 5z \leq 10$. Note that the matrix-oriented Gurobi APIs (C, MATLAB, and R) require the right-hand side of a quadratic constraint to be a constant, while the object-oriented APIs (C++, Java, .NET, and Python) allow arbitrary quadratic expressions on both sides of the comparator.

The computed solution should satisfy the stated constraint to within FeasibilityTol. Quadratic constraints are often much more challenging to satisfy than linear constraints, so tightening the parameter may increase runtimes dramatically.

Gurobi can handle both convex and non-convex quadratic constraints. However, there are some subtle and important differences in how the different constraint types are handled. In general, it is much easier to solve a model whose constraints all have convex feasible regions. It is actually quite difficult to recognize all such cases, but the following forms are always recognized:

- $x^T Q x + q^T x \leq b$, where Q is Positive Semi-Definite (PSD)
- $x^T x \leq y^2$, where x is a vector of variables, and y is a non-negative variable (a Second-Order Cone constraint)
- $x^T x \leq yz$, where x is a vector of variables, and y and z are non-negative variables (a rotated Second-Order Cone constraint)

To be more precise, a quadratic constraint will always be recognized as convex if presolve is able to transform it into one of these forms. Note that if all quadratic terms in a quadratic constraint contain at least one binary variable, then presolve will always be able to transform it to a convex form.

Why distinguish between convex and non-convex quadratic constraints? In some situations you may know that your problem should be convex, and thus it may be a sign of a modeling error if your model isn't recognized as such. To avoid accidentally solving a much harder problem than you may have intended, you can set the NonConvex parameter to either 0 or 1. In the default setting of -1 or if the NonConvex parameter is set to 2, Gurobi will accept arbitrary quadratic constraints and attempt to solve the resulting model using the appropriate algorithm.

Note that other non-convex quadratic solvers often only find locally optimal solutions. The algorithms in Gurobi explore the entire search space, so they provide a globally valid lower bound on the optimal objective value, and given enough time they will find a globally optimal solution (subject to tolerances).

We would like to note a subtle point here regarding terminology. A quadratic constraint that involves only products of disjoint pairs of variables is often called a *bilinear constraint*, and a model that contains bilinear constraints is often called a *bilinear program*. Bilinear constraints are a special case of non-convex quadratic constraints, and the algorithms Gurobi uses to handle the latter are also well suited to solving bilinear programming problems.

General Constraints

The previously-described constraints are typically (but not always) handled directly by the underlying optimization algorithms. Gurobi includes an additional set of higher-level constraints, which

we collectively refer to as *general constraints*, that require special handling. We think of these as belonging to two types: simple constraints and function constraints

Simple general constraints are a modeling convenience. They allow you to state fairly simple relationships between variables (min, max, absolute value, logical OR, etc.). Techniques for translating these constraints into lower-level modeling objects (typically using auxiliary binary variables and linear or SOS constraints) are well known and can be found in optimization modeling textbooks. By automating these translations and removing the need for the modeler to perform them, the hope is that these simple general constraints will allow you to write more readable and more maintainable models.

Function constraints allow you to state much more complex relationships between variables; you can require that $y = f(x)$, where x and y are Gurobi decision variables and $f()$ is chosen from a predefined list of nonlinear functions. The common theme among these functions is that there is no simple way to restate the associated constraints using the primitive objects (like integer variables and linear constraints) that a standard MIP solver wants to work with. Alternate algorithms are required, and the resulting models are typically much more difficult to solve than models that do not contain these constraints.

Simple General Constraints

Gurobi supports the following simple general constraints, each with its own syntax and semantics:

- **MAX constraint:** The constraint $r = \max\{x_1, \ldots, x_k, c\}$ states that the *resultant variable* r should be equal to the maximum of the *operand variables* $x_1, \ldots, x_k$ and the *constant* c. For example, a solution $(r = 3, x_1 = 2, x_2 = 3, x_3 = 0)$ would be feasible for the constraint $r = \max\{x_1, x_2, x_3, 1.7\}$ because 3 is indeed the maximum of 2, 3, 0, and 1.7.

- **MIN constraint:** Similar to a MAX constraint, the constraint $r = \min\{x_1, \ldots, x_k, c\}$ states that the *resultant variable* r should be equal to the minimum of the *operand variables* $x_1, \ldots, x_k$ and the *constant* c.

- **ABS constraint:** The constraint $r = \text{abs}\{x\}$ states that the *resultant variable* r should be equal to the absolute value of the *operand variable* x. For example, a solution $(r = 3, x = -3)$ would be feasible for the constraint $r = \text{abs}\{x\}$.

- **AND constraint:** The constraint $r = \text{and}\{x_1, \ldots, x_k\}$ states that the binary *resultant variable* r should be 1 if and only if all of the binary *operand variables* $x_1, \ldots, x_k$ are equal to 1. For example, a solution $(r = 1, x_1 = 1, x_2 = 1, x_3 = 1)$ would be feasible for the constraint $r = \text{and}\{x_1, x_2, x_3\}$. Note that any involved variables that are not already binary are converted to binary.

- **OR constraint:** Similar to an AND constraint, the constraint $r = \text{or}\{x_1, \ldots, x_k\}$ states that the binary *resultant variable* r should be 1 if and only if at least one of the binary *operand variables* $x_1, \ldots, x_k$ is equal to 1. Note that any involved variables that are not already binary are converted to binary.

- **NORM constraint:** The constraint $r = \text{norm}\{x_1, \ldots, x_k\}$ states that the *resultant variable* r should be equal to the vector norm of the *operand variables* $x_1, \ldots, x_k$. A few options are available: the 0-norm, 1-norm, 2-norm, and infinity-norm.

- **INDICATOR constraints:** An indicator constraint $y = f \rightarrow a^T x \leq b$ states that if the binary *indicator variable* y is equal to f in a given solution, where $f \in \{0, 1\}$, then the linear constraint $a^T x \leq b$ has to be satisfied. On the other hand, if $y \neq f$ (i.e., $y = 1 - f$) then the linear constraint may be violated. Note that the sense of the linear constraint can also be $=$ or $\geq$; refer to this earlier section for a more detailed description of linear constraints. Note also that declaring an INDICATOR constraint implicitly declares the indicator variable to be of binary type.

- **Piecewise-linear constraints:** A piecewise-linear constraint $y = f(x)$ states that the point (x, y) must lie on the piecewise-linear function $f()$ defined by a set of points $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined.

Note that adding any of these constraints to an otherwise continuous model will transform it into a MIP

As stated above, each general constraint has an equivalent MIP formulation that consists of linear and SOS constraints, and possibly auxiliary variables. Thus, you could always model such constraints yourself without using a Gurobi general constraint. For example, the MAX constraint $r = \max\{x_1, \ldots, x_k, c\}$ can be modeled as follows:

$$\begin{array}{rcll} r & = & x_j + s_j & \text{for all } j = 1, \ldots, k \\ r & = & c + s_{k+1} & \\ z_1 + \ldots + z_{k+1} & = & 1 & \\ SOS1(s_j, z_j) & & & \text{for all } j = 1, \ldots, k+1 \\ s_j & \geq & 0 & \text{for all } j = 1, \ldots, k+1 \\ z_j & \in & \{0, 1\} & \text{for all } j = 1, \ldots, k+1 \end{array}$$

The first two constraints state that $r \geq \max\{x_1, \ldots, x_k, c\}$, i.e., that the resultant variable r must be at least as large as each of the operand variables x_j and the constant c. This could be modeled using inequalities, but the explicit slack variables play an important role in the constraints that follow.

The next two constraints enforce $r \leq \max\{x_1, \ldots, x_k, c\}$, which ensures that r is equal to the MAX expression. Enforcing this side of the equality is actually a lot more complicated. We need to introduce binary auxiliary variables $z_j \in \{0, 1\}$, and SOS1 constraints to required that at most one of the two variables s_j and z_j can be non-zero, which models the implication $z_j = 1 \rightarrow s_j = 0$. Due to the third constraint, one z_j will be equal to 1 and thus at least one s_j will be zero. Hence, $r = x_j$ for at least one j due to the first constraint, or $r = c$ due to the second constraint.

Tolerances play a role in general constraints, although as you might expect, the exact role depends on the constraint type. As a general rule, violations in the resultant will be smaller than the feasibility tolerance, and integrality violations in integer resultants will also satisfy the integrality tolerance.

By most measures, general constraints are just a means of concisely capturing relationships between variables while removing the burden of creating an equivalent MIP formulation. However, general constraints have another potential advantage: Gurobi might be able to simplify the MIP formulation if it can prove during presolve that the simplified version suffices for the correctness of the model. For this reason, Gurobi might be able to produce a smaller or tighter representation of the general constraint than you would get from the most general formulation. For example, it

might be the case that $r \leq \max\{x_1, \ldots, x_k, c\}$ is already implied by the other constraints in the model, so that a simple set of inequalities

$$\begin{array}{rcll} r & \geq & x_j & \text{for all } j = 1, \ldots, k \\ r & \geq & c & \end{array}$$

to describe $r \geq \max\{x_1, \ldots, x_k, c\}$ suffices to model the relevant part of the MAX constraint.

Norm Constraint

The norm constraint introduces a few complications that are important to be aware of. As mentioned above, this constraint allows you to set one variable equal to the norm of a vector of variables. A few norms are available. The L1 norm is equal to the sum of the absolute values of the operand variables. The L-infinity norm is equal to the maximum absolute value of any operand. The L2 norm is equal to the square root of the sum of the squares of the operands. The L0 norm counts the number of non-zero values among the operands.

Regarding the L2 norm, one obvious complication comes from the fact that enforcing it requires a quadratic constraint. If your model only ever bounds the result from above (e.g., $r = ||x|| <= 1$), then the resulting constraint will be convex. If your model was otherwise convex, the resulting model will be a (convex) QCP. However, if you try to bound the result from below (e.g., $r = ||x|| >= y$), adding the L2 norm constraint will lead to a non-convex QCP model, which will typically be significantly harder to solve.

Regarding the L0 norm, note that results obtained with this constraint can be counter-intuitive. This is a consequence of the fact that for nearly any feasible solution with a variable at exactly 0, you can add a small value into that variable while still satisfying all associated constraints to tolerances. The net result is that a lower bound on the L0 norm is often satisfied by "cheating" - by setting enough variables to values that are slightly different from zero. We strongly recommend that you only bound the result from above. That is, you should avoid using the resultant in situations where the model incentivizes a larger value. This would include situations where the objective coefficient is negative, as well as situations where a larger value for the variable could help to satisfy a constraint (e.g., a greater-than constraint where the resultant appears with a positive coefficient).

Function Constraints

Gurobi supports the following function constraints, each with somewhat different syntax and semantics (x and y below are Gurobi decision variables, and other terms are constants provided as input when the constraint is added to the model):

- **Polynomial:** $y = p_0 x^n + p_1 x^{n-1} + ... + p_n x + p_{n+1}$
- **Natural exponential:** $y = \exp(x)$ or $y = e^x$
- **Exponential:** $y = a^x$, where $a > 0$ is the base for the exponential function
- **Natural logarithm:** $y = \log_e(x)$ or $y = \ln(x)$
- **Logarithm:** $y = \log_a(x)$, where $a > 0$ is the base for the logarithmic function
- **Logistic:** $y = \frac{1}{1+exp(-x)}$ or $y = \frac{1}{1+e^{-x}}$
- **Power:** $y = x^a$, where $x \geq 0$ for any a and $x > 0$ for $a < 0$

- **Sine:** $y = \sin(x)$
- **Cosine:** $y = \cos(x)$
- **Tangent:** $y = \tan(x)$

As noted earlier, there is no good way to restate these nonlinear constraints so that a standard MIP solver can find optimal solutions that satisfy the them to high accuracy in a reasonable amount of time. The most effective approaches to solving such problems involve replacing the constraints with piecewise-linear approximations. Gurobi provides two different mechanisms for doing so:

- A static piecewise-linear approximation, where the nonlinear function is approximated once, before the MIP solution process begins. That model can then be solved using the standard MIP solver, but the accuracy of the approximation will of course depend heavily on the number of pieces used. For some nonlinear functions, the number of pieces required to achieve the desired accuracy may be lead to excessive solution times.
- A dynamic piecewise-linear approximation, using *outer approximation* and *spatial branch-and-bound*. In this approach, the MIP solver is extended to compute an approximation in the neighborhood of the current relaxation solution at each node of the branch-and-bound search. The solver can compute very accurate solutions using this approach, but having the model change at each node in the branch-and-bound tree greatly complicates the search, which can lead to much less efficient tree exploration.

The most effective approach will depend on your model and your accuracy goals. That's why we provide both options.

Function Constraints with Static Piecewise-Linear Approximation

As noted earlier, the first option for managing function constraints is to perform a static piecewise-linear approximation, yielding a MIP model that can be handed to the standard MIP solver. Gurobi will take this approach for any function where the FuncNonlinear attribute is set to 0 (or that attribute is set to -1 and the global FuncNonlinear parameter is set to 0).

With this approach to handling non-linearity, you face a fundamental cost-versus-accuracy tradeoff: adding more pieces produces smaller approximation errors, but also increases the cost of solving the problem. The tradeoff can be complex. Gurobi provides a set of three attributes that help to navigate this tradeoff: FuncPieces, FuncPieceLength, FuncPieceError. They are used as follows:

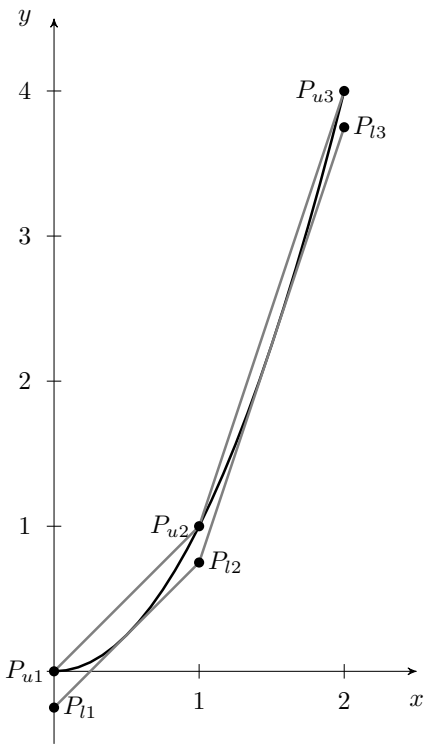
- If you would like to choose the number of pieces to use for the approximation, set the FuncPieces attribute to the desired value. All pieces will have equal width. This approach allows you to control the size of the approximation.
- If you would like to choose the width of each piece, set the FuncPieces attribute to a special value of 1 and set the FuncPieceLength attribute equal to the desired width of each piece. This approach provides some control over both the size and the error of the approximation. While this may appear to be a minor variation of the first option, note that presolve may tighten the domain of x, often substantially, which can make it difficult to predict the relationship between the width of each piece and the number of pieces.

- If you would like to set the maximum error you are willing to tolerate in the approximation, set the FuncPieces attribute to a special value of -1 and set the FuncPieceError attribute equal to the maximum absolute approximation you are willing to tolerate. Gurobi will choose pieces, typically of different sizes, to achieve that error bound. Note that the number of pieces required may be quite large if you set a tight error tolerance. You can control the maximum relative error rather than the absolute error by setting the FuncPieces attribute to -2 instead of -1.

These are attributes on the general constraints, so you can choose different values for each individual constraint.

The other relevant attribute is FuncPieceRatio, which controls whether the approximation is an underestimate of the function (0.0), an overestimate (1.0), or somewhere in between (any value strictly between 0.0 and 1.0). You can also choose the special value of -1, which will choose points that are on the original function.

Consider the following simple example:



The goal is to find an approximation of the polynomial $y = x^2$. We've set `FuncPieces` to 1 and `FuncPieceLength` to 1.0, so we're performing an approximation with fixed-width pieces of width 1.0. The domain of x is $[0, 2]$, so the approximation has two pieces. The figure shows 6 points: $P_{u1}(0, 0), P_{u2}(1, 1), P_{u3}(2, 4)$, and $P_{l1}(0, -0.25), P_{l2}(1, 0.75), P_{l3}(2, 3.75)$. If `FuncPieceRatio` is set to 0.0, the approximation would be built from the points below the function (P_{l1}, P_{l2}, and P_{l3}). Similarly, if it is set to 1.0, the approximation would be built from the points above the function (P_{u1}, P_{u2}, and P_{u3}). A value of 0.6 would use weighted combinations of the points: 0.6 times P_{ui}

plus 0.4 times P_{li}. In this case, the line segments would be built from the points $(0, -0.1), (1, 0.9)$, and $(2, 3.9)$. If `FuncPieceRatio` is set to -1, meaning that the approximation would be built from points that are on the original function, in this case the upper points (P_{u1}, P_{u2}, and P_{u3}) fit the bill. This will always be the case for a convex function.

Recall that you can set `FuncPieces` to -1 to control the maximum absolute error. In this case, choosing a `FuncPieceError` value of 0.25 would give the piecewise approximation shown in the figure, since the distance between the upper and lower curves is always 0.25. A smaller error value would of course lead to more pieces. We should add that piece widths will typically be non-uniform when limiting the maximum approximation error. The approximation algorithms we use try to limit the number of pieces needed to meet the error targets, which often requires more refinement in some portions of the domain than in others.

Note that the approximations are guaranteed to be under- and over-estimates in all cases except for polynomials of degree greater than 5. Finding the roots of higher-degree polynomials, which would be required to guarantee this property, is quite difficult.

If you wish to experiment with different approaches to approximating a set of functions, it is often convenient to be able to change the approach for all functions at once. We provide a set of parameters with the same names as the attributes to make this easier: FuncPieces, FuncPieceLength, FuncPieceError, and FuncPieceRatio. If you set the `FuncPieces` attribute on a function constraint to 0, then the approximation approach for that constraint will be determined by the parameter settings instead.

For some of the supported functions, modest x values can lead to enormous y values (and vice-versa). This can cause numerical issues when solving the resulting piecewise-linear MIP model. To avoid such issues, we limit the range of any x or y that participates in a function constraint to [-1e+6, 1e+6]. The parameter FuncMaxVal allows you to change these limits, but we recommend that you proceed with caution.

We should point out we handle violations and tolerances differently for PWL approximations, which can sometimes lead to unexpected results. For one, the feasibility tolerance for function constraints is specified in FuncPieceError rather than through the standard feasibility tolerance. We also interpret violations differently. The violation of a function constraint is computed as the Euclidian distance from the solution to the nearest point on the function. Computing this distance exactly can be quite involved for some functions, so we actually compute an over-estimate in some cases. The net result is that the reported violations may be larger than you expect.

Another possible source of unexpected results comes from the fact that solutions that satisfy the original nonlinear function may not satisfy the piecewise-linear approximation of that function. This may lead to sub-optimal solutions or even conclusions of infeasibility. Consider a simple example with two constraints: $y = 2x - 1$ and $y = x^2$. Clearly $(x, y) = (1, 1)$ is a feasible solution, but a piecewise-linear approximation could introduce breakpoints at $x = 0.9$ and $x = 1.1$. The resulting approximation gives a y value of 1.01 at $x = 1$, which is sufficiently far from the actual function value that Gurobi will not consider that a valid solution and declare the model infeasible, since there are no other solutions to the constraints. Reducing the maximum approximation error (by setting FuncPieces to -1 and FuncPieceError to a much smaller value) would help, but this isn't always the best way to address the problem, since tighter error tolerances can substantially increase the number of pieces in the approximation and thus the cost. We recommend the following approach when you encounter unexpected results. For inequalities, you should ask for an approximation that always overestimates or underestimates the function (depending on the sense of the constraint), to

ensure that your approximation will always satisfy the constraint. The FuncPieceRatio parameter allows you to do this. For equalities, if you have a sense of where your solution is likely to lie, one option for managing the size of the approximation is to introduce additional variables to capture your function in different ranges, and then perform approximations with different levels of accuracy on these different pieces.

While users could perform piecewise-linear approximations themselves, there are several advantages to asking Gurobi to do it instead. First, Gurobi can often reduce the domains of variables, by using bound strengthening in presolve, or by exploiting repetition in periodic functions like sine or cosine. Smaller domains means fewer pieces to achieve the same accuracy. Gurobi also provides many options to make experimentation easier (for error control, piece length, etc.). These options can be quite difficult to implement and maintain.

Function Constraints With Dynamic Piecewise-Linear Approximation

The alternative for managing function constraints is a more dynamic approach, using *spatial branching* and *outer approximation*. Gurobi will take this approach for any function where the FuncNonlinear attribute is set to 1 (or that attribute is set to -1 and the global FuncNonlinear parameter is set to 1).

Solving a model with (nonconvex) nonlinear constraints to global optimality is well known to be a hard task. The idea behind spatial branching is to divide the overall solution space into portions, and to compute valid primal and dual bounds for each portion. Such bounds are obtained by computing a linear approximation to the feasible space for each region (a so-called *outer approximation*) that contains all feasible solutions for the original nonlinear constraints in that region. Bounds for a region can often be tightened by *spatial branching*, where the domain of a variable (integer or continuous) is split, allowing hopefully tighter outer approximations to be computed for the resulting feasible subregions. Bounds for the original model can be obtained by combining the bounds from the leafs of the resulting branch-and-bound tree. This process continues until the desired optimality gap (MIPGap) is achieved.

Valid primal bounds come from feasible solutions found during the search. They can be found using various heuristics, or they may come from solutions to the current node relaxation that happen to satisfy the nonlinear and integrality constraints (relaxation solutions will always satisfy linear and bound constraints because these are still present in the relaxation). Valid dual bounds are found by solving an LP over a polyhedron that contains all feasible solutions to the original linear and nonlinear constraints in that region (the outer approximation).

Currently, Gurobi only directly supports the univariate nonlinear functions listed in the Function Constraints section. More complex functions must be disaggregated into a cascade of supported univariate functions. For example, the nonlinear equality constraint

$$x_3 = 100000 \cdot \frac{x_1}{x_1 + 1000 \cdot x_2}$$

can be disaggregated into

$$\begin{aligned} z_1 &= x_1 + 1000 \cdot x_2 \\ z_2 &= z_1^{-1} \\ x_3 &= 100000 \cdot x_1 \cdot z_2. \end{aligned}$$

Such disaggregation is standard practice for most global non-linear solvers (but it is typically performend behind the scenes).

While the disaggregated reformulation is mathematically equivalent to the original model, we should note that the presence of large coefficients in the expression can lead to larger violations of the original constraints than you might expect, due to the accumulation of violations of the disaggregated constraints.

Suppose we are using the default feasibility tolerance of 10^{-6} and Gurobi produces the following solution for the disaggregated model:

$$\begin{aligned} x_1 &= 1 \\ x_2 &= 1 \\ z_1 &= 1001.000001 \\ z_2 &= 0.001000000998003 \\ x_3 &= 100.0000998003 \end{aligned}$$

As one can see, this solution is indeed feasible within the tolerance for the disaggregated model. However,

$$100000 \cdot \tfrac{1}{1+1000 \cdot 1} = 99.9000999001 \neq 100.0000998003 = x_3.$$

The 100000 coefficient magnifies the error in the intermediate result, leading to a violation of almost 0.1 on the original model, which is much larger than the specified feasibility tolerance.

This example illustrates that a solution that is feasible for the disaggregated model may not necessarily be feasible for the aggregated (original) model. Clearly, the above example is an extreme case, but this issue can definitely arise in other situations. This makes scaling and tight coefficient ranges extremely important when formulating and solving nonlinear models.

10.3 Objectives

Every optimization model has an objective function, which is the function on the decision variables that you wish to minimize or maximize. The objective is meant to capture your goals in solving the problem. Given a set of feasible solutions, the objective tells the solver which is preferred.

Most optimization problems have multiple optimal solutions, plus many solutions whose objectives are within a small gap from the optimal value. The solution that is returned by Gurobi depends on the type of problem you are solving. The simple rule is that Gurobi returns a single optimal solution for continuous models (LP, QP, and QCP), and a sequence of improving solutions for discrete models (MIP, MIQP, and MIQCP).

The Gurobi algorithms work on solving a model until they find a solution that is optimal to within the specified tolerances. For the simplex algorithms (including barrier with crossover), the relevant tolerance is the OptimalityTol. For the barrier algorithm (without crossover), the relevant tolerances are the BarConvTol or BarQCPConvTol (depending on the problem type). You can relax these tolerances, but note that it is rare for this to significantly improve solution times. The simplex and barrier algorithms both return a single optimal solution.

For discrete models, while you can ask the MIP solver to find a solution with the best possible objective value, it is more common to stop when the solution objective is within a specified gap from the optimal value. This optimality gap is controlled by the MIPGap parameter, and the default value is 0.01%.

The MIP solver typically finds multiple sub-optimal solutions on the way to eventually finding an optimal solution. These intermediate solutions can be queried once the optimization is complete

(using the Xn attribute). You can use the Solution Pool feature to take a more systematic approach to finding multiple solutions. This feature allows you to indicate how many solutions you would like, to specify the largest gap to the optimal value you are willing to accept, etc.

We should add that it is possible to specify a pure feasibility problem, where the sole goal is to find a solution that satisfies the constraints. You can think of a feasibility problem as an optimization problem with a constant objective function.

The available objective types are linear, piecewise-linear, quadratic (both convex and non-convex), and multi-objective. While the property of having multiple objectives may appear to be orthogonal to the types of the objectives, Gurobi only supports multi-objective models where all objectives are linear.

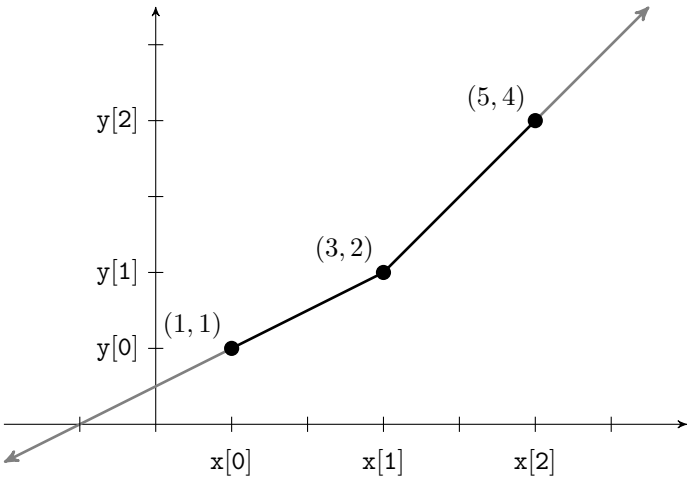
Linear Objectives

The simplest and most common objective function is linear - minimizing or maximizing a linear function on the decision variables (e.g., $3x + 4y + 2$). Linear objectives can be specified in a few ways. The first is by providing an objective term when the decision variable is added to the model (typically through the `addVar` method). The second is by setting the Obj attribute on the variable. The third and most convenient approach, available in the object-oriented interfaces, is through the `setObjective` method (in C++, Java, .NET, or Python). This method accepts a linear expression object as its argument.

A model with a linear objective, only linear constraints, and only continuous variables is a Linear Program (LP). It can be solved using the simplex or barrier algorithms.

Piecewise-Linear Objectives

A useful variant of a linear objective is a *piecewise*-linear objective, where the objective for a single variable is captured in a set of linear pieces. For example, suppose we want to define the objective value $f(x)$ for variable x as follows:



The vertices of the function occur at the points $(1, 1)$, $(3, 2)$ and $(5, 4)$, so we define $f(1) = 1$, $f(3) = 2$ and $f(5) = 4$. Other objective values are linearly interpolated between neighboring points. The first pair and last pair of points each define a ray, so values outside the specified x values are extrapolated from these points. Thus, in our example, $f(-1) = 0$ and $f(6) = 5$.

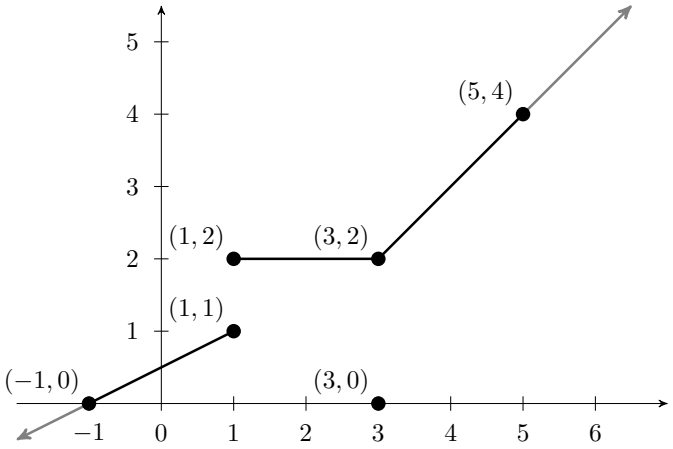
More formally, a piecewise-linear function is defined by a set of n points:

$$\mathtt{x} = [x_1, \ldots, x_n], \quad \mathtt{y} = [y_1, \ldots, y_n]$$

These define the following piecewise-linear function:

$$f(v) = \begin{cases} y_1 + \frac{y_2 - y_1}{x_2 - x_1}(v - x_1), & \text{if } v \leq x_1, \\ y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(v - x_i), & \text{if } v \geq x_i \text{ and } v \leq x_{i+1}, \\ y_n + \frac{y_n - y_{n-1}}{x_n - x_{n-1}}(v - x_n), & \text{if } v \geq x_n. \end{cases}$$

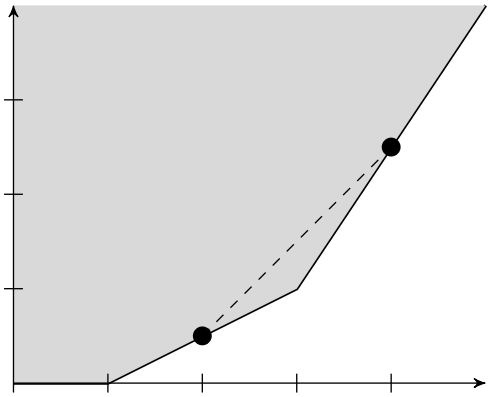
We also allow special cases, such as jumps and single points, which are quite useful to define the fixed charges or the penalties. A jump at $x = x_i$ means that the left piece and the right piece don't intersect at $x = x_i$, i.e. we have $(x_{i-1}, y_{i-1}), (x_i, y_i), (x_{i+1}, y_{i+1}), (x_{i+2}, y_{i+2})$ with $x_i = x_{i+1}$ and $y_i \neq y_{i+1}$. So for the left piece, i.e. $x_{i-1} \leq x < x_i$, the line segment between points (x_{i-1}, y_{i-1}) and (x_i, y_i) defines y, for the right piece, i.e. $x_i \leq x < x_{i+2}$, the line segment between points (x_{i+1}, y_{i+1}) and (x_{i+2}, y_{i+2}) defines y. Since we must allow some tolerance for numeric computation, it means that at $x = x_i$, y can take the value of either y_i or y_{i+1}. A single point at $x = x_i$ means that both left and right pieces extend to $x = x_i$, but both have different y values than y_i. It can be described by the five points $(x_{i-2}, y_{i-2}), (x_{i-1}, y_{i-1}), (x_i, y_i), (x_{i+1}, y_{i+1}), (x_{i+2}, y_{i+2})$ with $x_{i-1} = x_i = x_{i+1}$ and $y_i \neq y_{i-1}$ and $y_i \neq y_{i+1}$. Note that y_{i-1} and y_{i+1} can be equal or different. Because of the tolerance, it means that at $x = x_i$, y can take the value of y_{i-1}, y_i or y_{i+1}. Here below is an example with a jump and a single point.



The above piecewise function for variable x are defined by 7 points $(-1, 0), (1, 1), (1, 2), (3, 2), (3, 0), (3, 2)$ and $(5, 4)$. It has a jump at $x = 1$ from $(1, 1)$ to $(1, 2)$ and a single point $(3, 0)$. Note that both left and right points have the same x coordinate and for this example the two points are the same.

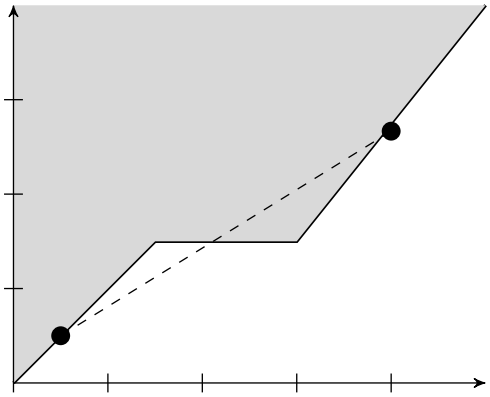
Note that a piecewise-linear objective can change the type of a model. Specifically, including a non-convex piecewise-linear objective function in a continuous model will transform that model into a MIP. This can significantly increase the cost of solving the model.

How do you determine whether your piecewise-linear objective is convex? A convex function has the property that you can't obtain a better objective value by interpolating between two points on the function. In the figure below, you will note that all convex combinations of points on the piecewise-linear function are in the shaded (feasible) region.



Stated another way, successive pieces will have non-decreasing slopes in a convex piecewise-linear objective (assuming you are minimizing).

In contrast, in a non-convex piecewise-linear function you can get a better value by interpolating between points. In the figure below, the value of $f(1)$ for the piecewise-linear function is worse (larger) than the value obtained by interpolation.



Piecewise-linear objectives are defined on variables using a special method (in C, C++, Java, .NET, or Python). Each variable can have its own piecewise-linear objective function, and each function requires a separate call to this method.

A variable can't have both a linear and a piecewise-linear objective term. Setting a piecewise-linear objective for a variable will set the Obj attribute on that variable to 0. Similarly, setting the `Obj` attribute will delete the piecewise-linear objective on that variable.

We should point out that it is fairly easy to specify a piecewise-linear function on a variable using a few extra variables and simple linear objective terms. The advantages of using the piecewise-linear

API methods are twofold. The first is convenience - specifying the function directly leads to simpler and more readable code. The second is that Gurobi includes a piecewise-linear simplex algorithm. If you provide a model that contains only linear constraints, only continuous variables, and only linear or convex piecewise-linear objective terms, then this specialized simplex algorithm will be used to solve the model. If your piecewise-linear function contains a large number of segments, the specialized algorithm will be much faster than the standard simplex solver.

Quadratic Objectives

Your optimization objective can also contain quadratic terms (e.g., $3x^2 + 4y^2 + 2xy + 2x + 3$). You specify quadratic objectives in the object-oriented interfaces by building quadratic expressions and then calling *setObjective* (C++, Java, .NET, or Python). In C, you input your quadratic terms using GRBaddqpterms.

There are four distinct algorithms that could be used to solve a model with a quadratic objective. The appropriate one depends on a few specific properties of the objective and the rest of the model.

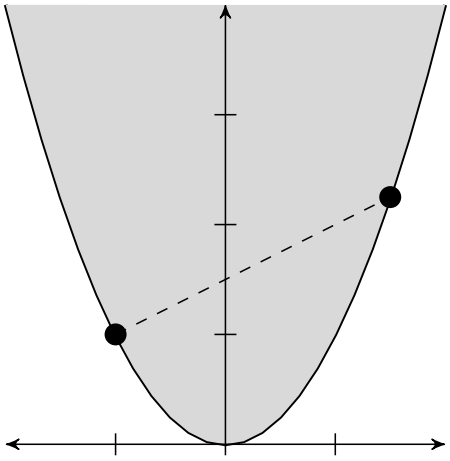
- **Continuous Convex QP** If your quadratic objective is convex and your model only contains linear constraints and continuous variables, then your model is a quadratic program (QP) and can be solved using either the simplex or barrier algorithms. QP simplex will return an optimal basic solution. Gurobi does not have a QP crossover, so QP barrier will return an interior solution.

- **Discrete QP with Convex Relaxation** If your quadratic objective is convex but the model contains discrete elements (integer variables, SOS constraints, general constraints, etc.), then your model is a mixed integer quadratic program (MIQP) and is solved using the MIP solver. Since MIP relies heavily on simplex bases, the root relaxation must be solved using the primal or dual simplex algorithm.

- **Non-convex QP and Discrete QP with Non-Convex Relaxation** If your quadratic objective is not convex, then the model will be solved using the MIP solver, even if your model has no explicit discrete elements. Non-convex problems often lead to much larger solve times, and it might be that the non-convexity in your model is unexpected, for example due to an error in the model data or in the model formulation. It you think that your model should be convex (or, in the discrete case, have a convex relaxation), you can set the NonConvex parameter to 0 or 1. With this setting, a non-convex quadratic objective leads to a `GRB_ERROR_Q_NOT_PSD` error.

These properties are checked on the presolved model. As is always the case, presolve will try to simplify the model. In this context, it will try to transform a non-convex MIQP into an equivalent convex MIQP. This simplification will always succeed if each quadratic term contains at least one binary variable.

How can you determine whether your quadratic objective is convex? As was noted earlier, the crucial property for convexity is that interpolation between any two points on the function never puts you below the function (assuming minimization). In this figure, all points on a line segment between any two points on the parabola are always in the shaded region.



How does this translate to multiple variables? For a quadratic function to be convex, the associated Q matrix must be Positive Semi-Definite (PSD).

Multiple Objectives

You can also specify multiple (linear) objectives for your model, and Gurobi provides tools that allow you explore the tradeoffs between them. Refer to the Multiple Objectives section for additional details.

10.4 Tolerances and Ill Conditioning -- A Caveat

As noted at several places in this section, finite-precision arithmetic limits the precision of the solutions Gurobi computes. This limitation is managed through numerical tolerances in most cases; we treat a solution as satisfying a constraint if the violation is smaller than the corresponding tolerance. The default tolerances are chosen to be sufficiently large so that numerical errors aren't an issue for most models, yet small enough that the results of the computations are meaningful.

Unfortunately, some models suffer from severe *ill conditioning*, which can greatly complicate the search for a solution. This can show itself in a few ways. Ill conditioning can severely hurt performance, and it can lead to solutions whose constraint violations are larger than the tolerances.

Ill conditioning is a measure of the amount of error that can result when solving linear systems of equations. As noted earlier, linear and mixed-integer programming are built on top of linear solves, so errors in solving linear systems directly lead to errors in LP and MIP solutions. Serious problems arise when the error in solving a linear system is comparable to the desired tolerance. If you want to solve a linear programming problem to the default feasibility tolerance of $1e-6$, for example, and if your linear system solves produce errors that are also roughly $1e-6$, then you have no way of knowing whether your current solution is truly feasible. This can lead to oscillations, as your solution bounces between feasible and infeasible due to nothing more than numerical error, which can make it extremely difficult to achieve forward progress towards an optimal solution.

When solving linear and quadratic programming problems, we recommend that you check final primal and dual constraint violations. Duality theory states that, if your solution is primal feasible, dual feasible, and complementary, then you have an optimal solution. Complementarity is automatically enforced by the simplex method, so achieving primal and dual feasibility (to tolerances) assures that the solution is optimal (to tolerances).

When solving a MIP model (which includes any model that contains discrete or non-convex features, such as non-convex objectives, general constraints, semi-continuous variables, etc.), there is unfortunately no simple method available to check the optimality of the result. While we work hard to identify and manage the negative effects of ill conditioning, we are unable to provide a mathematical proof that the solution returned is truly optimal.

For additional information on numerical issues, please refer to the Gurobi Guidelines for Numerical Issues section of this manual.

An environment is a multi-purpose data structure in Gurobi. It is typically the first Gurobi object you create and the last one you destroy. While the workings of environments are actually quite simple, the breadth of uses can sometimes cause confusion. This section lays out the different usage scenarios to make this clearer.

At the highest level, environments provide three basic functions: (i) to capture a set of parameter settings, (ii) to delineate a (single-threaded) Gurobi session, and (iii) to hold a Gurobi license. Things only get interesting once you consider the many different ways these capabilities are used throughout the product:

- Session boundaries: Environments indicate when your program starts and stops using Gurobi.
- Configuration parameters: Environments allow you to configure your session (whether you will run locally or on a Compute Server or on the Instant Cloud, your login credentials, etc.).
- Algorithmic parameters: Environments enable you to modify algorithmic parameters, which influence how the solver solves your model.
- Concurrent environments: When using a concurrent algorithm, concurrent environments allow you to control the algorithmic parameters used in each independent solve.
- Multi-objective environments: When solving a multi-objective model, multi-objective environments allow you to control the algorithmic parameters used for each objective.

11.1 Session boundaries

One of the main purposes of an environment is to indicate when your program will start to use Gurobi, and when it is done. When Gurobi is running on your own machine, creating an environment will obtain a license, and disposing of the environment will release that license. When you are a client of a Gurobi Compute Server, starting an environment will start a job on the server (or place the job in the queue if the server is fully occupied). Disposing of the environment will end that job, allowing the next job in the queue to start. On Gurobi Instant Cloud, creating an environment will launch a cloud instance (if it hasn't been launched already). Disposing of the environment will end that session, which may result in the cloud instance being shut down (depending on the policy you've set in your Instant Cloud configuration).

If your program repeatedly creates, solves, and destroys optimization models, we strongly recommend that you do so within a single Gurobi environment. Creating a Gurobi environment incurs overhead, ranging anywhere from a quick local license check all the way to spinning up a machine on the cloud. By reusing a single environment, you avoid paying this overhead multiple times.

We also recommend that you dispose of your environment as soon as your program is done using Gurobi. Doing so releases all resources associated with that session, which in many cases can make those resources available to other users. You should pay particular attention to this topic when

using programming languages that perform garbage-collection. While it is true that environments will be disposed of eventually by automated garbage-collection, that will often happen much earlier if you dispose of them explicitly.

The actual steps for disposing of an environment will depend on the API you are using:

`Python`: Call Model.dispose() on all `Model` objects, Env.dispose() on any `Env` objects you created, or disposeDefaultEnv() if you used the default environment instead. For example:

```
# Clean up model and environment
model.dispose()
env.dispose()
gp.disposeDefaultEnv()
```

`Java`: Call GRBModel.dispose() on all `GRBModel` objects, then call GRBEnv.dispose() on the `GRBEnv` object. For example:

```
// Clean up model and environment
model.dispose()
env.dispose()
```

`.NET`: Call GRBModel.Dispose() on all `GRBModel` objects, then call GRBEnv.Dispose() on the `GRBEnv` object.

`C++`: Call the `delete` operator on all `GRBModel` objects, and then on the `GRBEnv` object.

`C`: Call GRBfreemodel() for each model, then call GRBfreeenv() for the Gurobi environment. For example:

```
/* Clean up model and environment */
GRBfreemodel(model);
GRBfreeenv(env);
```

Note that the boundaries established by an environment are for a single thread. Gurobi environments are not thread-safe, so you can't have more than one thread of control within a single environment. You can however have a single program that launches multiple threads, each with its own environment.

In Python you can take advantage of context managers for environment and model objects. Using them will guarantee that these objects are automatically disposed of. Refer to Env class documentation for more information. A typical use pattern is also shown in the example `mip1_remote.py`.

11.2 Configuration parameters

When you start a Gurobi session, you often have to provides details about your configuration. You may need to indicate whether you want to use a license on your local machine, a license from a Token Server, or perhaps you want to offload your computation to a Compute Server or to Gurobi Instant Cloud. In the case of a Token Server or a Compute Server, you have to provide the name of the server. For Compute Server and Instant Cloud, you also need to provide login credentials.

In many situations, the configuration information you need is already stored in your license file (`gurobi.lic`) or in your environment file (`gurobi.env`). These files are read automatically, so you

can simply create a standard Gurobi environment object (using GRBloadenv in **C**, or through the appropriate `GRBEnv` constructor in the object-oriented interfaces).

What if you need to provide configuration information from your application at runtime instead? You can use an *empty environment* to split environment creation into a few steps (as opposed to the standard, single-step approach mentioned above). In the first step, you would create an empty environment object (using GRBemptyenv in **C**, or through the appropriate `GRBEnv` constructor in the object-oriented interfaces). You would then set configuration parameters on this environment using the standard parameter API. Finally, you would start the environment (using GRBstartenv in **C**, or using the `env.start()` method in the object-oriented interfaces), which will use the configuration parameters you just set.

Empty environment example

To give a simple example, if you want your **Python** program to offload the optimization computation to a Compute Server named `server1`, you could say:

```
import gurobipy as gp
from gurobipy import GRB

# Set up environment
env = gp.Env(empty=True)
env.setParam('ComputeServer', 'server1:61000')
env.setParam('ServerPassword', 'passwd')
env.start()

# Load model and optimize
model = gp.read('misc07.mps', env=env)
model.optimize()
```

An equivalent **Java** program would look like this:

```
import gurobi.*;
...
  // Set up environment
  GRBenv env = new GRBEnv(true);
  env.set(GRB.StringParam.ComputeServer,  "server1:61000");
  env.set(GRB.StringParam.ServerPassword, "passwd");
  env.start();

  // Load model and optimize
  GRBModel model = new GRBModel(env, "misc07.mps");
  model.optimize()
```

An equivalent **C** program would look like this:

```
#include "gurobi_c.h"
int main(void) {
  GRBenv   *env   = NULL;
  GRBmodel *model = NULL;
  int error   = 0;

  /* Set up environment */
  error = GRBemptyenv(&env);
  if (error) goto QUIT;
  error = GRBsetstrparam(GRB_STR_PAR_COMPUTESERVER, "server1:61000");
  if (error) goto QUIT;
```

```
  error = GRBsetstrparam(GRB_STR_PAR_SERVERPASSWORD, "passwd");
  if (error) goto QUIT;
  error = GRBstartenv(env);
  if (error) goto QUIT;

  /* Load model and optimize */
  error = GRBreadmodel(env, "misc07.mps", &model);
  if (error) goto QUIT;
  error = GRBoptimize(model);
  if (error) goto QUIT;

QUIT:
  /* Clean up model and environment */
  GRBfreemodel(model);
  GRBfreeenv(env);
  return error;
}
```

This example uses the ComputeServer parameter to connect to a Compute Server. To give a few more examples of configuration parameters, you can use the CloudAccessID and CloudSecretKey parameters to provide your credentials in order to launch an Instant Cloud instance. To connect to a token server, you would use the TokenServer parameter. You can also use the LicenseID, WLSAccessID, and WLSSecret parameters to provide your IDs and secret key for your WLS license. You can find more information about WLS licenses in your account on the Gurobi Web License Manager site. The full list of Gurobi parameters can be found here.

Configuration parameters must be set before you start the Gurobi environment. Changes have no effect once the environment has been started.

In Python you can also provide such configuration parameters directly as a `dict` argument to the environment constructor, without creating an empty environment first. Please refer to the Env() documentation for an example.

11.3 Algorithmic parameters

Environments can also be used to set algorithmic parameters - parameters that control the behavior of the optimization solver. To give a few examples, the TimeLimit parameter indicates the maximum allowed runtime for any solve, while the Method parameter chooses the algorithm used to solve continuous optimization models.

Having said that, we actually recommend that you set algorithmic parameters on the model rather than on the environment. The object-oriented APIs all include `model.set` methods that enable you to do so. What's the difference between setting parameters on the environment versus on the model? Parameter values are copied from the environment when the model is created, so changes to parameter values in the environment have no effect on models that have already been created. This is a frequent source of confusion.

In the C API, there is no method to set parameters on the model directly. Instead, the model environment can be retrieved by calling GRBgetenv.

Please see the Parameter Examples section for examples on how parameters are set.

11.4 Concurrent environments

One algorithmic option available in Gurobi is concurrent optimization, where multiple independent solves are performed in parallel and Gurobi takes care of collecting and combining the results. Concurrent optimization can be quite powerful; it is actually the default approach for solving linear programming models.

The power of concurrent optimization comes from the fact that different approaches to solving a model may have different performance characteristics, and performing them in parallel allows you to stop when the first one finishes. All of our concurrent schemes have default choices for determining the strategy that each independent solve uses. However, it is also possible for you to pick different strategies. This is done through *concurrent environments*. By creating two or more concurrent environments for a model, and setting parameters on these environments, you can control exactly what each concurrent solve does.

Concurrent environments are created via API routines (in C, C++, Java, .NET, or Python). You set parameters on these environments as you would with any other environment, but in this case they only affect one of the several independent solves.

To give a simple example, in Python you could do the following:

```
# Create concurrent environments
env0 = model.getConcurrentEnv(0)
env1 = model.getConcurrentEnv(1)

# Set parameters on concurrent environments
env0.setParam('MIPFocus', 1)
env1.setParam('MIPFocus', 2)

# Perform (concurrent) optimization
model.optimize()
```

This would launch two concurrent solves on your model, one with the MIPFocus parameter set to 1 and the other with it set to 2.

Please note that parameter values are copied from the model when the concurrent environment is created. Therefore, changes to parameter values on the model have no effect on concurrent environments that have already been created. This is a frequent source of confusion.

Users of our command-line interface can set concurrent optimization parameters with `.prm` files using the ConcurrentSettings parameter.

11.5 Multi-objective environments

When solving a multi-objective model, the solution process typically proceeds in phases, where each phase solves for one objective. The standard algorithmic parameters influence the strategy used to solve the overall multi-objective model. However, in some cases you may want finer-grain control over the strategies used in each phase. The solver enables this through *multi-objective environments*.

Multi-objective environments are created via API routines (in C, C++, Java, .NET, or Python). You set parameters on these environments as you would with any other environment, but in this case they only affect one of the several objective solves.

To give a simple example, in Python you could do the following:

```
# Create multi-objective environments
env0 = model.getMultiobjEnv(0)
env1 = model.getMultiobjEnv(1)

# Set parameters on multi-objective environments
env0.setParam('Method', 2)
env1.setParam('Method', 1)
env1.setParam('Presolve', 0)

# Perform multi-objective optimization
model.optimize()
```

This would use the barrier solver (`Method=2`) for the first objective, and the dual simplex solver (`Method=1`) with no presolve (`Presolve=0`) for the second. Note that you don't need a multi-objective environment for each objective - only for those where you want parameters to take different values from those of the model itself.

Please note that parameter values are copied from the model when the multi-objective environment is created. Therefore, changes to parameter values on the model have no effect on multi-objective environments that have already been created. This is a frequent source of confusion.

The primary mechanism for querying and modifying properties of a Gurobi model is through the attribute interface. A variety of different attributes are available. Some are only populated at certain times (e.g., those related to the solution of a model), while others are available at all times (e.g., the number of variables in the model). Attributes can be associated with variables (e.g., lower bounds), constraints (e.g., the right-hand side), SOSs (e.g., IIS membership), or with the model as a whole (e.g., the objective value for the current solution).

The following tables list the full set of Gurobi attributes. The attributes have been grouped by associated modeling object. Attributes associated with the model take scalar values. So do attributes associated with solution quality, objectives in multi-objective models, scenarios in multi-scenario models, and batches. Attributes associated with variables, linear constraints, SOS constraints, quadratic constraints, and general constraints contain one entry per variable or constraint in the model. The APIs provide methods to query attribute values for individual constraints or variables, or to query their values for arrays of constraints or variables (refer to our Attribute Examples section for examples). Array queries are generally more efficient.

Note that the attributes that provide solution quality information have been split off into a separate table at the end of this section. These attributes are also associated with the model as a whole.

Some solution attributes require information that is only computed by certain Gurobi algorithms. Such cases are noted in the detailed attribute descriptions that follow. For example, the `VBasis` and `CBasis` attributes can only be queried when a simplex basis is available (a basis is available when a continuous model has been solved using primal simplex, dual simplex, or barrier with crossover). Sensitivity information (`SAObjLow`, `SAObjUp`, etc.) is also only available for basic solutions.

Model attributes:

These attributes provide information about the overall model (as opposed to information about individual variables or constraints in the model).

Attribute name	Description
NumVars	Number of variables
NumConstrs	Number of linear constraints
NumSOS	Number of SOS constraints
NumQConstrs	Number of quadratic constraints
NumGenConstrs	Number of general constraints
NumNZs	Number of non-zero coefficients in the constraint matrix
DNumNZs	Number of non-zero coefficients in the constraint matrix (in double format)
NumQNZs	Number of non-zero quadratic objective terms
NumQCNZs	Number of non-zero terms in quadratic constraints
NumIntVars	Number of integer variables
NumBinVars	Number of binary variables
NumPWLObjVars	Number of variables with piecewise-linear objective functions
ModelName	Model name
ModelSense	Model sense (minimization or maximization)
ObjCon	Constant offset for objective function
Fingerprint	Model fingerprint
ObjVal	Objective value for current solution
ObjBound	Best available objective bound (lower bound for minimization, upper bound for maximization)
ObjBoundC	Best available objective bound, without rounding (lower bound for minimization, upper bound for maximization)
PoolObjBound	Bound on best objective for solutions not in pool (lower bound for minimization, upper bound for maximization)
PoolObjVal	Objective value of alternatives solutions stored during the optimization process
MIPGap	Current relative MIP optimality gap
Runtime	Runtime for most recent optimization
Work	Work spent on most recent optimization
Status	Current optimization status
SolCount	Number of stored solutions
IterCount	Number of simplex iterations performed in most recent optimization
BarIterCount	Number of barrier iterations performed in most recent optimization
NodeCount	Number of branch-and-cut nodes explored in most recent optimization
OpenNodeCount	Number of open branch-and-cut nodes at the end of most recent optimization
ConcurrentWinMethod	Winning method of most recent concurrent optimization of continuous model
IsMIP	Indicates whether the model is a MIP
IsQP	Indicates whether the model is a QP/MIQP
IsQCP	Indicates whether the model is a QCP/MIQCP
IsMultiObj	Indicates whether the model has multiple objectives
IISMinimal	Indicates whether the current IIS is minimal

MaxCoeff	Maximum constraint matrix coefficient (in absolute value)
MinCoeff	Minimum (non-zero) constraint matrix coefficient (in absolute value)
MaxBound	Maximum finite variable bound
MinBound	Minimum finite variable bound
MaxObjCoeff	Maximum linear objective coefficient (in absolute value)
MinObjCoeff	Minimum (non-zero) linear objective coefficient (in absolute value)
MaxRHS	Maximum constraint right-hand side (in absolute value)
MinRHS	Minimum (non-zero) constraint right-hand side (in absolute value)
MaxQCCoeff	Maximum quadratic constraint matrix coefficient of quadratic part (in absolute value)
MinQCCoeff	Minimum (non-zero) quadratic constraint matrix coefficient of quadratic part (in absolute value)
MaxQCLCoeff	Maximum quadratic constraint matrix coefficient in linear part (in absolute value)
MinQCLCoeff	Minimum (non-zero) quadratic constraint matrix coefficient in linear part (in absolute value)
MaxQCRHS	Maximum quadratic constraint right-hand side (in absolute value)
MinQCRHS	Minimum (non-zero) quadratic constraint right-hand side (in absolute value)
MaxQObjCoeff	Maximum quadratic objective coefficient (in absolute value)
MinQObjCoeff	Minimum (non-zero) quadratic objective coefficient (in absolute value)
Kappa	Estimated basis condition number
KappaExact	Exact basis condition number
FarkasProof	Magnitude of infeasibility violation in Farkas infeasibility proof
TuneResultCount	Number of improved parameter sets found by tuning tool
NumStart	Number of MIP starts
LicenseExpiration	License expiration date

Variable attributes:

These attributes provide information that is associated with specific variables.

Attribute name	Description
LB	Lower bound
UB	Upper bound
Obj	Linear objective coefficient
VType	Variable type (continuous, binary, integer, etc.)
VarName	Variable name
VTag	Variable tag
X	Value in the current solution
Xn	Value in a sub-optimal MIP solution
RC	Reduced cost
BarX	Value in the best barrier iterate (before crossover)
Start	MIP start value (for constructing an initial MIP solution)
VarHintVal	MIP hint value
VarHintPri	MIP hint priority
BranchPriority	Branching priority
Partition	Variable partition

VBasis	Basis status
PStart	Simplex start vector
IISLB	Indicates whether the lower bound participates in the IIS
IISUB	Indicates whether the upper bound participates in the IIS
IISLBForce	Forces variable lower bound into (1) or out of (0) the computed IIS
IISUBForce	Forces variable upper bound into (1) or out of (0) the computed IIS
PoolIgnore	Flag variables to ignore when checking whether two solutions are identical
PWLObjCvx	Indicates whether the variable has a convex piecewise-linear objective
SAObjLow	Objective coefficient sensitivity information
SAObjUp	Objective coefficient sensitivity information
SALBLow	Lower bound sensitivity information
SALBUp	Lower bound sensitivity information
SAUBLow	Upper bound sensitivity information
SAUBUp	Upper bound sensitivity information
UnbdRay	Unbounded ray

Linear constraint attributes:

These attributes provide information that is associated with specific linear constraints.

Attribute name	Description
Sense	Constraint sense ('<', '>', or '=')
RHS	Right-hand side value
ConstrName	Constraint name
CTag	Constraint tag
Pi	Dual value (also known as the *shadow price*)
Slack	Slack in the current solution
CBasis	Basis status
DStart	Simplex start vector
Lazy	Determines whether a constraint is treated as a lazy constraint
IISConstr	Indicates whether the constraint participates in the IIS
IISConstrForce	Forces constraint into (1) or out of (0) the computed IIS
SARHSLow	Right-hand side sensitivity information
SARHSUp	Right-hand side sensitivity information
FarkasDual	Farkas infeasibility proof

SOS attributes:

These attributes provide information that is associated with specific Special-Ordered Set (SOS) constraints.

Attribute name	Description
IISSOS	Indicates whether the SOS constraint participates in the IIS
IISSOSForce	Forces the SOS constraint into (1) or out of (0) the computed IIS

Quadratic constraint attributes:

These attributes provide information that is associated with specific quadratic constraints.

Attribute name	Description
QCSense	Constraint sense ('<', '>', or '=')
QCRHS	Right-hand side
QCName	Quadratic constraint name
QCPi	Dual value
QCSlack	Slack in the current solution
QCTag	Quadratic constraint tag
IISQConstr	Indicates whether the quadratic constraint participates in the IIS
IISQConstrForce	Forces the quadratic constraint into (1) or out of (0) the computed IIS

General constraint attributes:

These attributes provide information that is associated with specific general constraints. Those starting with "Func" are only for function constraints.

Attribute name	Description
FuncPieceError	Error allowed for PWL translation
FuncPieceLength	Piece length for PWL translation
FuncPieceRatio	Controls whether to under- or over-estimate function values in PWL approximation
FuncPieces	Sets strategy for PWL function approximation
FuncNonlinear	Chooses the approximation approach used to handle a function constraint
GenConstrType	Type of general constraint
GenConstrName	General constraint name
IISGenConstr	Indicates whether the general constraint participates in the IIS
IISGenConstrForce	Forces the general constraint into (1) or out of (0) the computed IIS

Solution quality attributes:

Attribute name	Description
MaxVio	Maximum (unscaled) violation
BoundVio	Maximum (unscaled) bound violation
BoundSVio	Maximum (scaled) bound violation
BoundVioIndex	Index of variable with the largest (unscaled) bound violation
BoundSVioIndex	Index of variable with the largest (scaled) bound violation
BoundVioSum	Sum of (unscaled) bound violations
BoundSVioSum	Sum of (scaled) bound violations
ConstrVio	Maximum (unscaled) constraint violation
ConstrSVio	Maximum (scaled) constraint violation
ConstrVioIndex	Index of constraint with the largest (unscaled) violation
ConstrSVioIndex	Index of constraint with the largest (scaled) violation
ConstrVioSum	Sum of (unscaled) constraint violations
ConstrSVioSum	Sum of (scaled) constraint violations
ConstrResidual	Maximum (unscaled) primal constraint error
ConstrSResidual	Maximum (scaled) primal constraint error
ConstrResidualIndex	Index of constraint with the largest (unscaled) primal constraint error
ConstrSResidualIndex	Index of constraint with the largest (scaled) primal constraint error
ConstrResidualSum	Sum of (unscaled) primal constraint errors
ConstrSResidualSum	Sum of (scaled) primal constraint errors

DualVio	Maximum (unscaled) reduced cost violation
DualSVio	Maximum (scaled) reduced cost violation
DualVioIndex	Index of variable with the largest (unscaled) reduced cost violation
DualSVioIndex	Index of variable with the largest (scaled) reduced cost violation
DualVioSum	Sum of (unscaled) reduced cost violations
DualSVioSum	Sum of (scaled) reduced cost violations
DualResidual	Maximum (unscaled) dual constraint error
DualSResidual	Maximum (scaled) dual constraint error
DualResidualIndex	Index of variable with the largest (unscaled) dual constraint error
DualSResidualIndex	Index of variable with the largest (scaled) dual constraint error
DualResidualSum	Sum of (unscaled) dual constraint errors
DualSResidualSum	Sum of (scaled) dual constraint errors
ComplVio	Maximum complementarity violation
ComplVioIndex	Index of variable with the largest complementarity violation
ComplVioSum	Sum of complementarity violations
IntVio	Maximum integrality violation
IntVioIndex	Index of variable with the largest integrality violation
IntVioSum	Sum of integrality violations

Multi-objective attributes:

Attribute name	Description
ObjN	Objectives of multi-objectives
ObjNCon	Constant terms of multi-objectives
ObjNPriority	Priorities of multi-objectives
ObjNWeight	Weights of multi-objectives
ObjNRelTol	Relative tolerances of multi-objectives
ObjNAbsTol	Absolute tolerances of multi-objectives
ObjNVal	Objective value of multi-objectives solutions
ObjNName	Names of multi-objectives
NumObj	Number of multi-objectives

Multi-scenario attributes:

Attribute name	Description
ScenNLB	Lower bound changes of current scenario in multi-scenario model
ScenNUB	Upper bound changes of current scenario in multi-scenario model
ScenNObj	Objective coefficient changes of current scenario in multi-scenario model
ScenNRHS	Right-hand side changes of current scenario in multi-scenario model
ScenNName	Name of current scenario in multi-scenario model
ScenNObjBound	Objective bound of current scenario in multi-scenario model
ScenNObjVal	Objective value for current solution of current scenario in multi-scenario model
ScenNX	Value in the current solution of current scenario in multi-scenario model
NumScenarios	Number of scenarios

Batch attributes:

Attribute name	Description
BatchErrorCode	Last error code received from the Cluster Manager
BatchErrorMessage	Last error message received from the Cluster Manager
BatchID	ID of the remote batch
BatchStatus	Status of the batch

12.1 Model Attributes

These are model attributes, meaning that they are associated with the overall model (as opposed to being associated with a particular variable or constraint of the model). You should use one of the various `get` routines to retrieve the value of an attribute. These are described at the beginning of this section. For the object-oriented interfaces, model attributes are retrieved by invoking the `get` method on the model object itself. For attributes that can be modified directly by the user, you can use one of the various `set` methods.

Attempting to query an attribute that is not available will produce an error. In C, the attribute query routine will return a `GRB_ERROR_DATA_NOT_AVAILABLE` error code. The object-oriented interfaces will throw an exception.

Additional model attributes can be found in the Quality Attributes, Multi-objective Attributes, and Multi-Scenario Attributes sections.

NumConstrs

Type: int
Modifiable: No

The number of linear constraints in the model.

For examples of how to query or modify attributes, refer to our Attribute Examples.

NumVars

Type: int
Modifiable: No

The number of decision variables in the model.
For examples of how to query or modify attributes, refer to our Attribute Examples.

NumSOS

Type: int
Modifiable: No
The number of Special Ordered Set (SOS) constraints in the model.
For examples of how to query or modify attributes, refer to our Attribute Examples.

NumQConstrs

Type: int
Modifiable: No
The number of quadratic constraints in the model.
For examples of how to query or modify attributes, refer to our Attribute Examples.

NumGenConstrs

Type: int
Modifiable: No
The number of general constraints in the model.
For examples of how to query or modify attributes, refer to our Attribute Examples.

NumNZs

Type: int
Modifiable: No
The number of non-zero coefficients in the linear constraints of the model. For models with more than 2 billion non-zero coefficients use DNumNZs.
For examples of how to query or modify attributes, refer to our Attribute Examples.

DNumNZs

Type: double
Modifiable: No
The number of non-zero coefficients in the linear constraints of the model. This attribute is provided in double precision format to accurately count the number of non-zeros in models that contain more than 2 billion non-zero coefficients.
For examples of how to query or modify attributes, refer to our Attribute Examples.

NumQNZs

Type: int
Modifiable: No
The number of terms in the lower triangle of the Q matrix in the quadratic objective.
For examples of how to query or modify attributes, refer to our Attribute Examples.

NumQCNZs

Type: int
Modifiable: No

The number of non-zero coefficients in the quadratic constraints.

For examples of how to query or modify attributes, refer to our Attribute Examples.

NumIntVars

Type: int
Modifiable: No

The number of integer variables in the model. This includes both binary variables and general integer variables.

For examples of how to query or modify attributes, refer to our Attribute Examples.

NumBinVars

Type: int
Modifiable: No

The number of binary variables in the model.

For examples of how to query or modify attributes, refer to our Attribute Examples.

NumPWLObjVars

Type: int
Modifiable: No

The number of variables in the model with piecewise-linear objective functions. You can query the function for a specific variable using the appropriate `getPWLObj` method for your language (in C, C++, C#, Java, and Python).

For examples of how to query or modify attributes, refer to our Attribute Examples.

ModelName

Type: string
Modifiable: Yes

The name of the model. The name has no effect on Gurobi algorithms. It is output in the Gurobi log file when a model is solved, and when a model is written to a file.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ModelSense

Type: int
Modifiable: Yes

Optimization sense. The default 1 value indicates that the objective is to minimize the objective. Setting this attribute to -1 changes the sense to maximization.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjCon

Type: double
Modifiable: Yes

A constant value that is added into the model objective. The default value is 0.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Fingerprint

Type: int
Modifiable: No

A hash value computed on model data and attributes that can influence the optimization process. The intent is that models that differ in any meaningful way will have different fingerprints (almost always).

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjVal

Type: double
Modifiable: No

The objective value for the current solution. If the model was solved to optimality, then this attribute gives the optimal objective value.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjBound

Type: double
Modifiable: No

The best known bound on the optimal objective. When solving a MIP model, the algorithm maintains both a lower bound and an upper bound on the optimal objective value. For a minimization model, the upper bound is the objective of the best known feasible solution, while the lower bound gives a bound on the best possible objective.

In contrast to ObjBoundC, this attribute takes advantage of objective integrality information to round to a tighter bound. For example, if the objective is known to take an integral value and the current best bound is 1.5, `ObjBound` will return 2.0 while `ObjBoundC` will return 1.5.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjBoundC

Type: double
Modifiable: No

The best known bound on the optimal objective. When solving a MIP model, the algorithm maintains both a lower bound and an upper bound on the optimal objective value. For a minimization model, the upper bound is the objective of the best known feasible solution, while the lower bound gives a bound on the best possible objective.

In contrast to ObjBound, this attribute does not take advantage of objective integrality information to round to a tighter bound. For example, if the objective is known to take an integral value and the current best bound is 1.5, `ObjBound` will return 2.0 while `ObjBoundC` will return 1.5.

For examples of how to query or modify attributes, refer to our Attribute Examples.

PoolObjBound

Type: double
Modifiable: No

Bound on the objective of undiscovered MIP solutions. The MIP solver stores solutions that it finds during the MIP search, but it only provides quality guarantees for those whose objective is at least as good as `PoolObjBound`. Specifically, further exploration of the MIP search tree will not find solutions whose objective is better than `PoolObjBound`.

The difference between `PoolObjBound` and ObjBound is that the former gives an objective bound for undiscovered solutions, while the latter gives a bound for any solution. Note that `PoolObjBound` and `ObjBound` can only have different values if parameter PoolSearchMode is set to 2.

Please consult the section on Solution Pools for a more detailed discussion of this topic.

For examples of how to query or modify attributes, refer to our Attribute Examples.

PoolObjVal

Type: double
Modifiable: No

This attribute is used to query the objective value of the k-th solution stored in the pool of feasible solutions found so far for the problem. You set k using the SolutionNumber parameter.

The number of stored solutions can be queried using the SolCount attribute.

Please consult the section on Solution Pools for a more detailed discussion of this topic.

For examples of how to query or modify attributes, refer to our Attribute Examples.

MIPGap

Type: double
Modifiable: No

Current relative MIP optimality gap; computed as $|ObjBound - ObjVal|/|ObjVal|$ (where `ObjBound` and `ObjVal` are the MIP objective bound and incumbent solution objective, respectively. Returns `GRB_INFINITY` when an incumbent solution has not yet been found, when no objective bound is available, or when the current incumbent objective is 0.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Runtime

Type: double
Modifiable: No

Runtime for the most recent optimization (in seconds). Note that all times reported by the Gurobi Optimizer are wall-clock times.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Work

Type: double
Modifiable: No

Work spent on the most recent optimization. In contrast to Runtime, work is deterministic, meaning that you will get exactly the same result every time provided you solve the same model on the same hardware with the same parameter and attribute settings. The units on this metric are arbitrary. One work unit corresponds very roughly to one second on a single thread, but this greatly depends on the hardware on which Gurobi is running and the model that is being solved.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Status

Type: int
Modifiable: No

Current optimization status for the model. Status values are described in the Status Code section.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SolCount

Type: int
Modifiable: No

Number of stored solutions from the most recent optimization.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IterCount

Type: double
Modifiable: No

Number of simplex iterations performed during the most recent optimization.

For examples of how to query or modify attributes, refer to our Attribute Examples.

BarIterCount

Type: int
Modifiable: No

Number of barrier iterations performed during the most recent optimization.

For examples of how to query or modify attributes, refer to our Attribute Examples.

NodeCount

Type: double
Modifiable: No

Number of branch-and-cut nodes explored in the most recent optimization.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConcurrentWinMethod

Type: int
Modifiable: No

This attribute is used to query the winning method after a continuous problem has been solved with concurrent optimization. In this case it returns the corresponding method identifier 0 (for primal Simplex), 1 (for dual Simplex), or 2 (for Barrier). In all other cases -1 is returned.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IsMIP

Type: int
Modifiable: No

Indicates whether the model is a MIP. Note that any discrete elements make the model a MIP. Discrete elements include binary, integer, semi-continuous, semi-integer variables, SOS constraints, and general constraints. In addition, models having multiple objectives or multiple scenarios are considered as MIP models, even when all variables are continuous and all constraints are linear.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IsQP

Type: int
Modifiable: No

Indicates whether the model is a quadratic programming problem. Note that a model with both a quadratic objective and quadratic constraints is classified as a QCP, not a QP.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IsQCP

Type: int
Modifiable: No

Indicates whether the model has quadratic constraints.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IsMultiObj

Type: int
Modifiable: No

Indicates whether the model has multiple objectives.

Note that the case where the model has a single objective (NumObj = 1) is slightly ambiguous. If you used `setObjectiveN` to set your objective, or if you set any of the multi-objective attributes (e.g., ObjNPriority), then the model is considered to be a multi-objective model. Otherwise, it is not.

To reset a multi-objective model back to a single objective model, you should set the `NumObj` attribute to 0, call model update, and then set a new single objective.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISMinimal

Type: int
Modifiable: No

Indicates whether the current Irreducible Inconsistent Subsystem (IIS) is minimal. This attribute is only available after you have computed an IIS on an infeasible model. It will normally take value 1, but it may take value 0 if the IIS computation was stopped early (e.g., due to a time limit or user interrupt).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MaxCoeff

Type: double
Modifiable: No

Maximum matrix coefficient (in absolute value) in the linear constraint matrix.

For examples of how to query or modify attributes, refer to our Attribute Examples.

MinCoeff

Type: double
Modifiable: No

Minimum non-zero matrix coefficient (in absolute value) in the linear constraint matrix.

For examples of how to query or modify attributes, refer to our Attribute Examples.

MaxBound

Type: double
Modifiable: No

Maximum (finite) variable bound.

For examples of how to query or modify attributes, refer to our Attribute Examples.

MinBound

Type: double
Modifiable: No

Minimum (non-zero) variable bound.

For examples of how to query or modify attributes, refer to our Attribute Examples.

MaxObjCoeff

Type: double
Modifiable: No

Maximum linear objective coefficient (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MinObjCoeff

Type: double
Modifiable: No

Minimum (non-zero) linear objective coefficient (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MaxRHS

Type: double
Modifiable: No

Maximum linear constraint right-hand side value (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MinRHS

Type: double
Modifiable: No

Minimum (non-zero) linear constraint right-hand side value (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MaxQCCoeff

Type: double
Modifiable: No

Maximum coefficient in the quadratic part of all quadratic constraint matrices (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MinQCCoeff

Type: double
Modifiable: No

Minimum (non-zero) coefficient in the quadratic part of all quadratic constraint matrices (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MaxQCLCoeff

Type: double
Modifiable: No

Maximum coefficient in the linear part of all quadratic constraint matrices (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MinQCLCoeff

Type: double
Modifiable: No

Minimum (non-zero) coefficient in the linear part of all quadratic constraint matrices (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MaxQCRHS

Type: double
Modifiable: No

Maximum quadratic constraint right-hand side value (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MinQCRHS

Type: double
Modifiable: No

Minimum (non-zero) quadratic constraint right-hand side value (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MaxQObjCoeff

Type: double
Modifiable: No

Maximum coefficient of the quadratic terms in the objective (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

MinQObjCoeff

Type: double
Modifiable: No

Minimum (non-zero) coefficient of the quadratic terms in the objective (in absolute value).

For examples of how to query or modify attributes, refer to our Attribute Examples.

OpenNodeCount

Type: double
Modifiable: No

Number of open branch-and-cut nodes at the end of the most recent optimization. An open node is one that has been created but not yet explored.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Kappa

Type: double
Modifiable: No

Estimated condition number for the current LP basis matrix. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

KappaExact

Type: double
Modifiable: No

Exact condition number for the current LP basis matrix. Only available for basic solutions. The exact condition number is much more expensive to compute than the estimate that you get from the Kappa attribute. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

FarkasProof

Type: double
Modifiable: No

Together, attributes FarkasDual and FarkasProof provide a certificate of infeasibility for the given infeasible problem. Specifically, FarkasDual provides a vector λ that can be used to form the following inequality from the original constraints that is trivially infeasible within the bounds of the variables:

$$\lambda^t A x \leq \lambda^t b.$$

This inequality is valid even if the original constraints have a mix of less-than and greater-than senses because $\lambda_i \geq 0$ if the i-th constraint has a $\leq$ sense and $\lambda_i \leq 0$ if the i-th constraint has a $\geq$ sense.

Let

$$\bar{a} := \lambda^t A$$

be the coefficients of this inequality and

$$\bar{b} := \lambda^t b$$

be its right hand side. With L_j and U_j being the lower and upper bounds of the variables x_j, we have $\bar{a}_j \geq 0$ if $U_j = \infty$, and $\bar{a}_j \leq 0$ if $L_j = -\infty$.

The minimum violation of the Farkas constraint is achieved by setting $x^*_j := L_j$ for $\bar{a}_j > 0$ and $x^*_j := U_j$ for $\bar{a}_j < 0$. Then, we can calculate the minimum violation as

$$\beta := \bar{a}^t x^* - \bar{b} = \sum_{j:\bar{a}_j > 0} \bar{a}_j L_j + \sum_{j:\bar{a}_j < 0} \bar{a}_j U_j - \bar{b}$$

where $\beta > 0$.

The FarkasProof attribute gives the value of β.

These attributes are only available when parameter InfUnbdInfo is set to 1.

For examples of how to query or modify attributes, refer to our Attribute Examples.

TuneResultCount

Type: int
Modifiable: No

After the tuning tool has been run, this attribute reports the number of parameter sets that were stored. This value will be zero if no improving parameter sets were found, and its upper bound is determined by the TuneResults parameter.

For examples of how to query or modify attributes, refer to our Attribute Examples.

NumStart

Type: int
Modifiable: Yes

Number of MIP starts in the model. Decreasing this attribute will discard existing MIP starts. Increasing it will create new MIP starts (initialized to `undefined`).

You can use the StartNumber parameter to query or modify start values for different MIP starts, or to append a new one. The value of `StartNumber` should always be less than `NumStart`.

For examples of how to query or modify attributes, refer to our Attribute Examples.

LicenseExpiration

Type: int
Modifiable: No

License expiration date. The format is YYYYMMDD, so for example if the license currently in use expired on July 20, 2018, the result would be 20180720. If the license has no expiration date, the result will be 99999999.

This attribute is available for node licenses and for clients of a Gurobi Compute Server. Unfortunately, this attribute isn't available for clients of a Gurobi Token Server.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.2 Variable Attributes

These are variable attributes, meaning that they are associated with specific variables in the model. You should use one of the various `get` routines to retrieve the value of an attribute. These are described at the beginning of this section. For the object-oriented interfaces, variable attributes are retrieved by invoking the `get` method on a variable object. For attributes that can be modified directly by the user, you can use one of the various `set` methods.

Attempting to query an attribute that is not available will produce an error. In C, the attribute query routine will return a `GRB_ERROR_DATA_NOT_AVAILABLE` error code. The object-oriented interfaces will throw an exception.

Additional variable attributes can be found in the Multi-objective Attributes and Multi-Scenario Attributes sections.

LB

Type: double
Modifiable: Yes

Variable lower bound. Note that any value less than or equal to `-1e20` is treated as negative infinity.

For examples of how to query or modify attributes, refer to our Attribute Examples.

UB

Type: double
Modifiable: Yes

Variable upper bound. Note that any value greater than or equal to `1e20` is treated as infinite.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Obj

Type: double
Modifiable: Yes

Linear objective coefficient. In our object-oriented interfaces, you typically use the `setObjective` method to set the objective, but this attribute provides an alternative for setting or modifying linear objective terms.

Note that this attribute interacts with our piecewise-linear objective feature. If you set a piecewise-linear objective function for a variable, that will automatically set the `Obj` attribute to zero. Similarly, if you set the `Obj` attribute for a variable, that will automatically delete any previously specified piecewise-linear objective.

For examples of how to query or modify attributes, refer to our Attribute Examples.

VarName

Type: string
Modifiable: Yes

Variable name.

For examples of how to query or modify attributes, refer to our Attribute Examples.

VTag

Type: string
Modifiable: Yes

Tag for variables.

If you will be retrieving the solution to your model in JSON format, you might define a tag for every variable that you plan to retrieve solution information for. Each variable tag must be unique, and if any tag is used (variable tag, constraint tag, quadratic constraint tag) only tagged elements will appear in the JSON solution string. Tags must consist of printable US-ASCII characters. Using extended characters or escaped characters will result in an error. The maximum supported length for a tag is 10240 characters.

For examples of how to query or modify attributes, refer to our Attribute Examples.

VType

Type: char
Modifiable: Yes

Variable type (`'C'` for continuous, `'B'` for binary, `'I'` for integer, `'S'` for semi-continuous, or `'N'` for semi-integer). Binary variables must be either 0 or 1. Integer variables can take any integer value between the specified lower and upper bounds. Semi-continuous variables can take any value between the specified lower and upper bounds, or a value of zero. Semi-integer variables can take any integer value between the specified lower and upper bounds, or a value of zero.

Refer to this section for more information on variable types.

For examples of how to query or modify attributes, refer to our Attribute Examples.

X

Type: double
Modifiable: No

Variable value in the current solution.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Xn

Type: double
Modifiable: No

The variable value in a sub-optimal MIP solution. Use parameter SolutionNumber to indicate which alternate solution to retrieve. Solutions are sorted in order of worsening objective value. Thus, when `SolutionNumber` is 1, `Xn` returns the second-best solution found. When `SolutionNumber` is equal to its default value of 0, querying attribute `Xn` is equivalent to querying attribute `X`.

The number of sub-optimal solutions found during the MIP search will depend on the values of a few parameters. The most important of these are PoolSolutions, PoolSearchMode, and PoolGap. Please consult the section on Solution Pools for a more detailed discussion of this topic.

Only available for MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

RC

Type: double
Modifiable: No

The reduced cost in the current solution. Only available for convex, continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

BarX

Type: double
Modifiable: No

The variable value in the best barrier iterate (before crossover). Only available when the barrier algorithm was selected.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Start

Type: double
Modifiable: Yes

The current MIP start vector. The MIP solver will attempt to build an initial solution from this vector when it is available. Note that the start can be partially populated -- the MIP solver will attempt to fill in values for missing start values. If you wish to leave the start value for a variable undefined, you can either avoid setting the `Start` attribute for that variable, or you can set it to a special `undefined` value (`GRB_UNDEFINED` in C and C++, or `GRB.UNDEFINED` in Java, .NET, and Python).

If the Gurobi MIP solver log indicates that your MIP start didn't produce a new incumbent solution, note that there can be multiple explanations. One possibility is that your MIP start is infeasible. Another, more common possibility is that one of the Gurobi heuristics found a solution that is as good as the solution produced by the MIP start, so the MIP start solution was cut off. Finally, if you specified a partial MIP start, it is possible that the limited MIP exploration done on this partial start was insufficient to find a new incumbent solution. You can try setting the StartNodeLimit parameter to a larger value if you want Gurobi to work harder to try to complete the partial start.

If you solve a sequence of models, where one is built by modifying the previous one, and if you don't provide a MIP start, then Gurobi will try to construct one automatically from the solution of the previous model. If you don't want it to try this, you should reset the model before starting the subsequent solve. If you provided a MIP start but would prefer to use the previous solution as the start instead, you should clear your start (by setting the `Start` attribute to `undefined` for all variables).

If you have multiple start vectors, you can provide them to Gurobi by using the `Start` attribute in combination with the NumStart attribute and the StartNumber parameter. Specifically, use the `NumStart` attribute to indicate how many start vectors you will supply. Then set the `StartNumber` parameter to a value between 0 and `NumStart`-1 to indicate which start you are supplying. For each value of `StartNumber`, populate the `Start` attribute to supply that start. Gurobi will use all of the provided starts. As an alternative, you can append new MIP start vectors to your model by setting the StartNumber parameter to `-1`. In this case, whenever you read a MIP start, or use a function to set a MIP start value for a set of variables, a new MIP start will be created, the parameter NumStart will be increased, and any unspecified variable will be left as `undefined`.

If you want to diagnose an infeasible MIP start, you can try fixing the variables in the model to their values in your MIP start (by setting their lower and upper bound attributes). If the resulting MIP model is infeasible, you can then compute an IIS on this model to get additional information that should help to identify the cause of the infeasibility.

Only affects MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

VarHintVal

Type: double
Modifiable: Yes

A set of user hints. If you know that a variable is likely to take a particular value in high quality solutions of a MIP model, you can provide that value as a hint. You can also (optionally) provide information about your level of confidence in a hint with the VarHintPri attribute.

The Gurobi MIP solver will use these variable hints in a number of different ways. Hints will affect the heuristics that Gurobi uses to find feasible solutions, and the branching decisions that Gurobi makes to explore the MIP search tree. In general, high quality hints should produce high quality MIP solutions faster. In contrast, low quality hints will lead to some wasted effort, but shouldn't lead to dramatic performance degradations.

Variables hints and MIP starts are similar in concept, but they behave in very different ways. If you specify a MIP start, the Gurobi MIP solver will try to build a single feasible solution from the provided set of variable values. If you know a solution, you should use a MIP start to provide it to the solver. In contrast, variable hints provide guidance to the MIP solver that affects the entire solution process. If you have a general sense of the likely values for variables, you should provide them through variable hints.

If you wish to leave the hint value for a variable undefined, you can either avoid setting the `VarHintVal` attribute for that variable, or you can set it to a special `undefined` value (`GRB_UNDEFINED` in C and C++, `GRB.UNDEFINED` in Java, .NET, and Python, `NA` in R or `nan` in Matlab).

Note that deleting variables from your model will cause several attributes to be discarded (variable hints and branch priorities). If you'd like them to persist, your program will need to repopulate them after deleting the variables and making a subsequent model update call.

Only affects MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

VarHintPri

Type: int
Modifiable: Yes

Priorities on user hints. After providing variable hints through the VarHintVal attribute, you can optionally also provide hint priorities to give an indication of your level of confidence in your hints.

Hint priorities are relative. If you are more confident in the hint value for one variable than for another, you simply need to set a larger priority value for the more solid hint. The default hint priority for a variable is 0.

Please refer to the VarHintVal discussion for more details on the role of variable hints.

Only affects MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

BranchPriority

Type: int
Modifiable: Yes

Variable branching priority. The value of this attribute is used as the primary criterion for selecting a fractional variable for branching during the MIP search. Variables with larger values always take priority over those with smaller values. Ties are broken using the standard branch variable selection criteria. The default variable branch priority value is zero.

Note that deleting variables from your model will cause several attributes to be discarded (variable hints and branch priorities). If you'd like them to persist, your program will need to repopulate them after deleting the variables and making a subsequent model update call.

Only affects MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Partition

Type: int
Modifiable: Yes

Variable partition. The MIP solver can perform a solution improvement heuristic using user-provided partition information. The provided partition number can be positive, which indicates that the variable should be included when the correspondingly numbered sub-MIP is solved, 0 which indicates that the variable should be included in every sub-MIP, or -1 which indicates that the variable should not be included in any sub-MIP. Variables that are not included in the sub-MIP are fixed to their values in the current incumbent solution. By default, all variables start with a value of -1.

To give an example, imagine you are solving a model with 400 variables and you set the partition attribute to -1 for variables 0-99, 0 for variables 100-199, 1 for variables 200-299, and 2 for variables 300-399. The heuristic would solve two sub-MIP models: sub-MIP 1 would fix variables 0-99 and 300-399 to their values in the incumbent and solve for the rest, while sub-MIP 2 would fix variables 0-99 and 200-299.

Use the PartitionPlace parameter to control where the partition heuristic runs.

Only affects MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

VBasis

Type: int
Modifiable: Yes

The status of a given variable in the current basis. Possible values are 0 (basic), -1 (non-basic at lower bound), -2 (non-basic at upper bound), and -3 (super-basic). Note that, if you wish to specify an advanced starting basis, you must set basis status information for all constraints and variables in the model. Only available for basic solutions.

Note that if you provide a valid starting extreme point, either through `PStart, DStart`, or through `VBasis, CBasis`, then LP presolve will be disabled by default. For models where presolve greatly reduces the problem size, this might hurt performance. For presolve to be enabled, the parameter LPWarmStart must be set to 2.

For examples of how to query or modify attributes, refer to our Attribute Examples.

PStart

Type: double
Modifiable: Yes

Primal start vector.

For LP models, this defines the current simplex start vector. If you set `PStart` values for every variable in the model and DStart values for every constraint, then simplex will use those values to compute a warm start basis. Note that you'll get much better performance if you warm start your linear program using a simplex basis (using VBasis and CBasis). The `PStart` attribute should only be used in situations where you don't have a basis or you don't want to disable presolve.

For non-convex (MI)QCP and (MI)NLP models, this defines the starting point for certain primal heuristics. If you set `PStart` values for every variable in the model, then these heuristics will be more likely to converge to feasible points in the vicinity of the given starting point.

For other problem types, the `Pstart` values will be ignored.

If you'd like to provide a *feasible* starting solution for MIP, non-convex (MI)QCP, or (MI)NLP models, you should input it using the Start attribute.

Note that any model modifications which are pending or are made after setting `PStart` (adding variables or constraints, changing coefficients, etc.) will discard the start. You should only set this attribute after you are done modifying your model. If you'd like to retract a previously specified start, set any `PStart` value to `GRB_UNDEFINED`.

Note that if you provide a valid starting extreme point, either through `PStart`, `DStart`, or through `VBasis`, `CBasis`, then LP presolve will be disabled by default. For models where presolve greatly reduces the problem size, this might hurt performance. For presolve to be enabled, the parameter LPWarmStart must be set to 2.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISLB

Type: int
Modifiable: No

For an infeasible model, indicates whether the lower bound participates in the computed Irreducible Inconsistent Subsystem (IIS). Note that the bounds for a binary variable are considered to be implicit in the variable type and will never participate in an IIS.

Only available after you have computed an IIS.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISLBForce

Type: int
Modifiable: Yes

When computing an Irreducible Inconsistent Subsystem (IIS) for an infeasible model, indicates whether the variable lower bound should be included or excluded from the IIS.

The default value of -1 lets the IIS algorithm decide.

If the attribute is set to 0, the bound is not eligible for inclusion in the IIS.

If the attribute is set to 1, the bound is included in the IIS and the IIS algorithm never considers the possibility of removing it.

Note that setting this attribute to 0 may make the resulting subsystem feasible (or consistent), which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

See the Model.computeIIS documentation for more details.

This attribute is ignored for LPs.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISUB

Type: int
Modifiable: No

For an infeasible model, indicates whether the upper bound participates in the computed Irreducible Inconsistent Subsystem (IIS). Note that the bounds for a binary variable are considered to be implicit in the variable type and will never participate in an IIS.

Only available after you have computed an IIS.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISUBForce

Type: int
Modifiable: Yes

When computing an Irreducible Inconsistent Subsystem (IIS) for an infeasible model, indicates whether the variable upper bound should be included or excluded from the IIS.

The default value of -1 lets the IIS algorithm decide.

If the attribute is set to 0, the bound is not eligible for inclusion in the IIS.

If the attribute is set to 1, the bound is included in the IIS and the IIS algorithm never considers the possibility of removing it.

Note that setting this attribute to 0 may make the resulting subsystem feasible (or consistent), which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

See the Model.computeIIS documentation for more details.

This attribute is ignored for LPs.

For examples of how to query or modify attributes, refer to our Attribute Examples.

PoolIgnore

Type: int
Modifiable: Yes

When solving a MIP model, the Gurobi Optimizer maintains a solution pool that contains the best solutions found during the search. The `PoolIgnore` attribute allows you to discard some solutions. Specifically, if multiple solutions differ only in variables where `PoolIgnore` is set to 1, only the solution with the best objective will be kept in the pool. The default value for the attribute is 0, meaning that the variable should be used to distinguish solutions.

This attribute is particularly helpful when used in conjunction with the PoolSearchMode parameter. By identifying variables that do not capture meaningful differences between solutions, you can make sure that the pool contains some interesting variety.

For examples of how to query or modify attributes, refer to our Attribute Examples.

PWLObjCvx

Type: int
Modifiable: No

Indicates whether a variable has a convex piecewise-linear objective. Returns 0 if the piecewise-linear objective function on the variable is non-convex. Returns 1 if the function is convex, or if the objective function on the variable is linear.

This attribute is useful for isolating the particular variable that caused a continuous model with a piecewise-linear objective function to become a MIP.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SAObjLow

Type: double
Modifiable: No

Objective coefficient sensitivity information: smallest objective coefficient value at which the current optimal basis would remain optimal. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SAObjUp

Type: double
Modifiable: No

Objective coefficient sensitivity information: largest objective coefficient value at which the current optimal basis would remain optimal. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SALBLow

Type: double
Modifiable: No

Lower bound sensitivity information: smallest lower bound value at which the current optimal basis would remain optimal. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SALBUp

Type: double
Modifiable: No

Lower bound sensitivity information: largest lower bound value at which the current optimal basis would remain optimal. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SAUBLow

Type: double
Modifiable: No

Upper bound sensitivity information: smallest upper bound value at which the current optimal basis would remain optimal. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SAUBUp

Type: double
Modifiable: No

Upper bound sensitivity information: largest upper bound value at which the current optimal basis would remain optimal. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

UnbdRay

Type: double
Modifiable: No

Unbounded ray (for unbounded linear models only). Provides a vector that, when added to any feasible solution, yields a new solution that is also feasible but improves the objective. Only available when parameter InfUnbdInfo is set to 1.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.3 Linear Constraint Attributes

These are linear constraint attributes, meaning that they are associated with specific linear constraints in the model. You should use one of the various `get` routines to retrieve the value of an attribute. These are described at the beginning of this section. For the object-oriented interfaces, linear constraint attributes are retrieved by invoking the `get` method on a constraint object. For attributes that can be modified directly by the user, you can use one of the various `set` methods.

Attempting to query an attribute that is not available will produce an error. In C, the attribute query routine will return a `GRB_ERROR_DATA_NOT_AVAILABLE` error code. The object-oriented interfaces will throw an exception.

Additional linear constraint attributes can be found in the Multi-Scenario Attributes section.

Sense

Type: char
Modifiable: Yes

Constraint sense (`'<'`, `'>'`, or `'='`).

For examples of how to query or modify attributes, refer to our Attribute Examples.

RHS

Type: double
Modifiable: Yes

Constraint right-hand side. Note that a less-than-or-equal constraint with a RHS value of `1e20` or larger, or a greater-than-or-equal constraint with RHS value of `-1e20` or smaller, will be treated

as always being satisfied. Equality constraints with very large RHS values can lead to numerical issues, so these should be avoided.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrName

Type: string
Modifiable: Yes

Constraint name.

For examples of how to query or modify attributes, refer to our Attribute Examples.

CTag

Type: string
Modifiable: Yes

Tag for constraints.

If you will be retrieving the solution to your model in JSON format, you might define a tag for every constraint that you plan to retrieve solution information for. Each constraint tag must be unique, and if any tag is used (variable tag, constraint tag, quadratic constraint tag) only tagged elements will appear in the JSON solution string. Tags must consist of printable US-ASCII characters. Using extended characters or escaped characters will result in an error. The maximum supported length for a tag is 10240 characters.

Note that constraint tags are only allowed for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Pi

Type: double
Modifiable: No

The constraint dual value in the current solution (also known as the *shadow price*).

Given a linear programming problem

$$\begin{array}{ll} \text{minimize} & c'x \\ \text{subject to} & Ax \geq b \\ & x \geq 0 \end{array}$$

and a corresponding dual problem

$$\begin{array}{ll} \text{maximize} & b'y \\ \text{subject to} & A'y \leq c \\ & y \geq 0 \end{array}$$

the `Pi` attribute returns y.

Of course, not all models fit this canonical form. In general, dual values have the following properties:

- Dual values for $\geq$ constraints are ≥ 0.
- Dual values for $\leq$ constraints are ≤ 0.

- Dual values for = constraints are unconstrained.

For models with a maximization sense, the senses of the dual values are reversed: the dual is ≥ 0 for a $\leq$ constraint and ≤ 0 for a $\geq$ constraint.

Note that constraint dual values for linear constraints of QCP models are only available when the QCPDual parameter is set to 1. To query the duals of the quadratic constraints in a QCP model, see QCPi.

Only available for convex, continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Slack

Type: double
Modifiable: No

The constraint slack in the current solution.

For examples of how to query or modify attributes, refer to our Attribute Examples.

CBasis

Type: int
Modifiable: Yes

The status of a given linear constraint in the current basis. Possible values are 0 (basic) or -1 (non-basic). A constraint is basic when its slack variable is in the simplex basis. Note that, if you wish to specify an advanced starting basis, you must set basis status information for all constraints and variables in the model. Only available for basic solutions.

Note that if you provide a valid starting extreme point, either through `PStart`, `DStart`, or through `VBasis`, `CBasis`, then LP presolve will be disabled by default. For models where presolve greatly reduces the problem size, this might hurt performance. For presolve to be enabled, the parameter LPWarmStart must be set to 2.

For examples of how to query or modify attributes, refer to our Attribute Examples.

DStart

Type: double
Modifiable: Yes

The current simplex start vector. If you set `DStart` values for every linear constraint in the model and PStart values for every variable, then simplex will use those values to compute a warm start basis. If you'd like to retract a previously specified start, set any `DStart` value to `GRB_UNDEFINED`.

Note that any model modifications which are pending or are made after setting `DStart` (adding variables or constraints, changing coefficients, etc.) will discard the start. You should only set this attribute after you are done modifying your model.

Note also that you'll get much better performance if you warm start your linear program from a simplex basis (using VBasis and CBasis). The `DStart` attribute should only be used in situations where you don't have a basis or you don't want to disable presolve.

If you'd like to provide a feasible starting solution for a MIP model, you should input it using the Start attribute.

Only affects LP models; it will be ignored for QP, QCP, or MIP models.

Note that if you provide a valid starting extreme point, either through `PStart`, `DStart`, or through `VBasis`, `CBasis`, then LP presolve will be disabled by default. For models where presolve greatly reduces the problem size, this might hurt performance. For presolve to be enabled, the parameter LPWarmStart must be set to 2.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Lazy

Type: int
Modifiable: Yes

Determines whether a linear constraint is treated as a *lazy constraint* or a *user cut*.

At the beginning of the MIP solution process, any constraint whose `Lazy` attribute is set to 1, 2, or 3 (the default value is 0) is treated as a lazy constraint; it is removed from the model and placed in the lazy constraint pool. Lazy constraints remain inactive until a feasible solution is found, at which point the solution is checked against the lazy constraint pool. If the solution violates any lazy constraints, the solution is discarded and one or more of the violated lazy constraints are pulled into the active model.

Larger values for this attribute cause the constraint to be pulled into the model more aggressively. With a value of 1, the constraint can be used to cut off a feasible solution, but it won't necessarily be pulled in if another lazy constraint also cuts off the solution. With a value of 2, all lazy constraints that are violated by a feasible solution will be pulled into the model. With a value of 3, lazy constraints that cut off the relaxation solution at the root node are also pulled in.

Any constraint whose Lazy attribute is set to -1 is treated as a user cut; it is removed from the model and placed in the user cut pool. User cuts may be added to the model at any node in the branch-and-cut search tree to cut off relaxation solutions.

The main difference between user cuts and lazy constraints is that that the former are not allowed to cut off integer-feasible solutions. In other words, they are redundant for the MIP model, and the solver is free to decide whether or not to use them to cut off relaxation solutions. The hope is that adding them speeds up the overall solution process. Lazy constraints have no such restrictions. They are essential to the model, and the solver is forced to apply them whenever a solution would otherwise not satisfy them.

Note that deleting constraints from your model will cause this attribute to be discarded. If you'd like it to persist, your program will need to repopulate it after deleting the constraints and making a subsequent model update call.

Note that no corresponding attribute is available for other constraint types (quadratic, SOS, or general constraints). This attribute only affects MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISConstr

Type: int
Modifiable: No

For an infeasible model, indicates whether the linear constraint participates in the computed Irreducible Inconsistent Subsystem (IIS). Only available after you have computed an IIS.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISConstrForce

Type: int
Modifiable: Yes

When computing an Irreducible Inconsistent Subsystem (IIS) for an infeasible model, indicates whether the linear constraint should be included or excluded from the IIS.

The default value of -1 lets the IIS algorithm decide.

If the attribute is set to 0, the constraint is not eligible for inclusion in the IIS.

If the attribute is set to 1, the constraint is included in the IIS and the IIS algorithm never considers the possibility of removing it.

Note that setting this attribute to 0 may make the resulting subsystem feasible (or consistent), which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

See the Model.computeIIS documentation for more details.

This attribute is ignored for LPs.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SARHSLow

Type: double
Modifiable: No

Right-hand side sensitivity information: smallest right-hand side value at which the current optimal basis would remain optimal. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

SARHSUp

Type: double
Modifiable: No

Right-hand side sensitivity information: largest right-hand side value at which the current optimal basis would remain optimal. Only available for basic solutions.

For examples of how to query or modify attributes, refer to our Attribute Examples.

FarkasDual

Type: double
Modifiable: No

Together, attributes FarkasDual and FarkasProof provide a certificate of infeasibility for the given infeasible problem. Specifically, FarkasDual provides a vector λ that can be used to form the following inequality from the original constraints that is trivially infeasible within the bounds of the variables:

$$\lambda^t A x \leq \lambda^t b.$$

This inequality is valid even if the original constraints have a mix of less-than and greater-than senses because $\lambda_i \geq 0$ if the i-th constraint has a $\leq$ sense and $\lambda_i \leq 0$ if the i-th constraint has a $\geq$ sense.

Let

$$\bar{a} := \lambda^t A$$

be the coefficients of this inequality and

$$\bar{b} := \lambda^t b$$

be its right hand side. With L_j and U_j being the lower and upper bounds of the variables x_j, we have $\bar{a}_j \geq 0$ if $U_j = \infty$, and $\bar{a}_j \leq 0$ if $L_j = -\infty$.

The minimum violation of the Farkas constraint is achieved by setting $x_j^* := L_j$ for $\bar{a}_j > 0$ and $x_j^* := U_j$ for $\bar{a}_j < 0$. Then, we can calculate the minimum violation as

$$\beta := \bar{a}^t x^* - \bar{b} = \sum_{j:\bar{a}_j>0} \bar{a}_j L_j + \sum_{j:\bar{a}_j<0} \bar{a}_j U_j - \bar{b}$$

where $\beta > 0$.

The FarkasProof attribute gives the value of β.

These attributes are only available when parameter InfUnbdInfo is set to 1.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.4 SOS Attributes

These are SOS attributes, meaning that they are associated with specific special-ordered set constraints in the model. You should use one of the various `get` routines to retrieve the value of an attribute. These are described at the beginning of this section. For the object-oriented interfaces, SOS attributes are retrieved by invoking the `get` method on an SOS object. For attributes that can be modified directly by the user, you can use one of the various `set` methods.

Attempting to query an attribute that is not available will produce an error. In C, the attribute query routine will return a `GRB_ERROR_DATA_NOT_AVAILABLE` error code. The object-oriented interfaces will throw an exception.

IISSOS

Type: int
Modifiable: No

For an infeasible model, indicates whether the SOS constraint participates in the computed Irreducible Inconsistent Subsystem (IIS). Only available after you have computed an IIS.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISSOSForce

Type: int
Modifiable: Yes

When computing an Irreducible Inconsistent Subsystem (IIS) for an infeasible model, indicates whether the SOS constraint should be included or excluded from the IIS.

The default value of -1 lets the IIS algorithm decide.

If the attribute is set to 0, the constraint is not eligible for inclusion in the IIS.

If the attribute is set to 1, the constraint is included in the IIS and the IIS algorithm never considers the possibility of removing it.

Note that setting this attribute to 0 may make the resulting subsystem feasible (or consistent), which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

See the Model.computeIIS documentation for more details.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.5 Quadratic Constraint Attributes

These are quadratic constraint attributes, meaning that they are associated with specific quadratic constraints in the model. You should use one of the various `get` routines to retrieve the value of an attribute. These are described at the beginning of this section. For the object-oriented interfaces, quadratic constraint attributes are retrieved by invoking the `get` method on a constraint object. For attributes that can be modified directly by the user, you can use one of the various `set` methods.

Attempting to query an attribute that is not available will produce an error. In C, the attribute query routine will return a `GRB_ERROR_DATA_NOT_AVAILABLE` error code. The object-oriented interfaces will throw an exception.

QCSense

Type: char
Modifiable: Yes

Quadratic constraint sense (`'<'`, `'>'`, or `'='`).

For examples of how to query or modify attributes, refer to our Attribute Examples.

QCRHS

Type: double
Modifiable: Yes

Quadratic constraint right-hand side. Note that a less-than-or-equal constraint with a RHS value of `1e20` or larger, or a greater-than-or-equal constraint with RHS value of `-1e20` or smaller, will be treated as always being satisfied. Equality constraints with very large RHS values can lead to numerical issues, so these should be avoided.

For examples of how to query or modify attributes, refer to our Attribute Examples.

QCName

Type: string
Modifiable: Yes

Quadratic constraint name.

For examples of how to query or modify attributes, refer to our Attribute Examples.

QCPi

Type: double
Modifiable: No

The constraint dual value in the current solution. Note that quadratic constraint dual values are only available when the QCPDual parameter is set to 1.

Only available for convex, continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

QCSlack

Type: double
Modifiable: No

The constraint slack in the current solution.

For examples of how to query or modify attributes, refer to our Attribute Examples.

QCTag

Type: string
Modifiable: Yes

Tag for quadratic constraints.

If you will be retrieving the solution to your model in JSON format, you might define a tag for every quadratic constraint that you plan to retrieve solution information for. Each quadratic constraint tag must be unique, and if any tag is used (variable tag, constraint tag, quadratic constraint tag) only tagged elements will appear in the JSON solution string. Tags must consist of printable US-ASCII characters. Using extended characters or escaped characters will result in an error. The maximum supported length for a tag is 10240 characters.

Note that quadratic constraint tags are only allowed for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISQConstr

Type: int
Modifiable: No

For an infeasible model, indicates whether the quadratic constraint participates in the computed Irreducible Inconsistent Subsystem (IIS). Only available after you have computed an IIS.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISQConstrForce

Type: int
Modifiable: Yes

When computing an Irreducible Inconsistent Subsystem (IIS) for an infeasible model, indicates whether the quadratic constraint should be included or excluded from the IIS.

The default value of -1 lets the IIS algorithm decide.

If the attribute is set to 0, the constraint is not eligible for inclusion in the IIS.

If the attribute is set to 1, the constraint is included in the IIS and the IIS algorithm never considers the possibility of removing it.

Note that setting this attribute to 0 may make the resulting subsystem feasible (or consistent), which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

See the Model.computeIIS documentation for more details.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.6 General Constraint Attributes

These are general constraint attributes, meaning that they are associated with specific general constraints in the model. If the attribute name contains string "GenConstr", then it is for all types of general constraints. If it starts with string "Func", then it is only for function constraints.

You should use one of the various `get` routines to retrieve the value of an attribute. These are described at the beginning of this section. For the object-oriented interfaces, general constraint attributes are retrieved by invoking the `get` method on a constraint object. For attributes that can be modified directly by the user, you can use one of the various `set` methods.

Attempting to query an attribute that is not available will produce an error. In C, the attribute query routine will return a `GRB_ERROR_DATA_NOT_AVAILABLE` error code. The object-oriented interfaces will throw an exception.

FuncPieceError

Type: double
Modifiable: Yes

If the FuncPieces attribute is set to value -1 or -2, this attribute provides the maximum allowed error (absolute for -1, relative for -2) in the piecewise-linear approximation.

For examples of how to query or modify attributes, refer to our Attribute Examples.

FuncPieceLength

Type: double
Modifiable: Yes

If the FuncPieces attribute is set to value 1, this attribute provides the length of each piece of the piecewise-linear approximation.

For examples of how to query or modify attributes, refer to our Attribute Examples.

FuncPieceRatio

Type: double
Modifiable: Yes

This attribute controls whether the piecewise-linear approximation of a function constraint is an underestimate of the function, an overestimate, or somewhere in between. A value of 0.0 will always underestimate, while a value of 1.0 will always overestimate. A value in between will interpolate between the underestimate and the overestimate. A special value of -1 chooses points that are on the original function.

See the discussion of function constraints for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

FuncPieces

Type: int
Modifiable: Yes

This attribute sets the strategy used for performing a piecewise-linear approximation of a function constraint. There are a few options:

- **FuncPieces = 0**: Ignore the attribute settings for this function constraint and use the parameter settings (FuncPieces, etc.) instead.
- **FuncPieces >= 2**: Sets the number of pieces; pieces are equal width.
- **FuncPieces = 1**: Uses a fixed width for each piece; the actual width is provided in the FuncPieceLength attribute.
- **FuncPieces = -1**: Bounds the absolute error of the approximation; the error bound is provided in the FuncPieceError attribute.
- **FuncPieces = -2**: Bounds the relative error of the approximation; the error bound is provided in the FuncPieceError attribute.

See the discussion of function constraints for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

FuncNonlinear

Type: int
Modifiable: Yes

This attribute controls whether the particular general function constraint is replaced with a static piecewise-linear approximation (0), or is handled inside the branch-and-bound tree using a dynamic outer-approximation approach (1). The default value (-1) means that the constraint handling will be controlled by the FuncNonlinear parameter.

See the discussion of function constraints for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

GenConstrType

Type: int
Modifiable: No

General constraint type.

The language APIs contain enums that allows you to map the integer constraint type to a more meaningful name. In C, you'll find `GRB_GENCONSTR_MAX`, `GRB_GENCONSTR_MIN`, etc. In the OO interfaces, you'll find `GRB.GENCONSTR_MAX`, `GRB.GENCONSTR_MIN`, etc.

For examples of how to query or modify attributes, refer to our Attribute Examples.

GenConstrName

Type: string
Modifiable: Yes

General constraint name.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISGenConstr

Type: int
Modifiable: No

For an infeasible model, indicates whether the general constraint participates in the computed Irreducible Inconsistent Subsystem (IIS). Only available after you have computed an IIS.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IISGenConstrForce

Type: int
Modifiable: Yes

When computing an Irreducible Inconsistent Subsystem (IIS) for an infeasible model, indicates whether the general constraint should be included or excluded from the IIS.

The default value of -1 lets the IIS algorithm decide.

If the attribute is set to 0, the constraint is not eligible for inclusion in the IIS.

If the attribute is set to 1, the constraint is included in the IIS and the IIS algorithm never considers the possibility of removing it.

Note that setting this attribute to 0 may make the resulting subsystem feasible (or consistent), which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

See the Model.computeIIS documentation for more details.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.7 Quality Attributes

These are solution quality attributes. They are associated with the overall model. You should use one of the various `get` routines to retrieve the value of an attribute. These are described at

the beginning of this section. For the object-oriented interfaces, quality attributes are retrieved by invoking the `get` method on a constraint object. For attributes that can be modified directly by the user, you can use one of the various `set` methods.

Attempting to query an attribute that is not available will produce an error. In C, the attribute query routine will return a `GRB_ERROR_DATA_NOT_AVAILABLE` error code. The object-oriented interfaces will throw an exception.

MaxVio

Type: double
Modifiable: No
Maximum of all (unscaled) violations that apply to model type.
For examples of how to query or modify attributes, refer to our Attribute Examples.

BoundVio

Type: double
Modifiable: No
Maximum (unscaled) bound violation.
Available for all model types.
For examples of how to query or modify attributes, refer to our Attribute Examples.

BoundSVio

Type: double
Modifiable: No
Maximum (scaled) bound violation.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

BoundVioIndex

Type: int
Modifiable: No
Index of variable with the largest (unscaled) bound violation.
Available for all model types.
For examples of how to query or modify attributes, refer to our Attribute Examples.

BoundSVioIndex

Type: int
Modifiable: No
Index of variable with the largest (scaled) bound violation.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

BoundVioSum

Type: double
Modifiable: No

Sum of (unscaled) bound violations.

Available for all model types.

For examples of how to query or modify attributes, refer to our Attribute Examples.

BoundSVioSum

Type: double
Modifiable: No

Sum of (scaled) bound violations.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrVio

Type: double
Modifiable: No

Reporting constraint violations for the simplex solver is actually more complex than it may appear, due to the treatment of slacks on linear inequality constraints. The simplex solver introduces explicit non-negative slack variables inside the algorithm. Thus, for example, $a^T x \leq b$ becomes $a^T x + s = b$. In this formulation, constraint errors can show up in two places: (i) as bound violations on the computed slack variable values, and (ii) as differences between $a^T x + s$ and b. We report the former as `ConstrVio` and the latter as `ConstrResidual`.

For MIP models, the maximum violation of the constraints, including linear, quadratic, SOS and general constraints, is reported in `ConstrVio`.

Available for all model types.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrSVio

Type: double
Modifiable: No

Maximum (scaled) slack bound violation.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrVioIndex

Type: int
Modifiable: No

Index of linear constraint with the largest (unscaled) slack bound violation for continuous linear models solved by simplex.

For MIP or other situations, it is for all the constraints. The constraint order is linear, quadratic, SOS and general. Assume there are l linear, q quadratic, s SOS and g general constraints and the

index i is between $l+q+s$ and $l+q+s+g$, then the general constraint with index $i-l-q-s$ has the biggest violation.

Available for all model types.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrSVioIndex

Type: int
Modifiable: No

Index of linear constraint with the largest (scaled) slack bound violation.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrVioSum

Type: double
Modifiable: No

Sum of (unscaled) slack bound violations.

Available for all model types.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrSVioSum

Type: double
Modifiable: No

Sum of (scaled) slack bound violations.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrResidual

Type: double
Modifiable: No

Reporting constraint violations for the simplex solver is actually more complex than it may appear, due to the treatment of slacks on linear inequality constraints. The simplex solver introduces explicit non-negative slack variables inside the algorithm. Thus, for example, $a^T x \leq b$ becomes $a^T x + s = b$. In this formulation, constraint errors can show up in two places: (i) as bound violations on the computed slack variable values, and (ii) as differences between $a^T x + s$ and b. We report the former as `ConstrVio` and the latter as `ConstrResidual`.

Only available for continuous models. For MIP models, constraint violations are reported in `ConstrVio`.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrSResidual

Type: double
Modifiable: No

Maximum (scaled) primal constraint error.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrResidualIndex

Type: int
Modifiable: No
Index of linear constraint with the largest (unscaled) constraint error.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrSResidualIndex

Type: int
Modifiable: No
Index of linear constraint with the largest (scaled) constraint error.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrResidualSum

Type: double
Modifiable: No
Sum of (unscaled) linear constraint violations.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

ConstrSResidualSum

Type: double
Modifiable: No
Sum of (scaled) linear constraint violations.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

DualVio

Type: double
Modifiable: No
Reporting dual constraint violations for the simplex solver is actually more complex than it may appear, due to the treatment of reduced costs for bounded variables. The simplex solver introduces explicit non-negative reduced-cost variables inside the algorithm. Thus, $a^T y \geq c$ becomes $a^T y - z = c$ (where y is the dual vector and z is the reduced cost). In this formulation, errors can show up in two places: (i) as bound violations on the computed reduced-cost variable values, and (ii) as differences between $a^T y - z$ and c. We report the former as `DualVio` and the latter as `DualResidual`.
`DualVio` reports the maximum (unscaled) reduced-cost bound violation.

Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

DualSVio

Type: double
Modifiable: No
Maximum (scaled) reduced cost violation.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

DualVioIndex

Type: int
Modifiable: No
Index of variable with the largest (unscaled) reduced cost violation. Note that the result may be larger than the number of variables in the model, which indicates that a constraint slack is the variable with the largest violation. Subtract the variable count from the result to get the index of the corresponding constraint.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

DualSVioIndex

Type: int
Modifiable: No
Index of variable with the largest (scaled) reduced cost violation. Note that the result may be larger than the number of variables in the model, which indicates that a constraint slack is the variable with the largest violation. Subtract the variable count from the result to get the index of the corresponding constraint.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

DualVioSum

Type: double
Modifiable: No
Sum of (unscaled) reduced cost violations.
Only available for continuous models.
For examples of how to query or modify attributes, refer to our Attribute Examples.

DualSVioSum

Type: double
Modifiable: No
Sum of (scaled) reduced cost violations.
Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

DualResidual

Type: double
Modifiable: No

Reporting dual constraint violations for the simplex solver is actually more complex than it may appear, due to the treatment of reduced costs for bounded variables. The simplex solver introduces explicit non-negative reduced-cost variables inside the algorithm. Thus, $a^T y \geq c$ becomes $a^T y - z = c$ (where y is the dual vector and z is the reduced cost). In this formulation, errors can show up in two places: (i) as bound violations on the computed reduced-cost variable values, and (ii) as differences between $a^T y - z$ and c. We report the former as `DualVio` and the latter as `DualResidual`.

`DualResidual` reports the maximum (unscaled) dual constraint error.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

DualSResidual

Type: double
Modifiable: No

Maximum (scaled) dual constraint error.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

DualResidualIndex

Type: int
Modifiable: No

Index of variable with the largest (unscaled) dual constraint error.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

DualSResidualIndex

Type: int
Modifiable: No

Index of variable with the largest (scaled) dual constraint error.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

DualResidualSum

Type: double
Modifiable: No

Sum of (unscaled) dual constraint errors.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

DualSResidualSum

Type: double
Modifiable: No

Sum of (scaled) dual constraint errors.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ComplVio

Type: double
Modifiable: No

Maximum complementarity violation. In an optimal solution, the product of the value of a variable and its reduced cost must be zero. This isn't always strictly true for interior point solutions. This attribute returns the maximum complementarity violation for any variable.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ComplVioIndex

Type: int
Modifiable: No

Index of variable with the largest complementarity violation.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ComplVioSum

Type: double
Modifiable: No

Sum of complementarity violation.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IntVio

Type: double
Modifiable: No

A MIP solver won't always assign strictly integral values to integer variables. This attribute returns the largest distance between the computed value of any integer variable and the nearest integer.

Only available for MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IntVioIndex

Type: int
Modifiable: No

Index of variable with the largest integrality violation.

Only available for MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

IntVioSum

Type: double
Modifiable: No

Sum of integrality violations.

Only available for MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.8 Multi-objective Attributes

These are the attributes for setting and querying multiple objectives (refer to this section for additional information on multi-objective optimization).

ObjN

Type: double
Modifiable: Yes

When the model has multiple objectives, this attribute is used to query or modify objective coefficients for objective n. You set n using the ObjNumber parameter. Note that when `ObjNumber` is equal to 0, `ObjN` is equivalent to Obj.

The number of objectives in the model can be queried (or modified) using the NumObj attribute.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjNCon

Type: double
Modifiable: Yes

When the model has multiple objectives, this attribute is used to query or modify the constant term for objective n. You set n using the ObjNumber parameter. Note that when `ObjNumber` is equal to 0, `ObjNCon` is equivalent to ObjCon.

The number of objectives in the model can be queried (or modified) using the NumObj attribute.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjNPriority

Type: int
Modifiable: Yes

This attribute is used to query or modify the priority of objective n when doing hierarchical multi-objective optimization. You set n using the ObjNumber parameter.

The default priority for an objective is 0.

The number of objectives in the model can be queried (or modified) using the NumObj attribute.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjNWeight

Type: double
Modifiable: Yes

This attribute is used to query or modify the weight of objective n when doing blended multi-objective optimization. You set n using the ObjNumber parameter.

The default weight for an objective is 1.0.

The number of objectives in the model can be queried (or modified) using the NumObj attribute.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjNRelTol

Type: double
Modifiable: Yes

This attribute is used to set the allowable degradation for objective n when doing hierarchical multi-objective optimization for MIP models. You set n using the ObjNumber parameter.

Hierarchical multi-objective MIP optimization will optimize for the different objectives in the model one at a time, in priority order. If it achieves objective value z when it optimizes for this objective, then subsequent steps are allowed to degrade this value by at most `ObjNRelTol`$^*|z|$.

Objective degradations are handled differently for multi-objective LP models. The allowable degradation is controlled strictly using the ObjNAbsTol.

The default relative tolerance for an objective is 0.

The number of objectives in the model can be queried (or modified) using the NumObj attribute.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjNAbsTol

Type: double
Modifiable: Yes

This attribute is used to set the allowable degradation for objective n when doing hierarchical multi-objective optimization. You set n using the ObjNumber parameter.

Hierarchical multi-objective MIP optimization will optimize for the different objectives in the model one at a time, in priority order. If it achieves objective value z when it optimizes for this objective, then subsequent steps are allowed to degrade this value by at most `ObjNAbsTol`.

Objective degradations are handled differently for multi-objective LP models. For LP models, solution quality for higher-priority objectives is maintained by fixing some variables to their values in previous optimal solutions. These fixings are decided using variable reduced costs. The value of the `ObjNAbsTol` parameter indicates the amount by which a fixed variable's reduced cost is allowed to violate dual feasibility. The value of the related ObjNRelTol attribute is ignored.

The default absolute tolerance for an objective is `1e-6`.

The number of objectives in the model can be queried (or modified) using the NumObj attribute.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjNVal

Type: double
Modifiable: No

This attribute is used to query the objective value obtained for objective n by the k-th solution stored in the pool of feasible solutions found so far for the problem. You set n using the ObjNumber parameter, while you set k using the SolutionNumber parameter.

The number of objectives in the model can be queried (or modified) using the NumObj attribute; while the number of stored solutions can be queried using the SolCount attribute.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ObjNName

Type: string
Modifiable: Yes

When the model has multiple objectives, this attribute is used to query or modify the name for objective n. You set n using the ObjNumber parameter.

The number of objectives in the model can be queried (or modified) using the NumObj attribute.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

NumObj

Type: int
Modifiable: Yes

Number of objectives in the model. If you modify this attribute, it will change the number of objectives in the model. Decreasing it will discard existing objectives. Increasing it will create new objectives (initialized to 0). Setting it to 0 will create a model with no objective (i.e., a feasibility

model). If you want to switch from a multi-objective model to a single-objective model you also need to set `NumObj` to 0 and call model update before installing a new single objective.

You can use the ObjNumber parameter, in conjunction with multi-objective attributes (ObjN, ObjNName, etc.), to query or modify attributes for different objectives. The value of `ObjNumber` should always be less than `NumObj`.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.9 Multi-Scenario Attributes

These are the attributes for setting and querying multiple scenarios (refer to this section for additional information on multi-scenario optimization).

ScenNLB

Type: double
Modifiable: Yes

When the model has multiple scenarios, this attribute is used to query or modify changes of the variable lower bounds in scenario n w.r.t. the base model. You set n using the ScenarioNumber parameter.

If an element of this array attribute is set to the `undefined` value (`GRB_UNDEFINED` in C and C++, or `GRB.UNDEFINED` in Java, .NET, and Python), it means that the corresponding value in the scenario is identical to the one in the base model.

The number of scenarios in the model can be queried (or modified) using the NumScenarios attribute.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ScenNUB

Type: double
Modifiable: Yes

When the model has multiple scenarios, this attribute is used to query or modify changes of the variable upper bounds in scenario n w.r.t. the base model. You set n using the ScenarioNumber parameter.

If an element of this array attribute is set to the `undefined` value (`GRB_UNDEFINED` in C and C++, or `GRB.UNDEFINED` in Java, .NET, and Python), it means that the corresponding value in the scenario is identical to the one in the base model.

The number of scenarios in the model can be queried (or modified) using the NumScenarios attribute.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ScenNObj

Type: double
Modifiable: Yes

When the model has multiple scenarios, this attribute is used to query or modify changes of the variable objective coefficients in scenario n w.r.t. the base model. You set n using the ScenarioNumber parameter.

If an element of this array attribute is set to the `undefined` value (`GRB_UNDEFINED` in C and C++, or `GRB.UNDEFINED` in Java, .NET, and Python), it means that the corresponding value in the scenario is identical to the one in the base model.

The number of scenarios in the model can be queried (or modified) using the NumScenarios attribute.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ScenNRHS

Type: double
Modifiable: Yes

When the model has multiple scenarios, this attribute is used to query or modify changes of the linear constraint right-hand sides in scenario n w.r.t. the base model. You set n using the ScenarioNumber parameter.

If an element of this array attribute is set to the `undefined` value (`GRB_UNDEFINED` in C and C++, or `GRB.UNDEFINED` in Java, .NET, and Python), it means that the corresponding value in the scenario is identical to the one in the base model.

The number of scenarios in the model can be queried (or modified) using the NumScenarios attribute.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ScenNName

Type: string
Modifiable: Yes

When the model has multiple scenarios, this attribute is used to query or modify the name for scenario n. You set n using the ScenarioNumber parameter.

The number of scenarios in the model can be queried (or modified) using the NumScenarios attribute.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ScenNObjBound

Type: double
Modifiable: No

When the model has multiple scenarios, this attribute is used to query the objective bound for scenario n. You set n using the ScenarioNumber parameter.

The number of scenarios in the model can be queried (or modified) using the NumScenarios attribute.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ScenNObjVal

Type: double
Modifiable: No

When the model has multiple scenarios, this attribute is used to query the objective value of the current solution for scenario n. You set n using the ScenarioNumber parameter. If no solution is available, this returns `GRB_INFINITY` (for a minimization objective).

The number of scenarios in the model can be queried (or modified) using the NumScenarios attribute.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

ScenNX

Type: double
Modifiable: No

When the model has multiple scenarios, this attribute is used to query the variable value in the current solution of scenario n. You set n using the ScenarioNumber parameter.

The number of scenarios in the model can be queried (or modified) using the NumScenarios attribute.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

NumScenarios

Type: int
Modifiable: Yes

Number of scenarios in the model. Modifying this attribute changes the number: decreasing it discards existing scenarios; increasing it creates new scenarios (initialized to have no changes w.r.t. the base model); setting it to 0 discards all scenarios so that the base model is no longer a multi-scenario model.

You can use the ScenarioNumber parameter, in conjunction with multi-scenario attributes (ScenNLB, ScenNUB, ScenNObj, ScenNRHS, ScenNName, etc.), to query or modify attributes for different scenarios. The value of `ScenarioNumber` should always be less than `NumScenarios`.

Please refer to the Multiple Scenarios discussion for more information.

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.10 Batch Attributes

BatchErrorCode

Type: int
Modifiable: No

Retrieve the last error code received from the Cluster Manager associated with this batch. A non-zero value indicates that a problem occurred. Refer to the Error Code table for a list of possible return values. Details on the error can be obtained by querying BatchErrorMessage.

Note that all Batch attributes are cached locally, and are only updated when you create a client-side batch object or when you explicitly update this cache (by calling the appropriate `update` function - GRBupdatebatch for **C**, update for **Python**, etc.).

For examples of how to query or modify attributes, refer to our Attribute Examples.

BatchErrorMessage

Type: string
Modifiable: No

Retrieve the last error message received from the Cluster Manager associated with this batch. The related error code can be queried through the BatchErrorCode attribute.

Note that all Batch attributes are cached locally, and are only updated when you create a client-side batch object or when you explicitly update this cache (by calling the appropriate `update` function - GRBupdatebatch for **C**, update for **Python**, etc.).

For examples of how to query or modify attributes, refer to our Attribute Examples.

BatchID

Type: string
Modifiable: No

The ID for the batch. Please refer to the Batch Optimization section for more details.

Note that all Batch attributes are cached locally, and are only updated when you create a client-side batch object or when you explicitly update this cache (by calling the appropriate `update` function - GRBupdatebatch for **C**, update for **Python**, etc.).

For examples of how to query or modify attributes, refer to our Attribute Examples.

BatchStatus

Type: int
Modifiable: No

Current status for the batch request. Status values are described in the Status Code section.

Note that all Batch attributes are cached locally, and are only updated when you create a client-side batch object or when you explicitly update this cache (by calling the appropriate `update` function - GRBupdatebatch for **C**, update for **Python**, etc.).

For examples of how to query or modify attributes, refer to our Attribute Examples.

12.11 Attribute Examples

Gurobi attribute handling is designed to be orthogonal, meaning that you only need to use a small number of routines to work with a large number of attributes. In particular:

- The names and meanings of the various Gurobi attributes remain constant across the different programming language APIs, although some decoration is required in each language.
- Given the type of an attribute (double, integer, etc.) and the programming language you wish to use it from, you simply need to identify the appropriate routine for that attribute type in that language in order to query or modify that attribute.

Consider the `LB` attribute, which captures the lower bound on a variable. You would refer to this attribute as follows in the different Gurobi APIs:

Language	Attribute
C	`GRB_DBL_ATTR_LB`
C++	`GRB_DoubleAttr_LB`
Java	`GRB.DoubleAttr.LB`
.NET	`GRB.DoubleAttr.LB`, or just `var.LB`
Python	`GRB.Attr.LB`, or just `var.lb`

To query the value of this attribute for an individual variable in the different APIs, you would do the following:

Language	Attribute Query Example
C	GRBgetdblattrelement`(model, GRB_DBL_ATTR_LB, var_index, &value);`
C++	var.get`(GRB_DoubleAttr_LB)`
Java	var.get`(GRB.DoubleAttr.LB)`
.NET	var.Get`(GRB.DoubleAttr.LB)`, or just `var.LB`
Python	var.getAttr`(GRB.Attr.LB)`, or just `var.lb`

Our APIs also include routines for querying attribute values for multiple variables or constraints at once, which is more efficient.

Attributes are referred to using a set of `enum` types in C++, Java, and .NET (one enum for double-valued attributes, one for int-valued attributes, etc.). In C and Python, the names listed above are simply constants that take string values. For example, `GRB_DBL_ATTR_LB` is defined in the C layer as:

```
#define GRB_DBL_ATTR_LB "LB"
```

In C and Python, you have the option of using the strings directly when calling attribute methods. If you wish to do so, note that character case and underscores are ignored. Thus, `MIN_COEFF` and `MinCoeff` are equivalent.

One important point to note about attributes modification is that it is done in a *lazy* fashion. Modifications don't actually affect the model until the next request to either update or optimize the model (GRBupdatemodel or GRBoptimize in C).

Refer to the following sections for more detailed examples of how to query or modify attributes from our various APIs:

- C

- C++
- C#
- Java
- Python
- Visual Basic

You can also also browse our Examples to get a better sense of how to use our attribute interface.

C Attribute Examples

Consider the case where you have a Gurobi model `m`. You can retrieve the number of variables in the model by querying the NumVars model attribute. This is an integer-valued, scalar attribute, so you use GRBgetintattr:

```
int cols;
error = GRBgetintattr(m, GRB_INT_ATTR_NUMVARS, &cols);
```

You can also use the name of the attribute directly:

```
int cols;
error = GRBgetintattr(m, "NumVars", &cols);
```

(Note that attribute capitalization doesn't matter in the C interface, so you could also use `"numVars"` or `"numvars"`).

If you've performed optimization on the model, the optimal objective value can be obtained by querying the ObjVal model attribute. This is a double-valued, scalar attribute, so you use GRBgetdblattr:

```
double objval;
error = GRBgetdblattr(m, GRB_DBL_ATTR_OBJVAL, &objval);
```

If you'd like to query the value that a variable takes in the computed solution, you can query the X variable attribute. This is a double-valued, vector attribute, so you have a few options for querying the associated values. You can retrieve the value for a single variable using GRBgetdblattrelement:

```
double x0;
error = GRBgetdblattrelement(m, GRB_DBL_ATTR_X, 0, &x0);
```

(we query the solution value for variable 0 in this example). You can also query attribute values for multiple variables using GRBgetdblattrarray or GRBgetdblattrlist:

```
double x[];
error = GRBgetdblattrarray(m, GRB_DBL_ATTR_X, 0, cols, x);
```

The former routine retrieves a contiguous set of values (`cols` values, starting from index `0` in our example). The latter allows you to provide a list of indices, and it returns the values for the corresponding entries.

For each attribute query routine, there's an analogous `set` routine. To set the upper bound of a variable, for example, you would use GRBsetdblattrelement:

```
error = GRBsetdblattrelement(m, GRB_DBL_ATTR_UB, 0, 0.0);
```

(In this example, we've set the upper bound for variable 0 to 0). You can set attribute values for multiple variables in a single call using GRBsetdblattrarray or GRBsetdblattrlist.

C++ Attribute Examples

Consider the case where you have a Gurobi model `m`. You can retrieve the number of variables in the model by querying the NumVars model attribute using the get method:

```
cols = m.get(GRB_IntAttr_NumVars);
```

If you've performed optimization on the model, the optimal objective value can be obtained by querying the ObjVal model attribute:

```
obj = m.get(GRB_DoubleAttr_ObjVal);
```

If you'd like to query the value that a variable takes in the computed solution, you can query the X attribute for the corresponding variable object:

```
vars = m.getVars()
for (int j = 0; j < cols; j++)
  xj = vars[j].get(GRB_DoubleAttr_X)
```

You can also query the value of `X` for multiple variables in a single get call on the model `m`:

```
double xvals[] = m.get(GRB_DoubleAttr_X, m.GetVars()))
```

For each attribute query method, there's an analogous `set` routine. To set the upper bound of a variable, for example:

```
v = m.getVars()[0]
v.set(GRB_DoubleAttr_UB, 0)
```

(In this example, we've set the upper bound for the first variable in the model to 0).

C# Attribute Examples

Consider the case where you have a Gurobi model `m`. You can retrieve the number of variables in the model by querying the NumVars model attribute (which is implemented as a .NET *property*):

```
cols = m.NumVars;
```

If you've performed optimization on the model, the optimal objective value can be obtained by querying the ObjVal model attribute:

```
obj = m.ObjVal;
```

If you'd like to query the value that a variable takes in the computed solution, you can query the X attribute for the corresponding variable object:

```
vars = m.GetVars()
for (int j = 0; j < cols; j++)
  xj = vars[j].X
```

You can also query the value of `X` for multiple variables in a single call using the Get method on the model `m`:

```
double[] xvals = m.Get(GRB.DoubleAttr.X, m.GetVars())
```

For each attribute query method, there's an analogous `Set` routine. To set the upper bound of a variable, for example:

```
v = m.GetVars()[0]
v.UB = 0
```

(In this example, we've set the upper bound for the first variable in the model to 0).

Java Attribute Examples

Consider the case where you have a Gurobi model `m`. You can retrieve the number of variables in the model by querying the NumVars model attribute using the get method:

```
cols = m.get(GRB.IntAttr.NumVars);
```

If you've performed optimization on the model, the optimal objective value can be obtained by querying the ObjVal model attribute:

```
obj = m.get(GRB.DoubleAttr.ObjVal);
```

If you'd like to query the value that a variable takes in the computed solution, you can query the X attribute for the corresponding variable object:

```
vars = m.getVars()
for (int j = 0; j < cols; j++)
  xj = vars[j].get(GRB.DoubleAttr.X)
```

You can also query the value of `X` for multiple variables in a single get call on the model `m`:

```
double[] xvals = m.get(GRB.DoubleAttr.X, m.getVars())
```

For each attribute query method, there's an analogous `set` routine. To set the upper bound of a variable, for example:

```
v = m.getVars()[0]
v.set(GRB.DoubleAttr.UB, 0)
```

(In this example, we've set the upper bound for the first variable in the model to 0).

Python Attribute Examples

Consider the case where you have a Gurobi model `m`. You can retrieve the number of variables in the model by querying the NumVars model attribute:

```
print(m.numVars)
```

(Note that attribute capitalization doesn't matter in the Python interface, so you could also query `m.NumVars` or `m.numvars`).

If you've performed optimization on the model, the optimal objective value can be obtained by querying the ObjVal model attribute:

```
print(m.objVal)
```

If you'd like to query the value that a variable takes in the computed solution, you can query the X attribute for the corresponding variable object:

```
for v in m.getVars():
  print(v.x)
```

You can also query the value of `X` for multiple variables in a single getAttr call on the model `m`:

```
print(m.getAttr(GRB.Attr.X, m.getVars()))
```

For each attribute query method, there's an analogous `set` routine. To set the upper bound of a variable, for example:

```
v = m.getVars()[0]
v.ub = 0
```

(In this example, we've set the upper bound for the first variable in the model to 0).

Visual Basic Attribute Examples

Consider the case where you have a Gurobi model `m`. You can retrieve the number of variables in the model by querying the NumVars model attribute (which is implemented as a .NET *property*):

```
cols = m.NumVars;
```

If you've performed optimization on the model, the optimal objective value can be obtained by querying the ObjVal model attribute:

```
obj = m.ObjVal;
```

If you'd like to query the value that a variable takes in the computed solution, you can query the X attribute for the corresponding variable object:

```
vars = m.GetVars()
For j As Integer = 0 To cols - 1
  xj = vars[j].X
```

You can also query the value of `X` for multiple variables in a single call using the Get method on the model `m`:

```
xvals = m.Get(GRB.DoubleAttr.X, m.GetVars()))
```

For each attribute query method, there's an analogous `Set` routine. To set the upper bound of a variable, for example:

```
v = m.GetVars()[0]
v.UB = 0
```

(In this example, we've set the upper bound for the first variable in the model to 0).

Parameters control the operation of the Gurobi solvers. They must be modified before the optimization begins. While you should feel free to experiment with different parameter settings, we recommend that you leave parameters at their default settings unless you find a compelling reason not to. For a discussion of when you might want to change parameter values, refer to our Parameter Guidelines.

The various Gurobi APIs all provide routines for querying and modifying parameter values. Refer to our Parameter Examples for additional information.

Some of the parameters below are used to configure a client program for use with a Compute Server, a Gurobi Instant Cloud instance, or a token server. Refer to our discussion of empty environments for details.

Available Gurobi Parameters

Termination: These parameters affect the termination of the algorithms. If the algorithm exceeds any of these limits, it will terminate and report a non-optimal termination status (see the Status Code section for further details). Note that the algorithm won't necessarily stop the moment it hits the specified limit. The termination check may occur well after the limit has been exceeded.

Parameter name	Purpose
BarIterLimit	Barrier iteration limit
BestBdStop	Best objective bound to stop
BestObjStop	Best objective value to stop
Cutoff	Objective cutoff
IterationLimit	Simplex iteration limit
MemLimit	Memory limit
NodeLimit	MIP node limit
SoftMemLimit	Soft memory limit
SolutionLimit	MIP feasible solution limit
TimeLimit	Time limit
WorkLimit	Work limit

Tolerances: These parameters control the allowable feasibility or optimality violations.

Parameter name	Purpose
BarConvTol	Barrier convergence tolerance
BarQCPConvTol	Barrier QCP convergence tolerance
FeasibilityTol	Primal feasibility tolerance
IntFeasTol	Integer feasibility tolerance
MarkowitzTol	Threshold pivoting tolerance

MIPGap	Relative MIP optimality gap
MIPGapAbs	Absolute MIP optimality gap
OptimalityTol	Dual feasibility tolerance
PSDTol	Positive semi-definite tolerance

Simplex: These parameters control the operation of the simplex algorithms.

Parameter name	Purpose
InfUnbdInfo	Generate additional info for infeasible/unbounded models
LPWarmStart	Warm start usage in simplex
NetworkAlg	Network simplex algorithm
NormAdjust	Simplex pricing norm
PerturbValue	Simplex perturbation magnitude
Quad	Quad precision computation in simplex
Sifting	Sifting within dual simplex
SiftMethod	LP method used to solve sifting sub-problems
SimplexPricing	Simplex variable pricing strategy

Barrier: These parameters control the operation of the barrier solver.

Parameter name	Purpose
BarCorrectors	Central correction limit
BarHomogeneous	Barrier homogeneous algorithm
BarOrder	Barrier ordering algorithm
Crossover	Barrier crossover strategy
CrossoverBasis	Crossover initial basis construction strategy
QCPDual	Compute dual variables for QCP models

Scaling: These parameters control the operation of the simplex algorithms and the barrier solver.

Parameter name	Purpose
ObjScale	Objective scaling
ScaleFlag	Model scaling

MIP: These parameters control the operation of the MIP algorithms.

Parameter name	Purpose
BranchDir	Branch direction preference
ConcurrentJobs	Enables distributed concurrent solver

ConcurrentMethod	Chooses continuous solvers to run concurrently
ConcurrentMIP	Enables concurrent MIP solver
ConcurrentSettings	Create concurrent environments from a list of .prm files
DegenMoves	Degenerate simplex moves
DistributedMIPJobs	Enables the distributed MIP solver
Heuristics	Turn MIP heuristics up or down
ImproveStartGap	Trigger solution improvement
ImproveStartNodes	Trigger solution improvement
ImproveStartTime	Trigger solution improvement
LazyConstraints	Programs that add lazy constraints must set this parameter
MinRelNodes	Minimum relaxation heuristic control
MIPFocus	Set the focus of the MIP solver
MIQCPMethod	Method used to solve MIQCP models
NLPHeur	Controls the NLP heuristic for non-convex quadratic models
NodefileDir	Directory for MIP node files
NodefileStart	Memory threshold for writing MIP tree nodes to disk
NodeMethod	Method used to solve MIP node relaxations
NonConvex	Control how to deal with non-convex quadratic programs
NoRelHeurTime	Limits the amount of time (in seconds) spent in the NoRel heuristic
NoRelHeurWork	Limits the amount of work performed by the NoRel heuristic
OBBT	Controls aggressiveness of optimality-based bound tightening
PartitionPlace	Controls when the partition heuristic runs
PumpPasses	Feasibility pump heuristic control
RINS	RINS heuristic
SolFiles	Location to store intermediate solution files
SolutionNumber	Sub-optimal MIP solution retrieval
StartNodeLimit	Node limit for MIP start sub-MIP
StartNumber	Set index of MIP start
SubMIPNodes	Nodes explored by sub-MIP heuristics
Symmetry	Symmetry detection
VarBranch	Branch variable selection strategy
ZeroObjNodes	Zero objective heuristic control

Presolve: These parameters control the operation of the presolve algorithms.

Parameter name	Purpose
AggFill	Allowed fill during presolve aggregation
Aggregate	Presolve aggregation control
DualReductions	Disables dual reductions in presolve
PreCrush	Allows presolve to translate constraints on the original model to equivalent constraints on the presolved model
PreDepRow	Presolve dependent row reduction
PreDual	Presolve dualization
PreMIQCPForm	Format of presolved MIQCP model
PrePasses	Presolve pass limit
PreQLinearize	Presolve Q matrix linearization
Presolve	Presolve level
PreSOS1BigM	Controls largest coefficient in SOS1 reformulation
PreSOS1Encoding	Controls SOS1 reformulation
PreSOS2BigM	Controls largest coefficient in SOS2 reformulation
PreSOS2Encoding	Controls SOS2 reformulation
PreSparsify	Presolve sparsify reduction

Tuning: These parameters control the operation of the parameter tuning tool.

Parameter name	Purpose
TuneBaseSettings	Comma-separated list of base parameter settings
TuneCleanup	Enables a tuning cleanup phase
TuneCriterion	Specify tuning criterion
TuneDynamicJobs	Enables distributed tuning using a dynamic set of workers
TuneJobs	Enables distributed tuning using a static set of workers
TuneMetric	Metric to aggregate results into a single measure
TuneOutput	Tuning output level
TuneResults	Number of improved parameter sets returned
TuneTargetMIPGap	A target gap to be reached
TuneTargetTime	A target runtime in seconds to be reached
TuneTimeLimit	Time limit for tuning
TuneTrials	Perform multiple runs on each parameter set to limit the effect of random noise
TuneUseFilename	Use model file names as model names

Multiple Solutions: These parameters allow you to modify the behavior of the MIP search in

order to find more than one solution to a MIP model.

Parameter name	Purpose
PoolGap	Relative gap for solutions in pool
PoolGapAbs	Absolute gap for solutions in pool
PoolSearchMode	Choose the approach used to find additional solutions
PoolSolutions	Number of solutions to keep in pool

MIP Cuts: These parameters affect the generation of MIP cutting planes. In all cases, a value of -1 corresponds to an automatic setting, which allows the solver to determine the appropriate level of aggressiveness in the cut generation. Unless otherwise noted, settings of 0, 1, and 2 correspond to no cut generation, conservative cut generation, or aggressive cut generation, respectively. The `Cuts` parameter provides global cut control, affecting the generation of all cuts. This parameter also has a setting of 3, which corresponds to very aggressive cut generation. The other parameters override the global `Cuts` parameter (so setting `Cuts` to 2 and `CliqueCuts` to 0 would generate all cut types aggressively, except clique cuts which would not be generated at all).

Parameter name	Purpose
BQPCuts	BQP cut generation
Cuts	Global cut generation control
CliqueCuts	Clique cut generation
CoverCuts	Cover cut generation
CutAggPasses	Constraint aggregation passes performed during cut generation
CutPasses	Root cutting plane pass limit
FlowCoverCuts	Flow cover cut generation
FlowPathCuts	Flow path cut generation
GomoryPasses	Root Gomory cut pass limit
GUBCoverCuts	GUB cover cut generation
ImpliedCuts	Implied bound cut generation
InfProofCuts	Infeasibility proof cut generation
LiftProjectCuts	Lift-and-project cut generation
MIPSepCuts	MIP separation cut generation
MIRCuts	MIR cut generation
MixingCuts	Mixing cut generation
ModKCuts	Mod-k cut generation
NetworkCuts	Network cut generation
ProjImpliedCuts	Projected implied bound cut generation
PSDCuts	PSD cut generation
RelaxLiftCuts	Relax-and-lift cut generation
RLTCuts	RLT cut generation
StrongCGCuts	Strong-CG cut generation
SubMIPCuts	Sub-MIP cut generation
ZeroHalfCuts	Zero-half cut generation

Distributed algorithms: Parameters that are used to control our distributed parallel algorithms

(distributed MIP, distributed concurrent, and distributed tuning).

Parameter name	Purpose
WorkerPassword	Password for distributed worker cluster
WorkerPool	Distributed worker cluster

Cloud: Parameters that are used to launch Gurobi Instant Cloud instances.

Parameter name	Purpose
CloudAccessID	Access ID for Gurobi Instant Cloud
CloudHost	Host for the Gurobi Cloud entry point
CloudSecretKey	Secret Key for Gurobi Instant Cloud
CloudPool	Cloud pool to use for Gurobi Instant Cloud instance

Compute Server: Parameters that are used to configure and launch Gurobi Compute Server jobs. You will normally set these in your license file, but you have the option of setting them through these parameters instead (by first constructing an empty environment). Refer to the Gurobi Remote Services Reference Manual for more information.

Parameter name	Purpose
ComputeServer	Name of a node in the Remote Services cluster.
ServerPassword	Client password for Remote Services cluster (or token server).
ServerTimeout	Network timeout interval
CSPriority	Job priority for Remote Services job
CSQueueTimeout	Queue timeout for new jobs
CSRouter	Router node for Remote Services cluster
CSGroup	Group placement request for cluster
CSTLSInsecure	Use insecure mode in Transport Layer Security (TLS)
CSIdleTimeout	Idle time before Compute Server kills a job
JobID	Job ID of current job

Cluster Manager: Parameters that are used to configure and launch Gurobi Cluster Manager. You will normally set these in your license file, but you have the option of setting them through these parameters instead (by first constructing an empty environment). Refer to the Gurobi Remote Services Reference Manual for more information.

Parameter name	Purpose
CSAPIAccessID	Access ID for Gurobi Cluster Manager
CSAPISecret	Secret key for Gurobi Cluster Manager
CSAppName	Application name of the batches or jobs
CSAuthToken	Token used internally for authentication
CSBatchMode	Controls Batch-Mode optimization
CSClientLog	Turns logging on or off
CSManager	URL for the Cluster Manager
UserName	User name to use when connecting to the Cluster Manager

Token server: Parameters that are used to launch jobs that check out tokens from a token server.

You will normally set these in your license file, but you have the option of setting them through these parameters instead (by first constructing an empty environment).

Parameter name	Purpose
ServerPassword	Client password for token server (or Remote Services cluster).
TokenServer	Name of your token server.
TSPort	Token server port number.

Web License Service: Parameters that are used to launch jobs that use the Web License Service (WLS). You will normally set these in your license file, but you have the option of setting them through these parameters instead (by first constructing an empty environment).

Parameter name	Purpose
LicenseID	License ID.
WLSAccessID	WLS access ID.
WLSSecret	WLS secret.
WLSToken	WLS token.
WLSTokenDuration	WLS token duration.
WLSTokenRefresh	Relative WLS token refresh interval.

Other: Other parameters.

Parameter name	Purpose
Disconnected	Disconnected component strategy
DisplayInterval	Frequency at which log lines are printed
FeasRelaxBigM	Big-M value for feasibility relaxations
FuncPieceError	Error allowed for PWL translation of function constraint
FuncPieceLength	Piece length for PWL translation of function constraint
FuncPieceRatio	Controls whether to under- or over-estimate function values in PWL approximation
FuncPieces	Sets strategy for PWL function approximation
FuncMaxVal	Maximum value for x and y variables in function constraints
FuncNonlinear	Controls whether general function constraints are treated as nonlinear functions or via PWL approximation
IgnoreNames	Indicates whether to ignore names provided by users
IISMethod	IIS method
InputFile	File to be read before optimization commences
IntegralityFocus	Set the integrality focus
JSONSolDetail	Controls the level of detail stored in generated JSON solution
LogFile	Log file name
LogToConsole	Console logging
Method	Algorithm used to solve continuous models
MultiObjMethod	Warm-start method to solve for subsequent objectives

MultiObjPre	Initial presolve on multi-objective models
MultiObjSettings	Create multi-objective settings from a list of .prm files
NumericFocus	Set the numerical focus
ObjNumber	Set index of multi-objectives
OutputFlag	Solver output control
Record	Enable API call recording
ResultFile	Result file written upon completion of optimization
ScenarioNumber	Set index of scenario in multi-scenario models
Seed	Modify the random number seed
SolutionTarget	Specify the solution target for LP
Threads	Number of parallel threads to use
UpdateMode	Change the behavior of lazy updates

13.1 Parameter Guidelines

This section provides a brief discussion of the roles of the various Gurobi parameters when solving continuous or MIP models, with some indication of their relative importance.

Note that you also have the option of using the Parameter Tuning Tool to tune parameters. We recommend that you browse this section, though, even if you use the tuning tool, to get a better understanding of the roles of the various parameters.

Continuous Models

If you wish to use Gurobi parameters to tune performance on continuous models, we offer the following guidelines.

Choosing the method for LP or QP

The most important parameter when solving an LP or QP is `Method`. The default setting (-1) uses the concurrent optimizer for an LP, and the parallel barrier solver for a QP. While the default is usually a good choice, you may want to choose a different method in a few situations.

If memory is tight, you should consider using the dual simplex method (`Method=1`) instead of the default. The default will invoke the barrier method, which can take a lot more memory than dual. In addition, the default for LP will try multiple algorithms simultaneously, and each requires a copy of the original model. By selecting dual simplex, you will only use one copy of the model.

Another scenario where you should change the default is when you must get the exact same optimal basis each time. For LP models, the default concurrent solver invokes multiple algorithms simultaneously on multi-core systems, returning the optimal basis from the one that finishes first. In rare cases, one algorithm may complete first in one run, while another completes first in another. This can potentially lead to different alternate optimal solutions. Selecting any other method, including the deterministic concurrent solver, will avoid this possibility. Note, however, that the deterministic concurrent solver can be significantly slower than the default concurrent solver.

Finally, if you are confronted with a difficult LP model, you should experiment with the different method options. While the default is rarely significantly slower than the best choice, you may find that one option is consistently faster or more robust for your models. There are no simple rules for predicting which method will work best for a particular family of models.

If you are solving QCP or SOCP models, note that the barrier algorithm is your only option.

Parallel solution

Among the remaining parameters that affect continuous models, the only one that you would typically want to adjust is `Threads`, which controls the number of threads used for the concurrent and parallel barrier algorithms. By default, concurrent and barrier will use all available cores in your machine (up to 32). Note that the simplex solvers can only use one thread, so this parameter has no effect on them.

If you would like to experiment with different strategies than the default ones when solving an LP model using the concurrent optimizer, we provide methods in C, C++, Java, .NET, and Python that allow you to create and configure concurrent environments.

Infeasible or unbounded models

If you are confronted with an infeasible or unbounded LP, additional details can be obtained when you set the `InfUnbdInfo` parameter. For an unbounded model, setting this parameter allows you to

retrieve an unbounded ray (using the `UnbdRay` attribute). For an infeasible model, setting this parameter allows you to retrieve a Farkas infeasibility proof (using the `FarkasDual` and `FarkasProof` attributes).

For the barrier algorithm, you should set the BarHomogeneous parameter to 1 whenever you have a model that you suspect is infeasible or unbounded. This algorithm is better at diagnosing infeasibility or unboundedness.

Special structure

If you wish to solve an LP model that has many more variables than constraints, you may want to try the sifting algorithm. Sifting is actually implemented within our dual simplex solver, so to select sifting, set the `Method` parameter to 1 (to select dual), and then set the `Sifting` parameter to a positive value. You can use the `SiftMethod` parameter to choose the algorithm that is used to solve the sub-problems that arise within the sifting algorithm. In general, sifting is only effective when the ratio between variables and constraints is extremely large (100 to 1 or more). Note that the default `Sifting` setting allows the Gurobi Optimizer to select sifting automatically when a problem has the appropriate structure, so you won't typically need to select it manually.

Additional parameters

The `ScaleFlag` parameter can be used to modify the scaling performed on the model. The default scaling value (-1) is usually the most effective choice, but turning off scaling entirely (0) can sometimes reduce constraint violations on the original model. Choosing a different scaling option (1, 2, or 3) can sometimes improve performance for particularly numerically difficult models. The `ObjScale` parameter allows you to scale just the objective. Objective scaling can be useful when the objective contains extremely large values, but it can also lead to large dual violations in the original, unscaled model, so it should be used sparingly.

The `SimplexPricing` parameter determines the method used to choose a simplex pivot. The default is usually the best choice. The `NormAdjust` parameter allows you to choose alternate simplex pricing norms. Again, the default is usually best. The `Quad` parameter allows you to force the simplex solver to use (or not use) quad precision. While quad precision can help for numerically difficult models, the default setting will typically recognize such cases automatically. The `PerturbValue` parameter allows you to adjust the magnitude of the simplex perturbation (used to overcome degeneracy). Again, the default value is typically effective.

Other Gurobi parameters control the details of the barrier solver. The `BarConvTol` and `BarQCPConvTol` parameters allow you to adjust barrier termination. While you can ask for more precision than the default, you will typically run into the limitations of double-precision arithmetic quite quickly. This parameter is typically used to indicate that you are willing to settle for a less accurate answer than the defaults would give. The `BarCorrectors` parameter allows you to adjust the number of central corrections applied in each barrier iteration. More corrections generally lead to more forward progress in each iteration, but at a cost of more expensive iterations. The `BarOrder` parameter allows you to choose the barrier ordering method. The default approach typically works well, but you can manually choose the less expensive Approximate Minimum Degree ordering option (`BarOrder=0`) if you find that ordering is taking too long.

MIP Models

While default settings generally work well, MIP models will often benefit from parameter tuning. We offer the following guidelines, but we also encourage you to experiment.

Most Important Parameters

The two most important Gurobi settings when solving a MIP model are probably the `Threads` and `MIPFocus` parameters. The `Threads` parameter controls the number of threads used by the parallel MIP solver. The default is to use all cores in the machine (up to 32). If you wish to leave some available for other activities, adjust this parameter accordingly.

The `MIPFocus` parameter allows you to modify your high-level solution strategy, depending on your goals. By default, the Gurobi MIP solver strikes a balance between finding new feasible solutions and proving that the current solution is optimal. If you are more interested in good quality feasible solutions, you can select `MIPFocus=1`. If you believe the solver is having no trouble finding the optimal solution, and wish to focus more attention on proving optimality, select `MIPFocus=2`. If the best objective bound is moving very slowly (or not at all), you may want to try `MIPFocus=3` to focus on the bound.

Solution Improvement

The `ImproveStartTime` and `ImproveStartGap` parameters can also be used to modify your high-level solution strategy, but in a different way. These parameters allow you to give up on proving optimality at a certain point in the search, and instead focus all attention on finding better feasible solutions from that point onward. The `ImproveStartTime` parameter allows you to make this transition after the specified time has elapsed, while the `ImproveStartGap` parameter makes the transition when the specified optimality gap has been achieved.

Termination

Another important set of Gurobi parameters affect solver termination. If the solver is unable to find a proven optimal solution within the desired time, you will need to indicate how to limit the search. The simplest option is to limit runtime using the `TimeLimit` parameter. Of course, using a wall-clock based time limit may lead to non-deterministic results. This means that performing the same optimization twice with exactly the same input data can lead to stopping at different points during the optimization process and thus producing different solver output. If a deterministic stopping criterion is desired, one may use the `WorkLimit` parameter instead. This specifies a limit on the total work that is spent on the optimization. One work unit corresponds very roughly to one second, but this greatly depends on the hardware on which Gurobi is running and on the model that has been solved.

Another common termination choice for MIP models is to set the `MIPGap` parameter. This parameter allows you to indicate that optimization should stop when the relative gap between the best known solution and the best known bound on the solution objective is less than the specified value. You can terminate when the absolute gap is below a desired threshold using the `MIPGapAbs` parameter. You can also terminate based strictly on the current lower or upper bound using the `BestBdStop` or `BestObjStop` parameters. Other termination options include `NodeLimit`, `IterationLimit`, `SolutionLimit`, and `Cutoff`. The first three indicate that optimization should terminate when the number of branch-and-bound nodes, the total number of simplex iterations, or the number of discovered feasible integer solutions exceeds the specified value, respectively. The `Cutoff` parameter indicates that the solver should only consider solutions whose objective values are better than the specified value, and should terminate if no such solutions are found.

Reducing Memory Usage

If you find that the Gurobi Optimizer exhausts memory when solving a MIP, you should modify the `NodefileStart` parameter. When the amount of memory used to store nodes (measured in GBytes) exceeds the specified parameter value, nodes are written to disk. We recommend a setting of `0.5`, but you may wish to choose a different value, depending on the memory available in your machine. By default, nodes are written to the current working directory. The `NodefileDir` parameter can be used to choose a different location.

If you still exhaust memory after setting the `NodefileStart` parameter to a small value, you should try limiting the thread count. Each thread in parallel MIP requires a copy of the model, as well as several other large data structures. Reducing the `Threads` parameter can sometimes significantly reduce memory usage.

Finally, to protect against exhausting the memory you can limit the memory that is available to Gurobi by setting the `MemLimit` or the `SoftMemLimit` parameters. If the total amount of memory that Gurobi tries to allocate exceeds this value (in GBytes), it will abort. In case of the `MemLimit` parameter, Gurobi will abort immediately with an `OUT_OF_MEMORY` error. In contrast, the `SoftMemLimit` may be overshot a little bit, but instead of returning an error it allows a graceful termination of the current optimization with a `MEMLIMIT` status code.

Speeding Up The Root Relaxation

The root relaxation in a MIP model can sometimes be quite expensive to solve. If you find that a lot of time is spent here, consider using the `Method` parameter to select a different continuous algorithm for the root. For example, `Method=2` would select the parallel barrier algorithm at the root, and `Method=3` would select the concurrent solver. Note that you can choose a different algorithm for the MIP node relaxations using the `NodeMethod` parameter, but it is rarely beneficial to change this from the default (dual simplex).

Difficult Relaxations

If you find that the solver is having trouble solving the root relaxation even after you have tried the recommendations above, or is spending an inordinate amount of time at the root node, you should try the `NoRel` heuristic (controlled by the `NoRelHeurTime` and `NoRelHeurWork` parameters). This heuristic attempts to find high-quality solutions without ever solving the MIP relaxation. It can often be quite effective, although of course it won't provide good lower bounds on the optimal objective.

Heuristics

A few Gurobi parameters control internal MIP strategies. The `Heuristics` parameter controls the fraction of runtime spent on feasibility heuristics. Increasing the parameter can lead to more and better feasible solutions, but it will also reduce the rate of progress in the best bound. The `SubMIPNodes` parameter controls the number of nodes explored in some of the more sophisticated local search heuristics inside the Gurobi solver. You can increase this if you are having trouble finding good feasible solutions. The `MinRelNodes`, `PumpPasses`, and `ZeroObjNodes` parameters control a set of expensive heuristics whose goal is to find a feasible solution. All are invoked at the end of the MIP root node and usually only if no feasible solution has been found already. Try these if you are having trouble finding any feasible solutions.

Cutting Planes

The Gurobi MIP solver employs a wide range of cutting plane strategies. The aggressiveness of these strategies can be controlled at a coarse level through the `Cuts` parameter, and at a finer grain through a further set of cuts parameters (e.g., `FlowCoverCuts`, `MIRCuts`, etc.). Each cut parameter can be set to Aggressive (2), Conservative (1), Automatic (-1), or None (0). The more specific parameters override the more general, so for example setting `MIRCuts` to None (0) while also setting `Cuts` to Aggressive (2) would aggressively generate all cut types, except MIR cuts which would not be generated at all. Very easy models can sometimes benefit from turning cuts off, while extremely difficult models can benefit from turning them to their Aggressive setting.

Presolve

Presolve behavior can be modified with a set of parameters. The `Presolve` parameter sets the aggressiveness level of presolve. Options are Aggressive (2), Conservative (1), Automatic (-1), or None (0). More aggressive application of presolve takes more time, but can sometimes lead to a significantly tighter model. The `PrePasses` provides finer-grain control of presolve. It limits the number of passes presolve performs. Setting it to a small value (e.g., 3) can reduce presolve runtime. The `Aggregate` parameter controls the aggregation level in presolve. Aggregation typically leads to a smaller formulation, but in rare cases it can introduce numerical issues. The `AggFill` parameter controls aggregation at a finer grain. It controls how much fill is tolerated in the constraint matrix from a single variable aggregation. The `PreSparsify` parameter enables an algorithm that can sometimes significantly reduce the number of non-zero values in the constraint matrix.

Coping with Integrality Violations

The MIP solver can sometimes exploit tolerances on integer variables to violate the intent of a constraint. The best-known example of this is probably *trickle flows*, where trivial integrality violations on fixed-charge (binary) variables can lead to solutions that allow significant flows down *closed* edges. The `IntegralityFocus` parameter allows you to tell the solver to take a much stricter approach to integrality (at a small performance penalty).

Additional Parameters

The `Symmetry` parameter controls symmetry detection. The default value usually works well. The `VarBranch` parameter controls the branching variable selection strategy within the branch-and-bound process. Variable selection can have a significant impact on overall time to solution, but the default strategy is usually the best choice.

Tolerances

The Gurobi solver includes a set of numerical tolerance parameters. These rarely require adjustment, and are included for advanced users who are having trouble with the numerical properties of their models. The `FeasibilityTol`, `IntFeasTol`, `MarkowitzTol`, and `OptimalityTol` parameters allow you to adjust the primal feasibility tolerance, the integer feasibility tolerance, the Markowitz tolerance for simplex basis factorization, and the dual feasibility tolerance, respectively.

13.2 Parameter Descriptions

AggFill

Presolve aggregation fill level

Type: int
Default value: -1
Minimum value: -1
Maximum value: MAXINT

Controls the amount of fill allowed during presolve aggregation. Larger values generally lead to presolved models with fewer rows and columns, but with more constraint matrix non-zeros.

The default value chooses automatically, and usually works well.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Aggregate

Presolve aggregation

Type: int
Default value: 1
Minimum value: 0
Maximum value: 2

Controls the aggregation level in presolve. The options are off (0), moderate (1), or aggressive (2). In rare instances, aggregation can lead to an accumulation of numerical errors. Turning it off can sometimes improve solution accuracy.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BarConvTol

Barrier convergence tolerance

Type: double
Default value: 1e-8
Minimum value: 0.0
Maximum value: 1.0

The barrier solver terminates when the relative difference between the primal and dual objective values is less than the specified tolerance (with a `GRB_OPTIMAL` status). Tightening this tolerance often produces a more accurate solution, which can sometimes reduce the time spent in crossover. Loosening it causes the barrier algorithm to terminate with a less accurate solution, which can be useful when barrier is making very slow progress in later iterations.

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BarCorrectors

Barrier central corrections

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	MAXINT

Limits the number of central corrections performed in each barrier iteration. The default value chooses automatically, depending on problem characteristics. The automatic strategy generally works well, although it is often possible to obtain higher performance on a specific model by selecting a value manually.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BarHomogeneous

Barrier homogeneous algorithm

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	1

Determines whether to use the homogeneous barrier algorithm. At the default setting (-1), it is only used when barrier solves a node relaxation for a MIP model. Setting the parameter to 0 turns it off, and setting it to 1 forces it on. The homogeneous algorithm is useful for recognizing infeasibility or unboundedness. It is a bit slower than the default algorithm.

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BarIterLimit

Barrier iteration limit

Type:	int
Default value:	1000
Minimum value:	0
Maximum value:	MAXINT

Limits the number of barrier iterations performed. This parameter is rarely used. If you would like barrier to terminate early, it is almost always better to use the BarConvTol parameter instead.

Optimization returns with an `ITERATION_LIMIT` status if the limit is exceeded (see the Status Code section for further details).

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BarOrder

Barrier ordering algorithm

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	1

Chooses the barrier sparse matrix fill-reducing algorithm. A value of 0 chooses Approximate Minimum Degree ordering, while a value of 1 chooses Nested Dissection ordering. The default value of -1 chooses automatically. You should only modify this parameter if you notice that the barrier ordering phase is consuming a significant fraction of the overall barrier runtime.

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BarQCPConvTol

Barrier convergence tolerance for QCP models

Type:	double
Default value:	1e-6
Minimum value:	0.0
Maximum value:	1.0

When solving a QCP model, the barrier solver terminates when the relative difference between the primal and dual objective values is less than the specified tolerance (with a `GRB_OPTIMAL` status). Tightening this tolerance may lead to a more accurate solution, but it may also lead to a failure to converge.

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BestBdStop

Objective bound to stop optimization

Type:	double
Default value:	Infinity
Minimum value:	-Infinity
Maximum value:	Infinity

Terminates as soon as the engine determines that the best bound on the objective value is at least as good as the specified value. Optimization returns with an `USER_OBJ_LIMIT` status in this case.

Note that you should always include a small tolerance in this value. Without this, a bound that satisfies the intended termination criterion may not actually lead to termination due to numerical round-off in the bound.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BestObjStop

Objective value to stop optimization

Type:	double
Default value:	-Infinity
Minimum value:	-Infinity
Maximum value:	Infinity

Terminate as soon as the engine finds a feasible solution whose objective value is at least as good as the specified value. Optimization returns with an `USER_OBJ_LIMIT` status in this case.

Note that you should always include a small tolerance in this value. Without this, a solution that satisfies the intended termination criterion may not actually lead to termination due to numerical round-off in the objective.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BQPCuts

BQP cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls Boolean Quadric Polytope (BQP) cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

BranchDir

Preferred branch direction

Type: int
Default value: 0
Minimum value: -1
Maximum value: 1

Determines which child node is explored first in the branch-and-cut search. The default value chooses automatically. A value of -1 will always explore the down branch first, while a value of 1 will always explore the up branch first.

Changing the value of this parameter rarely produces a significant benefit.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CliqueCuts

Clique cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls clique cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value choose automatically. Overrides the Cuts parameter.

We have observed that setting this parameter to its aggressive setting can produce a significant benefit for some large set partitioning models.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CloudAccessID

Access ID for Gurobi Instant Cloud

Type: string
Default value: ""

Set this parameter to the Access ID for your Instant Cloud license when launching a new instance. You can retrieve this string from your account on the Gurobi Instant Cloud Manager website.

You must set this parameter through either a `gurobi.lic` file (using `CLOUDACCESSID=id`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CloudHost

Host for the Gurobi Cloud entry point

Type: string
Default value: ""

Set this parameter to the host name of the Gurobi Cloud entry point. Currently `cloud.gurobi.com`.

You must set this parameter through either a `gurobi.lic` file (using `CLOUDHOST=host`) or an empty environment. Changing the parameter after your environment has been started will result in an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CloudSecretKey

Secret Key for Gurobi Instant Cloud

Type: string
Default value: ""

Set this parameter to the Secret Key for your Instant Cloud license when launching a new instance. You can retrieve this string from your account on the Gurobi Instant Cloud Manager website.

You must set this parameter through either a `gurobi.lic` file (using `CLOUDSECRETKEY=key`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CloudPool

Cloud pool to use for Gurobi Instant Cloud instance

Type: string
Default value: ""

Set this parameter to the name of the cloud pool you would like to use for your new Instant Cloud instance. You can browse your existing cloud pools or create new ones from your account on the Gurobi Instant Cloud Manager website.

You must set this parameter through either a `gurobi.lic` file (using `CLOUDPOOL=pool`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ComputeServer

Name of a node in the Remote Services cluster

Type: string
Default value: ""

Set this parameter to the name of a node in the Remote Services cluster where you'd like your Compute Server job to run. You can refer to the server using its name or its IP address. If you are using a non-default port, the server name should be followed by the port number (e.g., `server1:61000`).

You will also need to set the ServerPassword parameter to supply the client password for the specified cluster.

You can provide a comma-separated list of nodes to increase robustness. If the first node in the list doesn't respond, the second will be tried, etc.

Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs.

You must set this parameter through either a `gurobi.lic` file (using `COMPUTESERVER=server`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ConcurrentJobs

Distributed concurrent optimizer job count

Type: int
Default value: 0
Minimum value: 0
Maximum value: MAXINT

Enables distributed concurrent optimization, which can be used to solve LP or MIP models on multiple machines. A value of `n` causes the solver to create `n` independent models, using different parameter settings for each. Each of these models is sent to a distributed worker for processing. Optimization terminates when the first solve completes. Use the ComputeServer parameter to indicate the name of the cluster where you would like your distributed concurrent job to run (or use WorkerPool if your client machine will act as manager and you just need a pool of workers).

By default, Gurobi chooses the parameter settings used for each independent solve automatically. You can create concurrent environments to choose your own parameter settings (refer to the concurrent optimization section for details). The intent of concurrent MIP solving is to introduce additional diversity into the MIP search. By bringing the resources of multiple machines to bear on a single model, this approach can sometimes solve models much faster than a single machine.

The distributed concurrent solver produces a slightly different log from the standard solver, and provides different callbacks as well. Please refer to the `Distributed Algorithms` section of the Gurobi Remote Services Reference Manual for additional details.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ConcurrentMethod

Controls the methods used by the concurrent continuous solver

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	3

This parameter is only evaluated when solving an LP with a concurrent solver (Method = 3 or 4). It controls which methods are run concurrently by the concurrent solver. Options are:

- -1=automatic,
- 0=barrier, dual, primal simplex,
- 1=barrier and dual simplex,
- 2=barrier and primal simplex, and
- 3=dual and primal simplex.

Which methods are actually run also depends on the number of threads available.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ConcurrentMIP

Enables the concurrent MIP solver

Type:	int
Default value:	1
Minimum value:	1
Maximum value:	MAXINT

This parameter enables the concurrent MIP solver. When the parameter is set to value `n`, the MIP solver performs `n` independent MIP solves in parallel, with different parameter settings for each. Optimization terminates when the first solve completes.

By default, Gurobi chooses the parameter settings used for each independent solve automatically. You can create concurrent environments to choose your own parameter settings (refer to the concurrent optimization section for details). The intent of concurrent MIP solving is to introduce additional diversity into the MIP search. This approach can sometimes solve models much faster than applying all available threads to a single MIP solve, especially on very large parallel machines.

The concurrent MIP solver divides available threads evenly among the independent solves. For example, if you have 6 threads available and you set `ConcurrentMIP` to 2, the concurrent MIP solver will allocate 3 threads to each independent solve. Note that the number of independent solves launched will not exceed the number of available threads.

The concurrent MIP solver produces a slightly different log from the standard MIP solver, and provides different callbacks as well. Please refer to the concurrent optimizer discussion for additional details.

Concurrent MIP is not deterministic. If runtimes for different independent solves are very similar, and if the model has multiple optimal solutions, you may get slightly different results from multiple runs on the same model.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ConcurrentSettings

Create concurrent environments from a list of .prm files

Type: string
Default value: ""

This command-line only parameter allows you to specify a comma-separated list of .prm files that are used to set parameters for the different instances in a concurrent MIP run.

To give an example, you could create two `.prm` files with the following contents...

s0.prm:

```
MIPFocus 0
```

s1.prm:

```
MIPFocus 1
```

Issuing the command `gurobi_cl ConcurrentSettings=s0.prm,s1.prm model.mps` would invoke the concurrent MIP solver, using parameter setting `MIPFocus=0` in one of the two concurrent solves and `MIPFocus=1` in the other.

Note that if you want to run concurrent MIP on multiple machines, you must also set the ConcurrentJobs parameter. The command for running distributed concurrent optimization using the two example parameter files on two machines would be

```
> gurobi_cl ConcurrentJobs=2 ConcurrentSettings=s0.prm,s1.prm model.mps
```

Note: Command-line only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CoverCuts

Cover cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls cover cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Crossover

Barrier crossover strategy

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	4

Determines the crossover strategy used to transform the interior solution produced by barrier into a basic solution (note that crossover is not available for QP or QCP models). Crossover consists of three phases: (i) a *primal push* phase, where primal variables are pushed to bounds, (ii) a *dual push* phase, where dual variables are pushed to bounds, and (iii) a *cleanup* phase, where simplex is used to remove any primal or dual infeasibilities that remain after the push phases are complete. The order of the first two phases and the algorithm used for the third phase are both controlled by the `Crossover` parameter:

Parameter value	First push	Second push	Cleanup
0	Disabled	Disabled	Disabled
1	Dual	Primal	Primal
2	Dual	Primal	Dual
3	Primal	Dual	Primal
4	Primal	Dual	Dual

The default value of -1 chooses the strategy automatically. Use value 0 to disable crossover; this setting returns the interior solution computed by barrier.

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CrossoverBasis

Crossover basis construction strategy

Type: int
Default value: -1
Minimum value: -1
Maximum value: 1

Determines the initial basis construction strategy for crossover. A value of 0 chooses an initial basis quickly. A value of 1 can take much longer, but often produces a more numerically stable start basis. The default value of -1 makes an automatic choice.

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSAPIAccessID

Access ID for Gurobi Cluster Manager

Type: string
Default value: ""

A unique identifier used to authenticate an application on a Gurobi Cluster Manager.

You can provide either an access ID and a secret key, or a username and password, to authenticate your connection to a Cluster Manager.

You must set this parameter through either a `gurobi.lic` file (using `CSAPIACCESSID=YOUR_API_ID`) or an empty environment. Changing the parameter after your environment has been started will result in an error.

Note: Cluster Manager only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSAPISecret

Secret key for Gurobi Cluster Manager

Type: string
Default value: ""

The secret password associated with an API access ID.

You can provide either an access ID and a secret key, or a username and password, to authenticate your connection to a Cluster Manager.

You must set this parameter through either a `gurobi.lic` file (using `CSAPISECRET=YOUR_API_SECRET_KEY`) or an empty environment. Changing the parameter after your environment has been started will result in an error.

Note: Cluster Manager only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSAppName

Application name of the batches or jobs

Type: string
Default value: ""

The application name which will be sent to the server to track which application is submitting the batches or jobs.

Note: Cluster Manager only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSAuthToken

JSON Web Token for accessing the Cluster Manager

Type: string
Default value: ""

When a client authenticates with a Cluster Manager using a username and password, a signed token is returned by the server to be used in further calls or command-line operations. It is used internally.

Note: Cluster Manager only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSBatchMode

Controls Batch-Mode optimization

Type: int
Default value: 0
Minimum value: 0
Maximum value: 1

When set to 1, enable the local creation of models, and later submit batch-optimization jobs to the Cluster Manager. See the Batch Optimization section for more details. Note that if `CSBatchMode` is enabled, only batch-optimization calls are allowed.

You must set this parameter through either a `gurobi.lic` file (using `CSBATCHMODE=1`) or an empty environment. Changing the parameter after your environment has been started will result in an error.

Note: Cluster Manager only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSClientLog

Turns logging on or off

Type: int
Default value: 0
Minimum value: 0
Maximum value: 3

Turns logging on or off for Compute Server. Options are off (0), only error messages (1), information and error messages (2), or (3) verbose, information, and error messages.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSGroup

Group placement request for cluster

Type: string
Default value: ""

Specifies one or more groups of cluster nodes to control the placement of the job. The list is a comma-separated string of group names, with optionally a priority for a group. For example, specifying `group1:10,group2:50` means that the job will run on machines of `group1` or `group2`, and if the job is queued, it will have priority 10 on group1 and 50 on group2. Note that if the group is not specified, the job may run on any node. If there are no nodes in the cluster having the specified groups, the job will be rejected.

Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs and in particular to Gurobi Remote Services Cluster Grouping for more information on grouping cluster nodes.

You must set this parameter through either a license file (using `GROUP=name`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSIdleTimeout

Idle time before Compute Server kills a job

Type: int
Default value: -1
Minimum value: -1
Maximum value: MAXINT

This parameter allows you to set a limit on how long a Compute Server job can sit idle before the server kills the job (in seconds). A job is considered idle if the server is not currently performing an optimization and the client has not issued any additional commands.

The default value will allow a job to sit idle indefinitely in all but one circumstance. Currently the only exception is the Gurobi Instant Cloud, where the default setting will automatically impose a 30 minute idle time limit (1800 seconds). If you are using an Instant Cloud pool, the actual value will be the maximum between this parameter value and the idle timeout defined by the pool.

You must set this parameter through either a `gurobi.lic` file (using `IDLETIMEOUT=n`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSManager

URL of the Cluster Manager for the Remote Services cluster

Type: string
Default value: ""

URL of the Cluster Manager for the Remote Services cluster.

You must set this parameter through either a `gurobi.lic` file (using `CSMANAGER=YOUR_MANAGER_URL`) or an empty environment. Changing the parameter after your environment has been started will result in an error.

Note: Cluster Manager only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSPriority

Job priority for Remote Services job

Type:	int
Default value:	0
Minimum value:	-100
Maximum value:	100

The priority of the Compute Server job. Priorities must be between -100 and 100, with a default value of 0 (by convention). Higher priority jobs are chosen from the server job queue before lower priority jobs. A job with priority 100 runs immediately, bypassing the job queue and ignoring the job limit on the server. You should exercise caution with priority 100 jobs, since they can severely overload a server, which can cause jobs to fail, and in extreme cases can cause the server to crash.

Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs.

You must set this parameter through either a `gurobi.lic` file (using `PRIORITY=n`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSQueueTimeout

Queue timeout for new jobs

Type:	double
Default value:	-1
Minimum value:	-1
Maximum value:	Infinity

This parameter allows you to set a limit (in seconds) on how long a new Compute Server job will wait in queue before it gives up (and reports a `JOB_REJECTED` error). Note that there might be a delay of up to 20 seconds for the actual signaling of the time out.

Any negative value will allow a job to sit in the Compute Server queue indefinitely.

You must set this parameter through a `gurobi.lic` file (using `QUEUETIMEOUT=n`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSRouter

Router node for Remote Services cluster

Type:	string
Default value:	""

The router node for a Remote Services cluster. A router can be used to improve the robustness of a Compute Server deployment. You can refer to the router using either its name or its IP address. A typical Remote Services deployment won't use a router, so you typically won't need to set this parameter.

Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs.

You must set this parameter through either a `gurobi.lic` file (using `ROUTER=name`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CSTLSInsecure

Use insecure mode in Transport Layer Security (TLS)

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	1

Indicates whether the Remote Services cluster is using insecure mode in the TLS (Transport Layer Security). Leave this at its default value of 0 unless your server administrator tells you otherwise.

Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs.

You must set this parameter through either a `gurobi.lic` file (using `CSTLSINSECURE`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CutAggPasses

Constraint aggregation passes in cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	MAXINT

A non-negative value indicates the maximum number of constraint aggregation passes performed during cut generation. Overrides the Cuts parameter.

Changing the value of this parameter rarely produces a significant benefit.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Cutoff

Objective cutoff

Type:	double
Default value:	Infinity for minimization, -Infinity for maximization
Minimum value:	-Infinity
Maximum value:	Infinity

Indicates that you aren't interested in solutions whose objective values are worse than the specified value. If the objective value for the optimal solution is equal to or better than the specified cutoff, the solver will return the optimal solution. Otherwise, it will terminate with a `CUTOFF` status (see the Status Code section for further details).

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

CutPasses

Cutting plane passes

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	MAXINT

A non-negative value indicates the maximum number of cutting plane passes performed during root cut generation. The default value chooses the number of cut passes automatically.

In addition to cutting plane separation, each cut pass also applies heuristics and node probing and also may launch parallel root helper threads. So even when the Cuts parameter is set to 0, the cut loop will apply probing, heuristics and parallel root helpers in a single cut loop iteration.

You should experiment with different values of this parameter if you notice the MIP solver spending significant time on root cut passes that have little impact on the objective bound.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Cuts

Global cut control

Type: int
Default value: -1
Minimum value: -1
Maximum value: 3

Global cut aggressiveness setting. Use value 0 to shut off cuts, 1 for moderate cut generation, 2 for aggressive cut generation, and 3 for very aggressive cut generation. The default -1 value chooses automatically. This parameter is overridden by the parameters that control individual cut types (e.g., CliqueCuts).

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

DegenMoves

Degenerate simplex moves

Type: int
Default value: -1
Minimum value: -1
Maximum value: MAXINT

Limits degenerate simplex moves. These moves are performed to improve the integrality of the current relaxation solution. By default, the algorithm chooses the number of degenerate move passes to perform automatically.

The default setting generally works well, but there can be cases where an excessive amount of time is spent after the initial root relaxation has been solved but before the cut generation process or the root heuristics have started. If you see multiple 'Total elapsed time' messages in the log immediately after the root relaxation log, you may want to try setting this parameter to 0.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Disconnected

Disconnected component strategy

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

A MIP or an LP model can sometimes be made up of multiple, completely independent sub-models. This parameter controls how aggressively we try to exploit this structure. A value of 0 ignores this structure entirely, while larger values try more aggressive approaches. The default value of -1 chooses automatically.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

DisplayInterval

Frequency of log lines

Type: int
Default value: 5
Minimum value: 1
Maximum value: MAXINT

Determines the frequency at which log lines are printed (in seconds).

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

DistributedMIPJobs

Distributed MIP job count

Type: int
Default value: 0
Minimum value: 0
Maximum value: MAXINT

Enables distributed MIP. A value of `n` causes the MIP solver to divide the work of solving a MIP model among `n` machines. Use the ComputeServer parameter to indicate the name of the cluster where you would like your distributed MIP job to run (or use WorkerPool if your client machine will act as manager and you just need a pool of workers).

The distributed MIP solver produces a slightly different log from the standard MIP solver, and provides different callbacks as well. Please refer to the `Distributed Algorithms` section of the Gurobi Remote Services Reference Manual for additional details.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

DualReductions

Controls dual reductions

Type:	int
Default value:	1
Minimum value:	0
Maximum value:	1

Determines whether dual reductions are performed during the optimization process. You should disable these reductions if you received an optimization status of `INF_OR_UNBD` and would like a more definitive conclusion.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FeasibilityTol

Primal feasibility tolerance

Type:	double
Default value:	1e-6
Minimum value:	1e-9
Maximum value:	1e-2

All constraints must be satisfied to a tolerance of `FeasibilityTol`. Tightening this tolerance can produce smaller constraint violations, but for numerically challenging models it can sometimes lead to much larger iteration counts.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FeasRelaxBigM

Big-M value for feasibility relaxations

Type:	double
Default value:	1e6
Minimum value:	0
Maximum value:	Infinity

When relaxing a constraint in a feasibility relaxation, it is sometimes necessary to introduce a big-M value. This parameter determines the default magnitude of that value.

For details about feasibility relaxations, refer to e.g. GRBfeasrelax in the C API.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FlowCoverCuts

Flow cover cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls flow cover cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FlowPathCuts

Flow path cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls flow path cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FuncPieceError

Error allowed for PWL translation of function constraints

Type:	double
Default value:	1e-3
Minimum value:	1e-6
Maximum value:	1e+6

If the FuncPieces parameter is set to value -1 or -2, this attribute provides the maximum allowed error (absolute for -1, relative for -2) in the piecewise-linear approximation.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FuncPieceLength

Piece length for PWL translation of function constraints

Type:	double
Default value:	1e-2
Minimum value:	1e-5
Maximum value:	1e+6

If the FuncPieces parameter is set to value 1, this parameter gives the length of each piece of the piecewise-linear approximation.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FuncPieceRatio

Control whether to under- or over-estimate function values in PWL approximation

Type:	double
Default value:	-1
Minimum value:	-1
Maximum value:	1

This parameter controls whether the piecewise-linear approximation of a function constraint is an underestimate of the function, an overestimate, or somewhere in between. A value of 0.0 will always underestimate, while a value of 1.0 will always overestimate. A value in between will interpolate between the underestimate and the overestimate. A special value of -1 chooses points that are on the original function.

See the discussion of function constraints for more information.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FuncPieces

Sets strategy for PWL function approximation

Type:	int
Default value:	0
Minimum value:	-2
Maximum value:	MAXINT

This parameter sets the strategy used for performing a piecewise-linear approximation of a function constraint. There are a few options:

- **FuncPieces >= 2**: Sets the number of pieces; pieces are equal width.
- **FuncPieces = 1**: Uses a fixed width for each piece; the actual width is provided in the FuncPieceLength parameter.

- **FuncPieces = 0**: Default value; chooses automatically. Currently it uses the relative error approach for the approximation, while for version 10.0 or earlier it mainly uses the number of function constraints to set the total number of pieces.
- **FuncPieces = -1**: Bounds the absolute error of the approximation; the error bound is provided in the FuncPieceError parameter.
- **FuncPieces = -2**: Bounds the relative error of the approximation; the error bound is provided in the FuncPieceError parameter.

This parameter only applies to function constraints whose FuncPieces attribute has been set to 0.

See the discussion of function constraints for more information.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FuncMaxVal

Maximum allowed value for x and y variables in function constraints with piecewise-linear approximation

Type:	double
Default value:	1e+6
Minimum value:	0
Maximum value:	Infinity

Very large values in piecewise-linear approximations can cause numerical issues. This parameter limits the bounds on the variables that participate in function constraints approximated by a piecewise-linear function. Specifically, any bound larger than `FuncMaxVal` (in absolute value) on the variables participating in such a function constraint will be truncated.

If the FuncNonlinear attribute of the constraint is set to 1, or if it is set to -1 and the global FuncNonlinear parameter is set to 1, the function constraint is not approximated by a piecewise-linear function and the `FuncMaxVal` parameter does not apply.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

FuncNonlinear

Controls whether to treat general function constraints as nonlinear or via piecewise-linear approximation

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	1

This parameter controls whether general function constraints with their FuncNonlinear attribute set to -1 are treated as nonlinear functions (1) or are approximated via the piecewise-linear approximation approach (0).

See the discussion of function constraints for more information.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

GomoryPasses

Gomory cut passes

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	MAXINT

A non-negative value indicates the maximum number of Gomory cut passes performed. Overrides the Cuts parameter.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

GUBCoverCuts

GUB cover cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls GUB cover cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Heuristics

Time spent in feasibility heuristics

Type: double
Default value: 0.05
Minimum value: 0
Maximum value: 1

Determines the amount of time spent in MIP heuristics. You can think of the value as the desired fraction of total MIP runtime devoted to heuristics (so by default, we aim to spend 5% of runtime on heuristics). Larger values produce more and better feasible solutions, at a cost of slower progress in the best bound.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

IgnoreNames

Indicates whether to ignore names provided by users.

Type: int
Default value: 0
Minimum value: 0
Maximum value: 1

This parameter affects how Gurobi deals with names. If set to 1, subsequent calls to add variables or constraints to the model will ignore the associated names. Names for objectives and the model will also be ignored. In addition, subsequent calls to modify name attributes will have no effect. Note that variables or constraints that had names at the point this parameter was changed to 1 will retain their names. If you wish to discard all name information, you should set this parameter to 1 before adding variables or constraints to the model.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

IISMethod

Selects method used to compute IIS

Type: int
Default value: -1
Minimum value: -1
Maximum value: 3

Chooses the IIS method to use. To compute an IIS for an LP, it is sufficient to solve an LP with dimensions similar to the dual of the original model. If the solve time for that LP is excessive, setting the IISMethod parameter to 1 may offer a faster alternative; other settings do not alter the default approach for infeasible LPs. For MIPs, filtering of constraints and variables is required, which involves solving a series of related MIP subproblems. Methods 0-2 all use filtering techniques. Method 0 is often faster than method 1, but may produce a larger IIS. Method 2 ignores the bound

constraints. It therefore tends to be faster than methods 0-1, but will fail if these bounds are necessary to make the problem infeasible. Method 3 will return the IIS for the LP relaxation of a MIP model if the relaxation is infeasible, even though the result may not be minimal when integrality constraints are included. The default value of -1 chooses automatically.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ImpliedCuts

Implied bound cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls implied bound cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ImproveStartGap

Solution improvement strategy control

Type: double
Default value: 0.0
Minimum value: 0.0
Maximum value: Infinity

The MIP solver can change parameter settings in the middle of the search in order to adopt a strategy that gives up on moving the best bound and instead devotes all of its effort towards finding better feasible solutions. This parameter allows you to specify an optimality gap at which the MIP solver switches to a solution improvement strategy. For example, setting this parameter to 0.1 will cause the MIP solver to switch strategies once the relative optimality gap is smaller than 0.1.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ImproveStartNodes

Solution improvement strategy control

Type: double
Default value: Infinity
Minimum value: 0.0
Maximum value: Infinity

The MIP solver can change parameter settings in the middle of the search in order to adopt a strategy that gives up on moving the best bound and instead devotes all of its effort towards finding better feasible solutions. This parameter allows you to specify the node count at which the MIP solver switches to a solution improvement strategy. For example, setting this parameter to 10 will cause the MIP solver to switch strategies once the node count is larger than 10.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ImproveStartTime

Solution improvement strategy control

Type: double
Default value: Infinity
Minimum value: 0.0
Maximum value: Infinity

The MIP solver can change parameter settings in the middle of the search in order to adopt a strategy that gives up on moving the best bound and instead devotes all of its effort towards finding better feasible solutions. This parameter allows you to specify the time when the MIP solver switches to a solution improvement strategy. For example, setting this parameter to 10 will cause the MIP solver to switch strategies 10 seconds after starting the optimization.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

InfProofCuts

Infeasibility proof cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls infeasibility proof cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

InfUnbdInfo

Additional info for infeasible/unbounded models

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	1

Determines whether simplex (and crossover) will compute additional information when a model is determined to be infeasible or unbounded. Set this parameter if you want to query the unbounded ray for unbounded models (through the UnbdRay attribute), or the infeasibility proof for infeasible models (through the FarkasDual and FarkasProof attributes).

Note that if a model is found to be either infeasible or unbounded, and you simply want to know which one it is, you should use the DualReductions parameter instead. It performs much less additional computation.

Note: Only affects linear programming (LP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

InputFile

Import data into a model before beginning optimization

Type:	string
Default value:	""

Specifies the name of a file that will be read before beginning a command-line optimization run. This parameter can be used to input a MIP start (a `.mst` or `.sol` file), MIP hints (a `.hnt` file), a simplex basis (a `.bas` file), Gurobi attributes (a `.attr` file), or a set of parameter settings (a `.prm` file) from the Gurobi command line. The suffix may optionally be followed by `.zip`, `.gz`, `.bz2`, or `.7z` if the input files are compressed.

Note: Command-line only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

IntegralityFocus

Integrality focus

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	1

One unfortunate reality in MIP is that integer variables don't always take exact integral values. While this typically doesn't create significant problems, in some situations the side-effects can be quite undesirable. The best-known example is probably a *trickle flow*, where a continuous variable that is meant to be zero when an associated binary variable is zero instead takes a non-trivial value. More precisely, given a constraint $y \leq Mb$, where y is a non-negative continuous variable, b is a binary variable, and M is a constant that captures the largest possible value of y, the constraint is intended to enforce the relationship that y must be zero if b is zero. With the default integer feasibility tolerance, the binary variable is allowed to take a value as large as $1e-5$ while still being considered as taking value zero. If the M value is large, then the Mb upper bound on the y variable can be substantial.

Reducing the value of the IntFeasTol parameter can mitigate the effects of such trickle flows, but often at a significant cost, and often with limited success. The `IntegralityFocus` parameter provides a better alternative. Setting this parameter to 1 requests that the solver work harder to try to avoid solutions that exploit integrality tolerances. More precisely, the solver tries to find solutions that are still (nearly) feasible if all integer variables are rounded to exact integral values. We should say that the solver won't always succeed in finding such solutions, and that this setting introduces a modest performance penalty, but the setting will significantly reduce the frequency and magnitude of such violations.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

IntFeasTol

Integer feasibility tolerance

Type:	double
Default value:	1e-5
Minimum value:	1e-9
Maximum value:	1e-1

An integrality restriction on a variable is considered satisfied when the variable's value is less than `IntFeasTol` from the nearest integer value. Tightening this tolerance can produce smaller integrality violations, but very tight tolerances may significantly increase runtime. Loosening this tolerance rarely reduces runtime.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

IterationLimit

Simplex iteration limit

Type: double
Default value: Infinity
Minimum value: 0
Maximum value: Infinity

Limits the number of simplex iterations performed. The limit applies to MIP, barrier crossover, and simplex. Optimization returns with an `ITERATION_LIMIT` status if the limit is exceeded (see the Status Code section for further details).

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

JobID

Compute Server Job ID

Type: string
Default value: ""

If you are running on a Compute Server, this parameter provides the Compute Server Job ID for the current job. Note that this is a read-only parameter.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

JSONSolDetail

Level of detail in JSON solution format

Type: int
Default value: 0
Minimum value: 0
Maximum value: 1

This parameter controls the amount of detail included in a JSON solution. For example, when this parameter is set to 1, the JSON string will contain data for all of the variables, even those with solution value 0.

For a precise description of the contents of the resulting JSON string, please refer to the JSON solution format section.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

LazyConstraints

Programs that use lazy constraints must set this parameter

Type: int
Default value: 0
Minimum value: 0
Maximum value: 1

Programs that add lazy constraints through a callback must set this parameter to value 1. The parameter tells the Gurobi algorithms to avoid certain reductions and transformations that are incompatible with lazy constraints.

Note that if you use lazy constraints by setting the Lazy attribute (and not through a callback), there's no need to set this parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

LicenseID

License ID

Type: int
Default value: 0
Minimum value: 0
Maximum value: MAXINT

When using a WLS license, set this parameter to the license ID. You can retrieve this value from your account on the Gurobi Web License Manager site.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

LiftProjectCuts

Lift-and-project cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls lift-and-project cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

LPWarmStart

Controls whether and how to warm-start LP optimization

Type:	int
Default value:	1
Minimum value:	0
Maximum value:	2

Controls whether and how Gurobi uses warm start information for an LP optimization. The non default setting of 2 is particularly useful for communicating advanced start information while retaining the performance benefits of presolve. A warm start can consist of any combination of basis statuses, a primal start vector, or a dual start vector. It is specified using the attributes VBasis and CBasis or PStart and DStart on the original model.

As a general rule, setting this parameter to 0 ignores any start information and solves the model from scratch. Setting it to 1 (the default) uses the provided warm start information to solve the original, unpresolved problem, regardless of whether presolve is enabled. Setting it to 2 uses the start information to solve the presolved problem, assuming that presolve is enabled. This involves mapping the solution of the original problem into an equivalent (or sometimes nearly equivalent) crushed solution of the presolved problem. If presolve is disabled, then setting 2 still prioritizes start vectors, while setting 1 prioritizes basis statuses. Taken together, the LPWarmStart parameter setting, the LP algorithm specified by Gurobi's Method parameter, and the available advanced start information determine whether Gurobi will use basis statuses only, basis statuses augmented with information from start vectors, or a basis obtained by applying the crossover method to the provided primal and dual start vectors to jump start the optimization.

When Gurobi's Method parameter requests the barrier solver, primal and dual start vectors are prioritized over basis statuses (but only if you provide both). These start vectors are fed to the crossover procedure. This is the same crossover that is used to compute a basic solution from the interior solution produced by the core barrier algorithm, but in this case crossover is started from arbitrary start vectors. If you set the LPWarmStart parameter to 1, crossover will be invoked on the original model using the provided vectors. Any provided basis information will not be used in this case. If you set LPWarmStart to 2, crossover will be invoked on the presolved model using crushed start vectors. If you set the parameter to 2 and provide a basis but no start vectors, the basis will be used to compute the corresponding primal and dual solutions on the original model. Those solutions will then be crushed and used as primal and dual start vectors for the crossover, which will then construct a basis for the presolved model. Note that for all of these settings and start combinations, no barrier algorithm iterations are performed.

The simplex algorithms provide more warm-starting options. With a parameter value of 1, simplex will start from a provided basis, if available. Otherwise, it uses a provided start vector to refine the crash basis it computes. Primal simplex will use `PStart` and dual simplex will use `DStart` in this refinement process.

With a value of 2, simplex will use the crushed start vector on the presolved model (`PStart` for primal simplex, `DStart` for dual) to refine the crash basis. This is true regardless of whether the start is derived from start vectors or a starting basis from the original model. The difference is that if you provide an advanced basis, the basis will be used to compute the corresponding primal and dual solutions on the original model from which the primal or dual start on the presolved model

will be derived.

Note: Only affects linear programming (LP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

LogFile

Name for Gurobi log file

Type: string
Default value: ""

Determines the name of the Gurobi log file. Modifying this parameter closes the current log file and opens the specified file. Use an empty string for no log file. Use OutputFlag to shut off all logging.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

LogToConsole

Control console logging

Type: int
Default value: 1
Minimum value: 0
Maximum value: 1

Enables or disables console logging. Use OutputFlag to shut off all logging.

Note that this refers to all the output of Gurobi to the console including the various display and print functions provided by the API in interactive environments. It also disables logging on the remote side of Instant Cloud or Compute Server environments.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MarkowitzTol

Threshold pivoting tolerance

Type: double
Default value: 0.0078125
Minimum value: 1e-4
Maximum value: 0.999

The Markowitz tolerance is used to limit numerical error in the simplex algorithm. Specifically, larger values reduce the error introduced in the simplex basis factorization. A larger value may avoid numerical problems in rare situations, but it will also harm performance.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MemLimit

Memory limit

Type:	double
Default value:	Infinity
Minimum value:	0
Maximum value:	Infinity

Limits the total amount of memory (in GB, i.e., 10^9 bytes) available to Gurobi. If more is needed, Gurobi will fail with an `OUT_OF_MEMORY` error (see the Error Code section for further details).

Note that it is not possible to retrieve solution information after an error termination. Thus, the behavior of this parameter is different from that of other termination criteria like SoftMemLimit, TimeLimit, or NodeLimit, where the solver will terminate with a Status Code and solution information will still be available.

One advantage of using this parameter rather than the similar SoftMemLimit is that `MemLimit` is checked after every memory allocation, so Gurobi will terminate at precisely the point where the limit is exceeded.

Note that allocated memory is tracked across all models within a Gurobi environment. If you create multiple models in one environment, these additional models will count towards overall memory consumption.

Memory usage is also tracked across all threads. One consequence of this is that termination may be non-deterministic for multi-threaded runs.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Method

Algorithm used to solve continuous models

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	5

Algorithm used to solve continuous models or the initial root relaxation of a MIP model. Options are:

- -1=automatic,
- 0=primal simplex,
- 1=dual simplex,
- 2=barrier,

- 3=concurrent,
- 4=deterministic concurrent, and
- 5=deterministic concurrent simplex (deprecated; see ConcurrentMethod).

Available settings and default behavior depend on the model type or the type of the initial root relaxation. In the current release, the default Automatic (-1) setting will typically choose non-deterministic concurrent (Method=3) for an LP, barrier (Method=2) for a QP or QCP, and dual (Method=1) for the MIP root relaxation. If the size of the MIP root relaxation is large, then it will often select deterministic concurrent (Method=4).

Concurrent methods aren't available for QP and QCP. Only the simplex and barrier algorithms are available for continuous QP models. If you select barrier (Method=2) to solve the root of an MIQP model, then you need to also select barrier for the node relaxations (i.e. set NodeMethod=2). Only barrier is available for continuous QCP models. However if you choose LP relaxations for solving MIQCP, you can also select the simplex algorithms (Method=0 or 1).

Concurrent optimizers run multiple solvers on multiple threads simultaneously and choose the one that finishes first. The solvers that are run concurrently can be controlled with the ConcurrentMethod parameter. The deterministic option (Method=4) give the exact same result each time, while the non-deterministic option (Method=3) is often faster but can produce different optimal bases when run multiple times.

The default setting is rarely significantly slower than the best possible setting, so you generally won't see a big gain from changing this parameter. There are classes of models where one particular algorithm is consistently fastest, though, so you may want to experiment with different options when confronted with a particularly difficult model.

Note that if memory is tight on an LP model, you should consider using the dual simplex method (Method=1). The concurrent optimizer, which is typically chosen when using the default setting, consumes a lot more memory than dual simplex alone.

In multiobjective LP optimization:

- The first objective is solved using LP defaults. It can be set by the user using the `Method` parameter.
- Subsequent objectives are solved by default using primal simplex to allow for warm starting. The method used here can be controlled using MultiObjMethod.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MinRelNodes

Minimum relaxation heuristic

Type: int
Default value: -1
Minimum value: -1
Maximum value: MAXINT

Number of nodes to explore in the minimum relaxation heuristic. Note that this heuristic is only applied at the end of the MIP root, and only when no other root heuristic finds a feasible solution.

This heuristic is quite expensive, and generally produces poor quality solutions. You should generally only use it if other means, including exploration of the tree with default settings, fail to produce a feasible solution.

The default value automatically chooses whether to apply the heuristic. It will only rarely choose to do so.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MIPFocus

MIP solver focus

Type: int
Default value: 0
Minimum value: 0
Maximum value: 3

The `MIPFocus` parameter allows you to modify your high-level solution strategy, depending on your goals. By default, the Gurobi MIP solver strikes a balance between finding new feasible solutions and proving that the current solution is optimal. If you are more interested in finding feasible solutions quickly, you can select `MIPFocus=1`. If you believe the solver is having no trouble finding good quality solutions, and wish to focus more attention on proving optimality, select `MIPFocus=2`. If the best objective bound is moving very slowly (or not at all), you may want to try `MIPFocus=3` to focus on the bound.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MIPGap

Relative MIP optimality gap

Type: double
Default value: 1e-4
Minimum value: 0
Maximum value: Infinity

The MIP solver will terminate (with an optimal result) when the gap between the lower and upper objective bound is less than `MIPGap` times the absolute value of the incumbent objective value. More precisely, if z_P is the primal objective bound (i.e., the incumbent objective value, which is the upper bound for minimization problems), and z_D is the dual objective bound (i.e., the lower bound for minimization problems), then the MIP gap is defined as

$$gap = |z_P - z_D|/|z_P|.$$

Note that if $z_P = z_D = 0$, then the gap is defined to be zero. If $z_P = 0$ and $z_D \neq 0$, the gap is defined to be infinity.

For most models, z_P and z_D will have the same sign throughout the optimization process, and then the gap is monotonically decreasing. But if z_P and z_D have opposite signs, the relative gap may increase after finding a new incumbent solution, even though the absolute gap $|z_P - z_D|$ has decreased.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MIPGapAbs

Absolute MIP optimality gap

Type:	double
Default value:	1e-10
Minimum value:	0
Maximum value:	Infinity

The MIP solver will terminate (with an optimal result) when the gap between the lower and upper objective bound is less than `MIPGapAbs`.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MIPSepCuts

MIP separation cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls MIP separation cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MIQCPMethod

Method used to solve MIQCP models

Type: int
Default value: -1
Minimum value: -1
Maximum value: 1

Controls the method used to solve MIQCP models. Value 1 uses a linearized, outer-approximation approach, while value 0 solves continuous QCP relaxations at each node. The default setting (-1) chooses automatically.

Note: Only affects MIQCP models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MIRCuts

MIR cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls Mixed Integer Rounding (MIR) cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MixingCuts

Mixing cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls Mixing cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ModKCuts

Mod-k cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls mod-k cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MultiObjMethod

Method used for multi-objective solves

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

When solving a continuous multi-objective model using a hierarchical approach, the model is solved once for each objective. The algorithm used to solve for the highest priority objective is controlled by the Method parameter. This parameter determines the algorithm used to solve for subsequent objectives. As with the `Method` parameters, values of 0 and 1 use primal and dual simplex, respectively. A value of 2 indicates that warm-start information from previous solves should be discarded, and the model should be solved from scratch (using the algorithm indicated by the `Method` parameter). The default setting of -1 usually chooses primal simplex.

Note: Only affects continuous multi-objective models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MultiObjPre

Initial presolve level on multi-objective models

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls the initial presolve level used for multi-objective models. Value 0 disables the initial presolve, value 1 applies presolve conservatively, and value 2 applies presolve aggressively. The default -1 value usually applies presolve conservatively. Aggressive presolve may increase the chance of the objective values being slightly different than those for other options.

Note: Only affects multi-objective models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

MultiObjSettings

Create multi-objective settings from a list of .prm files

Type: string
Default value: ""

This command-line only parameter allows you to specify a comma-separated list of .prm files that are used to set parameters for the different solves in a multi-objective model.

To give an example, you could create two `.prm` files with the following contents...

vb0.prm:

```
VarBranch 0
```

vb1.prm:

```
VarBranch 1
```

Issuing the command `gurobi_cl MultiObjSettings=vb0.prm,vb1.prm model.mps` would use different branching strategies when solving for the two objectives of the multi-objective model.

Note: Command-line only
Note: Only affects multi-objective models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NetworkAlg

Network simplex algorithm

Type: int
Default value: -1
Minimum value: -1
Maximum value: 1

Controls whether to use network simplex. Value 0 doesn't use network simplex. Value 1 indicates to use network simplex, if an LP is a network problem. The default -1 value chooses automatically.

Note: Only affects linear programming (LP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NetworkCuts

Network cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls network cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NLPHeur

Controls the NLP heuristic

Type: int
Default value: 1
Minimum value: 0
Maximum value: 1

The NLP heuristic uses a non-linear barrier solver to find feasible solutions to non-convex quadratic models. It can often find solutions much more quickly than the alternative, but in some cases it can consume significant runtime without producing a solution. By default, the heuristic is enabled (1). Use 0 to disable the heuristic.

Note: Only affects models with nonconvex quadratic expressions in the objective or constraints

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NodefileDir

Directory for node files

Type: string
Default value: "."

Determines the directory into which nodes are written when node memory usage exceeds the specified NodefileStart value.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NodefileStart

Write MIP nodes to disk

Type: double
Default value: Infinity
Minimum value: 0
Maximum value: Infinity

If you find that the Gurobi Optimizer exhausts memory when solving a MIP, you should modify the `NodefileStart` parameter. When the amount of memory used to store nodes (measured in GB, i.e., 10^9 bytes) exceeds the specified parameter value, nodes are compressed and written to disk. We recommend a setting of `0.5`, but you may wish to choose a different value, depending on the memory available in your machine. By default, nodes are written to the current working directory. The NodefileDir parameter can be used to choose a different location.

If you still exhaust memory after setting the `NodefileStart` parameter to a small value, you should try limiting the thread count. Each thread in parallel MIP requires a copy of the model, as well as several other large data structures. Reducing the Threads parameter can sometimes significantly reduce memory usage.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NodeLimit

MIP node limit

Type: double
Default value: Infinity
Minimum value: 0
Maximum value: Infinity

Limits the number of MIP nodes explored. Optimization returns with an `NODE_LIMIT` status if the limit is exceeded (see the Status Code section for further details). Note that if multiple threads

are used for the optimization, the actual number of explored nodes may be slightly larger than the set limit.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NodeMethod

Method used to solve MIP node relaxations

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Algorithm used for MIP node relaxations (except for the initial root node relaxation, see Method). Options are: -1=automatic, 0=primal simplex, 1=dual simplex, and 2=barrier. Note that barrier is not an option for MIQP node relaxations.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NonConvex

Strategy for handling non-convex quadratic programs

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Sets the strategy for handling non-convex quadratic objectives or non-convex quadratic constraints. With setting 0, an error is reported if the original user model contains non-convex quadratic constructs (unless Q matrix linearization, as controlled by the PreQLinearize parameter, removes the non-convexity). With setting 1, an error is reported if non-convex quadratic constructs could not be discarded or linearized during presolve. With setting 2, non-convex quadratic problems are solved by translating them into bilinear form and applying spatial branching. The default -1 setting is currently almost equivalent to 2, except that it takes less care to avoid presolve reductions that might transform a convex constraint into one that can no longer be detected to be convex, and thus can sometimes perform more presolve reductions.

Note: Only affects QP, QCP, MIQP, and MIQCP models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NoRelHeurTime

Limits the amount of time spent in the NoRel heuristic

Type:	double
Default value:	0
Minimum value:	0
Maximum value:	Infinity

Limits the amount of time (in seconds) spent in the NoRel heuristic. This heuristic searches for high-quality feasible solutions before solving the root relaxation. It can be quite useful on models where the root relaxation is particularly expensive.

Note that this parameter will introduce non-determinism - different runs may take different paths. Use the NoRelHeurWork parameter for deterministic results.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NoRelHeurWork

Limits the amount of work spent in the NoRel heuristic

Type:	double
Default value:	0
Minimum value:	0
Maximum value:	Infinity

Limits the amount of work spent in the NoRel heuristic. This heuristic searches for high-quality feasible solutions before solving the root relaxation. It can be quite useful on models where the root relaxation is particularly expensive.

The work metric used in this parameter is tough to define precisely. A single unit corresponds to roughly a second, but this will depend on the machine, the core count, and in some cases the model. You may need to experiment to find a good setting for your model.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NormAdjust

Choose simplex pricing norm.

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	3

Chooses from among multiple pricing norm variants. The details of how this parameter affects the simplex pricing algorithm are subtle and difficult to describe, so we've simply labeled the options 0 through 3. The default value of -1 chooses automatically.

Changing the value of this parameter rarely produces a significant benefit.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

NumericFocus

Numerical focus

Type: int
Default value: 0
Minimum value: 0
Maximum value: 3

The `NumericFocus` parameter controls the degree to which the code attempts to detect and manage numerical issues. The default setting (0) makes an automatic choice, with a slight preference for speed. Settings 1-3 increasingly shift the focus towards being more careful in numerical computations. With higher values, the code will spend more time checking the numerical accuracy of intermediate results, and it will employ more expensive techniques in order to avoid potential numerical issues.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

OBBT

Controls aggressiveness of optimality-based bound tightening

Type: int
Default value: -1
Minimum value: -1
Maximum value: 3

Value 0 disables optimality-based bound tightening (OBBT). Levels 1-3 describe the amount of work allowed for OBBT ranging from moderate to aggressive. The default -1 value is an automatic setting which chooses a rather moderate setting.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ObjNumber

Selects objective index of multi-objectives

Type: int
Default value: 0
Minimum value: 0
Maximum value: MAXINT

When working with multiple objectives, this parameter selects the index of the objective you want to work with. When you query or modify an attribute associated with multiple objectives (ObjN, ObjNVal, etc.), the `ObjNumber` parameter will determine which objective is actually affected. The value of this parameter should be less than the value of the NumObj attribute (which captures the number of objectives in the model).

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ObjScale

Objective scaling

Type: double
Default value: 0.0
Minimum value: -1
Maximum value: Infinity

When positive, divides the model objective by the specified value to avoid numerical issues that may result from very large or very small objective coefficients. The default value of 0 decides on the scaling automatically. A value less than zero uses the maximum coefficient to the specified power as the scaling (so `ObjScale=-0.5` would scale by the square root of the largest objective coefficient).

Note that objective scaling can lead to large dual violations on the original, unscaled objective when the optimality tolerance with the scaled objective is barely satisfied, so it should be used sparingly. Note also that scaling will be more effective when all objective coefficients are of similar orders of magnitude, as opposed to objectives with a wide range of coefficients. In the latter case, consider using the Multiple Objectives feature instead.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

OptimalityTol

Dual feasibility tolerance

Type: double
Default value: 1e-6
Minimum value: 1e-9
Maximum value: 1e-2

Reduced costs must all be smaller than `OptimalityTol` in the improving direction in order for a model to be declared optimal.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

OutputFlag

Controls Gurobi output

Type: int
Default value: 1
Minimum value: 0
Maximum value: 1

Enables or disables solver output. Use LogFile and LogToConsole for finer-grain control. Setting `OutputFlag` to 0 is equivalent to setting LogFile to `""` and LogToConsole to 0.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PartitionPlace

Controls where the partition heuristic runs

Type: int
Default value: 15
Minimum value: 0
Maximum value: 31

Setting the Partition attribute on at least one variable in a model enables the partitioning heuristic, which uses large-neighborhood search to try to improve the current incumbent solution.

This parameter determines where that heuristic runs. Options are:

- Before the root relaxation is solved (16)
- At the start of the root cut loop (8)
- At the end of the root cut loop (4)
- At the nodes of the branch-and-cut search (2)
- When the branch-and-cut search terminates (1)

The parameter value is a bit vector, where each bit turns the heuristic on or off at that place. The numerical values next to the options listed above indicate which bit controls the corresponding

option. Thus, for example, to enable the heuristic at the beginning and end of the root cut loop (and nowhere else), you would set the 8 bit and the 4 bit to 1, which would correspond to a parameter value of 12.

The default value of 15 indicates that we enable every option except the first one listed above.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PerturbValue

Simplex perturbation

Type:	double
Default value:	0.0002
Minimum value:	0
Maximum value:	Infinity

Magnitude of the simplex perturbation. Note that perturbation is only applied when progress has stalled, so the parameter will often have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PoolGap

Maximum relative gap for stored solutions

Type:	double
Default value:	Infinity
Minimum value:	0
Maximum value:	Infinity

Determines how large a (relative) gap to tolerate in stored solutions. When this parameter is set to a non-default value, solutions whose objective values exceed that of the best known solution by more than the specified (relative) gap are discarded. For example, if the MIP solver has found a solution at objective 100, then a setting of `PoolGap=0.2` would discard solutions with objective worse than 120 (assuming a minimization objective).

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PoolGapAbs

Maximum absolute gap for stored solutions

Type: double
Default value: Infinity
Minimum value: 0
Maximum value: Infinity

Determines how large a (absolute) gap to tolerate in stored solutions. When this parameter is set to a non-default value, solutions whose objective values exceed that of the best known solution by more than the specified (absolute) gap are discarded. For example, if the MIP solver has found a solution at objective 100, then a setting of `PoolGapAbs=20` would discard solutions with objective worse than 120 (assuming a minimization objective).

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PoolSearchMode

Selects different modes for exploring the MIP search tree

Type: int
Default value: 0
Minimum value: 0
Maximum value: 2

Selects different modes for exploring the MIP search tree. With the default setting (`PoolSearchMode=0`), the MIP solver tries to find an optimal solution to the model. It keeps other solutions found along the way, but those are incidental. By setting this parameter to a non-default value, the MIP search will continue after the optimal solution has been found in order to find additional, high-quality solutions. With a non-default value (`PoolSearchMode=1` or `PoolSearchMode=2`), the MIP solver will try to find `n` solutions, where `n` is determined by the value of the PoolSolutions parameter. With a setting of 1, there are no guarantees about the quality of the extra solutions, while with a setting of 2, the solver will find the `n` best solutions. The cost of the solve will increase with increasing values of this parameter.

Once optimization is complete, the PoolObjBound attribute can be used to evaluate the quality of the solutions that were found. For example, a value of `PoolObjBound=100` indicates that there are no other solutions with objective better 100, and thus that any known solutions with objective better than 100 are better than any as-yet undiscovered solutions.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PoolSolutions

Number of MIP solutions to store

Type: int
Default value: 10
Minimum value: 1
Maximum value: MAXINT

Determines how many MIP solutions are stored. For the default value of PoolSearchMode, these are just the solutions that are found along the way in the process of exploring the MIP search tree. For other values of `PoolSearchMode`, this parameter sets a target for how many solutions to find, so larger values will impact performance.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PreCrush

Controls presolve reductions that affect user cuts

Type: int
Default value: 0
Minimum value: 0
Maximum value: 1

Shuts off a few reductions in order to allow presolve to transform any constraint on the original model into an equivalent constraint on the presolved model. You should consider setting this parameter to 1 if you are using callbacks to add your own cuts. A cut that cannot be applied to the presolved model will be silently ignored. The impact on the size of the presolved problem is usually small.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PreDepRow

Controls the presolve dependent row reduction

Type: int
Default value: -1
Minimum value: -1
Maximum value: 1

Controls the presolve dependent row reduction, which eliminates linearly dependent constraints from the constraint matrix. The default setting (-1) applies the reduction to continuous models but not to MIP models. Setting 0 turns the reduction off for all models. Setting 1 turns it on for all models.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PreDual

Controls presolve model dualization

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls whether presolve forms the dual of a continuous model. Depending on the structure of the model, solving the dual can reduce overall solution time. The default setting uses a heuristic to decide. Setting 0 forbids presolve from forming the dual, while setting 1 forces it to take the dual. Setting 2 employs a more expensive heuristic that forms both the presolved primal and dual models (on two threads), and heuristically chooses one of them.

Note: Mainly affects LP, QP, and QCP models, but it is also used for the initial root relaxation of mixed integer programs.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PreMIQCPForm

Format of presolved MIQCP model

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Determines the format of the presolved version of an MIQCP model. Option 0 leaves the model in MIQCP form, so the branch-and-cut algorithm will operate on a model with arbitrary quadratic constraints. Option 1 always transforms the model into MISOCP form; quadratic constraints are transformed into second-order cone constraints. Option 2 always transforms the model into disaggregated MISOCP form; quadratic constraints are transformed into rotated cone constraints, where each rotated cone contains two terms and involves only three variables.

The default setting (-1) choose automatically. The automatic setting works well, but there are cases where forcing a different form can be beneficial.

Note: Only affects MIQCP models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PrePasses

Presolve pass limit

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	MAXINT

Limits the number of passes performed by presolve. The default setting (-1) chooses the number of passes automatically. You should experiment with this parameter when you find that presolve is consuming a large fraction of total solve time.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PreQLinearize

Presolve quadratic linearization

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls presolve Q matrix linearization. Binary variables in quadratic expressions provide some freedom to state the same expression in multiple different ways. Options 1 and 2 of this parameter attempt to linearize quadratic constraints or a quadratic objective, replacing quadratic terms with linear terms, using additional variables and linear constraints. This can potentially transform an MIQP or MIQCP model into an MILP. Option 1 focuses on producing an MILP reformulation with a strong LP relaxation, with a goal of limiting the size of the MIP search tree. Option 2 aims for a compact reformulation, with a goal of reducing the cost of each node. Option 0 attempts to leave Q matrices unmodified; it won't add variables or constraints, but it may still perform adjustments on quadratic objective functions to make them positive semi-definite (PSD). The default setting (-1) chooses automatically.

Note: Only affects MIQP and MIQCP models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Presolve

Controls the presolve level

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls the presolve level. A value of -1 corresponds to an automatic setting. Other options are off (0), conservative (1), or aggressive (2). More aggressive application of presolve takes more time, but can sometimes lead to a significantly tighter model.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PreSOS1BigM

Threshold for SOS1-to-binary reformulation

Type: double
Default value: -1
Minimum value: -1
Maximum value: 1e10

Controls the automatic reformulation of SOS1 constraints into binary form. SOS1 constraints are often handled more efficiently using a binary representation. The reformulation often requires `big-M` values to be introduced as coefficients. This parameter specifies the largest `big-M` that can be introduced by presolve when performing this reformulation. Larger values increase the chances that an SOS1 constraint will be reformulated, but very large values (e.g., 1e8) can lead to numerical issues.

The default value of -1 chooses a threshold automatically. You should set the parameter to 0 to shut off SOS1 reformulation entirely, or a large value to force reformulation.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Please refer to this section for more information on SOS constraints.

PreSOS1Encoding

Encoding used for SOS1 reformulation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 3

Controls the automatic reformulation of SOS1 constraints. Such constraints can be handled directly by the MIP branch-and-cut algorithm, but they are often handled more efficiently by reformulating them using binary or integer variables. There are several diffent ways to perform this reformulation; they differ in their size and strength. Smaller reformulations add fewer variables and constraints to the model. Stronger reformulations reduce the number of branch-and-cut nodes required to solve the resulting model.

Options 0 and 1 of this parameter encode an SOS1 constraint using a formulation whose size is linear in the number of SOS members. Option 0 uses a so-called multiple choice model. It usually produces an LP relaxation that is easier to solve. Option 1 uses an incremental model. It often gives a stronger representation, reducing the amount of branching required to solve harder problems.

Options 2 and 3 of this parameter encode the SOS1 using a formulation of logarithmic size. They both only apply when all the variables in the SOS1 are non-negative. Option 3 additionally requires that the sum of the variables in the SOS1 is equal to 1. Logarithmic formulations are often advantageous when the SOS1 constraint has a large number of members. Option 2 focuses on a formulation whose LP relaxation is easier to solve, while option 3 has better branching behavior.

The default value of -1 chooses a reformulation for each SOS1 constraint automatically.

Note that the reformulation of SOS1 constraints is also influenced by the PreSOS1BigM parameter. To shut off the reformulation entirely you should set that parameter to 0.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Please refer to this section for more information on SOS constraints.

PreSOS2BigM

Threshold for SOS2-to-binary reformulation

Type: double
Default value: -1
Minimum value: -1
Maximum value: 1e10

Controls the automatic reformulation of SOS2 constraints into binary form. SOS2 constraints are often handled more efficiently using a binary representation. The reformulation often requires `big-M` values to be introduced as coefficients. This parameter specifies the largest `big-M` that can be introduced by presolve when performing this reformulation. Larger values increase the chances that an SOS2 constraint will be reformulated, but very large values (e.g., 1e8) can lead to numerical issues.

The default value of -1 chooses a threshold automatically. You should set the parameter to 0 to shut off SOS2 reformulation entirely, or a large value to force reformulation.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Please refer to this section for more information on SOS constraints.

PreSOS2Encoding

Encoding used for SOS2 reformulation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 3

Controls the automatic reformulation of SOS2 constraints. Such constraints can be handled directly by the MIP branch-and-cut algorithm, but they are often handled more efficiently by reformulating them using binary or integer variables. There are several diffent ways to perform this reformulation; they differ in their size and strength. Smaller reformulations add fewer variables and constraints to the model. Stronger reformulations reduce the number of branch-and-cut nodes required to solve the resulting model.

Options 0 and 1 of this parameter encode an SOS2 constraint using a formulation whose size is linear in the number of SOS members. Option 0 uses a so-called multiple choice model. It usually produces an LP relaxation that is easier to solve. Option 1 uses an incremental model. It often gives a stronger representation, reducing the amount of branching required to solve harder problems.

Options 2 and 3 of this parameter encode the SOS2 using a formulation of logarithmic size. They both only apply when all the variables in the SOS2 are non-negative. Option 3 additionally requires that the sum of the variables in the SOS2 is equal to 1. Logarithmic formulations are often advantageous when the SOS2 constraint has a large number of members. Option 2 focuses on a formulation whose LP relaxation is easier to solve, while option 3 has better branching behavior.

The default value of -1 chooses a reformulation for each SOS2 constraint automatically.

Note that the reformulation of SOS2 constraints is also influenced by the PreSOS2BigM parameter. To shut off the reformulation entirely you should set that parameter to 0.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Please refer to this section for more information on SOS constraints.

PreSparsify

Controls the presolve sparsify reduction

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls the presolve sparsify reduction. This reduction can sometimes significantly reduce the number of non-zero values in the presolved model. Value 0 shuts off the reduction, while value 1 forces it on for mixed integer programming (MIP) models and value 2 forces it on for all types of models, including linear programming (LP) models, and MIP relaxations. The default value of -1 chooses automatically.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ProjImpliedCuts

Projected implied bound cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls projected implied bound cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PSDCuts

PSD cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls PSD cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects models with nonconvex quadratic expressions in the objective or constraints

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PSDTol

Positive semi-definite tolerance

Type: double
Default value: 1e-6
Minimum value: 0
Maximum value: Infinity

Sets a limit on the amount of diagonal perturbation that the optimizer is allowed to perform on a Q matrix in order to correct minor PSD violations. If a larger perturbation is required, the optimizer will terminate with a `GRB_ERROR_Q_NOT_PSD` error.

Note: Only affects QP, QCP, MIQP, and MIQCP models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

PumpPasses

Passes of the feasibility pump heuristic

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	MAXINT

Number of passes of the feasibility pump heuristic. Note that this heuristic is only applied at the end of the MIP root.

This heuristic is quite expensive, and generally produces poor quality solutions. You should generally only use it if other means, including exploration of the tree with default settings, fail to produce a feasible solution.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

QCPDual

Dual variables for QCP models

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	1

Determines whether dual variable values are computed for QCP models. Computing them can add significant time to the optimization, so you should only set this parameter to 1 if you need them.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Quad

Controls quad precision in simplex

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	1

Enables or disables quad precision computation in simplex. The -1 default setting allows the algorithm to decide. Quad precision can sometimes help solve numerically challenging models,

but it can also significantly increase runtime. Quad precision is only available on processors that support quadruple precision, e.g., common Intel processors. On other processors, the parameter has no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Record

Enables API call recording

Type: int
Default value: 0
Minimum value: 0
Maximum value: 1

Enables API call recording. When enabled, Gurobi will write one or more files (named `gurobi000.grbr` or similar) that capture the sequence of Gurobi commands that your program issued. This file can subsequently be replayed using the Gurobi command-line tool. Replaying the file will repeat the exact same sequence of commands, and when completed will show the time spent in Gurobi API routines, the time spent in Gurobi algorithms, and will indicate whether any Gurobi environments or models were leaked by your program. Replay files are particularly useful in tech support situations. They provide an easy way to relay to Gurobi tech support the exact sequence of Gurobi commands that led to a question or issue.

This parameter must be set before starting an empty environment (or in a `gurobi.env` file). All Gurobi commands will be recorded until the environment is freed or the program ends.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ResultFile

Write a result file upon completion of optimization

Type: string
Default value: ""

Specifies the name of the result file to be written upon completion of optimization. The type of the result file is determined by the file suffix. The most commonly used suffixes are `.sol` (to capture the solution vector), `.bas` (to capture the simplex basis), and `.mst` (to capture the solution vector on the integer variables). You can also write a `.ilp` file (to capture the IIS for an infeasible model), or a `.mps`, `.rew`, `.lp`, or `.rlp` file (to capture the original model), or a `.dua` or `.dlp` file (to capture the dual of a pure LP model). The file suffix may optionally be followed by `.gz`, `.bz2`, or `.7z`, which produces a compressed result.

More information on the file formats can be found in the File Format section.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

RINS

Relaxation Induced Neighborhood Search (RINS) heuristic frequency

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	MAXINT

Frequency of the RINS heuristic. Default value (-1) chooses automatically. A value of 0 shuts off RINS. A positive value `n` applies RINS at every `n-th` node of the MIP search tree.

Increasing the frequency of the RINS heuristic shifts the focus of the MIP search away from proving optimality, and towards finding good feasible solutions. We recommend that you try MIPFocus, ImproveStartGap, or ImproveStartTime before experimenting with this parameter.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

RelaxLiftCuts

Relax-and-lift cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls relax-and-lift cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

RLTCuts

RLT cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls Relaxation Linearization Technique (RLT) cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ScaleFlag

Model scaling

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	3

Controls model scaling. By default, the rows and columns of the model are scaled in order to improve the numerical properties of the constraint matrix. The scaling is removed before the final solution is returned. Scaling typically reduces solution times, but it may lead to larger constraint violations in the original, unscaled model. Turning off scaling (`ScaleFlag=0`) can sometimes produce smaller constraint violations. Choosing a different scaling option can sometimes improve performance for particularly numerically difficult models. Using geometric mean scaling (`ScaleFlag=2`) is especially well suited for models with a wide range of coefficients in the constraint matrix rows or columns. Settings 1 and 3 are not as directly connected to any specific model characteristics, so experimentation with both settings may be needed to assess performance impact.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ScenarioNumber

Selects scenario index of multi-scenario models

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	MAXINT

When working with multiple scenarios, this parameter selects the index of the scenario you want to work with. When you query or modify an attribute associated with multiple scenarios (ScenNLB, ScenNUB, ScenNObj, ScenNRHS, etc.), the `ScenarioNumber` parameter will determine which scenario is actually affected. The value of this parameter should be less than the value of the NumScenarios attribute (which captures the number of scenarios in the model).

Please refer to the discussion of Multiple Scenarios for more information on the use of alternative scenarios.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an

integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Seed

Random number seed

Type: int
Default value: 0
Minimum value: 0
Maximum value: MAXINT

Modifies the random number seed. This acts as a small perturbation to the solver, and typically leads to different solution paths.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ServerPassword

Client password for Remote Services cluster or token server

Type: string
Default value: ""

The password for connecting to the server (either a Compute Server or a token server).

For connecting to the Remote Services cluster referred to by the ComputeServer parameter, you'll need to supply the client password. Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs.

Supply the token server password (if needed) when connecting to the server referred to by the TokenServer parameter,

You must set this parameter through either a `gurobi.lic` file (using `PASSWORD=pwd`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ServerTimeout

Network timeout

Type: int
Default value: 60
Minimum value: 1
Maximum value: MAXINT

Network time-out for Compute Server and token server (in seconds). If the client program is unable to contact the server for more than the specified amount of time, the client will quit with a network error.

Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server jobs.

You must set this parameter using an empty environment. Changing the parameter after your environment has been created will have no effect.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Sifting

Controls sifting within dual simplex

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Enables or disables sifting within dual simplex. Sifting can be useful for LP models where the number of variables is many times larger than the number of constraints (we typically only see significant benefits when the ratio is 100 or more). Options are Automatic (-1), Off (0), Moderate (1), and Aggressive (2). With a Moderate setting, sifting will be applied to LP models and to the initial root relaxation for MIP models. With an Aggressive setting, sifting will be applied any time dual simplex is used, including at the nodes of a MIP. Note that this parameter has no effect if you aren't using dual simplex. Note also that Gurobi will ignore this parameter in cases where sifting is obviously a worse choice than dual simplex.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SiftMethod

LP method used to solve sifting sub-problems

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

LP method used to solve sifting sub-problems. Options are Automatic (-1), Primal Simplex (0), Dual Simplex (1), and Barrier (2). Note that this parameter only has an effect when you are using dual simplex and sifting has been selected (either automatically by dual simplex, or through the Sifting parameter).

Changing the value of this parameter rarely produces a significant benefit.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SimplexPricing

Simplex pricing strategy

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	3

Determines the simplex variable pricing strategy. Available options are Automatic (-1), Partial Pricing (0), Steepest Edge (1), Devex (2), and Quick-Start Steepest Edge (3).

Changing the value of this parameter rarely produces a significant benefit.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SoftMemLimit

Soft memory limit

Type:	double
Default value:	Infinity
Minimum value:	0
Maximum value:	Infinity

Limits the total amount of memory (in GB, i.e., 10^9 bytes) available to Gurobi. If more is needed, Gurobi will terminate with a `MEM_LIMIT` status code (see the Status Code section for further details).

In contrast to the MemLimit parameter, the `SoftMemLimit` parameter leads to a graceful exit of the optimization, such that it is possible to retrieve solution information afterwards or (in the case of a MIP solve) resume the optimization.

A disadvantage compared to MemLimit is that the `SoftMemLimit` is only checked at places where optimization can be terminated gracefully, so memory use may exceed the limit between these checks.

Note that allocated memory is tracked across all models within a Gurobi environment. If you create multiple models in one environment, these additional models will count towards overall memory consumption.

Memory usage is also tracked across all threads. One consequence of this is that termination may be non-deterministic for multi-threaded runs.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SolutionLimit

MIP solution limit

Type: int
Default value: MAXINT
Minimum value: 1
Maximum value: MAXINT

Limits the number of feasible MIP solutions found. Optimization returns with a `SOLUTION_LIMIT` status once the limit has been reached (see the Status Code section for further details). To find a feasible solution quickly, Gurobi executes additional feasible point heuristics when the solution limit is set to exactly 1.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SolutionTarget

Solution Target for LP

Type: int
Default value: -1
Minimum value: -1
Maximum value: 1

Specifies the solution target for linear programs (LP). Options are Automatic (-1), primal and dual optimal, and basic (0), primal and dual optimal (1).

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SolFiles

Location to store intermediate solution files

Type: string
Default value: ""

During the MIP solution process, multiple incumbent solutions are typically found on the path to finding a proven optimal solution. Setting this parameter to a non-empty string causes these solutions to be written to files (in .sol format) as they are found. The MIP solver will append `_n.sol` to the value of the parameter to form the name of the file that contains solution number n. For example, setting the parameter to value `solutions/mymodel` will create files `mymodel_0.sol`, `mymodel_1.sol`, etc., in directory `solutions`.

Note that intermediate solutions can be retrieved as they are generated through a callback (by requesting the `MIPSOL_SOL` in a `MIPSOL` callback). This parameter makes the process simpler.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SolutionNumber

Select a sub-optimal MIP solution

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	MAXINT

When querying attribute Xn, ObjNVal, or PoolObjVal to retrieve an alternate MIP solution, this parameter determines which alternate solution is retrieved. The value of this parameter should be less than the value of the SolCount attribute.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

StartNodeLimit

Limit MIP start sub-MIP nodes

Type:	int
Default value:	-1
Minimum value:	-3
Maximum value:	MAXINT

This parameter limits the number of branch-and-bound nodes explored when completing a partial MIP start. The default value of -1 uses the value of the `SubMIPNodes` parameter. A value of -2 means to only check full MIP starts for feasibility and to ignore partial MIP starts. A value of -3 shuts off MIP start processing entirely. Non-negative values are node limits.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

StartNumber

Selects MIP start index

Type:	int
Default value:	0
Minimum value:	-1
Maximum value:	MAXINT

This parameter selects the index of the MIP start you want to work with. When you modify a MIP start value (using the Start attribute) the `StartNumber` parameter will determine which MIP start is actually affected. The value of this parameter should be less than the value of the NumStart attribute (which captures the number of MIP starts in the model).

The special value -1 is meant to append new MIP start to a model, but querying a MIP start when StartNumber is -1 will result in an error.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

StrongCGCuts

Strong-CG cut generation

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	2

Controls Strong Chvátal-Gomory (Strong-CG) cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SubMIPCuts

Sub-MIP cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls sub-MIP cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

SubMIPNodes

Nodes explored in sub-MIP heuristics

Type: int
Default value: 500
Minimum value: 0
Maximum value: MAXINT

Limits the number of nodes explored by MIP-based heuristics (such as RINS). Exploring more nodes can produce better solutions, but it generally takes longer.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Symmetry

Symmetry detection

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls symmetry detection. A value of -1 corresponds to an automatic setting. Other options are off (0), conservative (1), or aggressive (2).

Symmetry can impact a number of different parts of the algorithm, including presolve, the MIP tree search, and the LP solution process. Default settings are quite effective, so changing the value of this parameter rarely produces a significant benefit.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Threads

Thread count

Type: int
Default value: 0
Minimum value: 0
Maximum value: The number of virtual processors available to the process running Gurobi

Controls the number of threads to apply to parallel algorithms (concurrent LP, parallel barrier, parallel MIP, etc.). The default value of 0 is an automatic setting. It will generally use as many threads as there are virtual processors. The number of virtual processors may exceed the number of cores due to hyperthreading or other similar hardware features.

While you will generally get the best performance by using all available cores in your machine, there are a few exceptions. One is of course when you are sharing a machine with other jobs. In this case, you should select a thread count that doesn't oversubscribe the machine.

We have also found that certain classes of MIP models benefit from reducing the thread count, often all the way down to one thread. Starting multiple threads introduces contention for machine resources. For classes of models where the first solution found by the MIP solver is almost always optimal, and that solution isn't found at the root, it is often better to allow a single thread to explore the search tree uncontested.

Another situation where reducing the thread count can be helpful is when memory is tight. Each thread can consume a significant amount of memory.

We've made the pragmatic choice to impose a soft limit of 32 threads for the automatic setting (0). If your machine has more, and you find that using more increases performance, you should feel free to set the parameter to a larger value.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TimeLimit

Time limit

Type: double
Default value: Infinity
Minimum value: 0
Maximum value: Infinity

Limits the total time expended (in seconds). Optimization returns with a `TIME_LIMIT` status if the limit is exceeded (see the Status Code section for further details).

Note that optimization may not stop immediately upon hitting the time limit. It will stop after performing the required additional computations of the attributes associated with the terminated optimization. As a result, the Runtime attribute may be larger than the specified TimeLimit upon

completion, and repeating the optimization with a TimeLimit set to the Runtime attribute of the stopped optimization may result in additional computations and a larger attribute value.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TokenServer

Name of your token server

Type: string
Default value: ""

When using a token license, set this parameter to the name of the token server. You can refer to the server using its name or its IP address.

You can provide a comma-separated list of token servers to increase robustness. If the first server in the list doesn't respond, the second will be tried, etc.

You must set this parameter through either a `gurobi.lic` file (using `TOKENSERVER=server`) or an empty environment. Changing the parameter after your environment has been created will have no effect.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TSPort

Port for token server

Type: int
Default value: 41954
Minimum value: 0
Maximum value: 65536

Port to use when connecting to the Gurobi token server. You should only change this if your network administrator tells you to.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneBaseSettings

Comma-separated list of base parameter settings

Type: string
Default value: ""

A list of parameter files (e.g., `base1.prm,base2.prm`) that define settings that should be tried first during the tuning process, in the order they are given. Default parameter settings will also be tried, but after the settings provided in these files. You can include an empty parameter file in the list if you would like default settings to be tried earlier.

Note: Command-line only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneCleanup

Enables a tuning cleanup phase

Type:	double
Default value:	0.0
Minimum value:	0.0
Maximum value:	1.0

Enables a cleanup phase at the end of tuning. The parameter indicates the percentage of total tuning time to devote to this phase, with a goal of reducing the number of parameter changes required to achieve the best tuning result.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneCriterion

Tuning criterion

Type:	int
Default value:	-1
Minimum value:	-1
Maximum value:	3

Modifies the tuning criterion for the tuning tool. The primary tuning criterion is always to minimize the runtime required to find a proven optimal solution. However, for MIP models that don't solve to optimality within the specified time limit, a secondary criterion is needed. Set this parameter to 1 to use the optimality gap as the secondary criterion. Choose a value of 2 to use the objective of the best feasible solution found. Choose a value of 3 to use the best objective bound. Choose 0 to ignore the secondary criterion and focus entirely on minimizing the time to find a proven optimal solution. The default value of -1 chooses automatically.

Note that for multi-objective problems value 1 and 3 are unsupported. See the Multiple Objectives section for more details on this.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneDynamicJobs

Dynamic distributed tuning job count

Type:	int
Default value:	0
Minimum value:	-1
Maximum value:	MAXINT

Enables distributed parallel tuning, which can significantly increase the performance of the tuning tool. A value of `n` causes the tuning tool to use a dynamic set of up to `n` workers in parallel. These workers are used for a limited amount of time and afterwards potentially released so that they are available for other remote jobs. A value of `-1` allows the solver to use an unlimited number of workers. Note that this parameter can be combined with TuneJobs to get a static set of workers and a dynamic set of workers for distributed tuning. You can use the WorkerPool parameter to provide a distributed worker cluster.

Note that distributed tuning is most effective when the worker machines have similar performance. Distributed tuning doesn't attempt to normalize performance by server, so it can incorrectly attribute a boost in performance to a parameter change when the associated setting is tried on a worker that is significantly faster than the others.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneJobs

Permanent distributed tuning job count

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	MAXINT

Enables distributed parallel tuning, which can significantly increase the performance of the tuning tool. A value of `n` causes the tuning tool to use a static set of up to `n` workers in parallel. Such workers are kept for the whole tuning run. Note that this parameter can be combined with TuneDynamicJobs to get a static set of workers and a dynamic set of workers for distributed tuning. You can use the WorkerPool parameter to provide a distributed worker cluster.

Note that distributed tuning is most effective when the worker machines have similar performance. Distributed tuning doesn't attempt to normalize performance by server, so it can incorrectly attribute a boost in performance to a parameter change when the associated setting is tried on a worker that is significantly faster than the others.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneMetric

Method for aggregating tuning results

Type: int
Default value: -1
Minimum value: -1
Maximum value: 1

A single tuning run typically produces multiple timing results for each candidate parameter set, either as a result of performing multiple trials, or tuning multiple models, or both. This parameter controls how these results are aggregated into a single measure. The default setting (-1) chooses the aggregation automatically; setting 0 computes the average of all individual results; setting 1 takes the maximum.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneOutput

Tuning output level

Type: int
Default value: 2
Minimum value: 0
Maximum value: 3

Controls the amount of output produced by the tuning tool. Level 0 produces no output; level 1 produces tuning summary output only when a new best parameter set is found; level 2 produces tuning summary output for each parameter set that is tried; level 3 produces tuning summary output, plus detailed solver output, for each parameter set tried.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneResults

Number of improved parameter sets returned

Type: int
Default value: -1
Minimum value: -2
Maximum value: MAXINT

The tuning tool often finds multiple parameter sets that improve over the baseline settings. This parameter controls how many of these sets should be retained when tuning is complete. A non-negative value indicates how many sets should be retained. The default value (-1) retains the efficient frontier of parameter sets. That is, it retains the best set for one changed parameter, the best for two changed parameters, etc. Sets that aren't on the efficient frontier are discarded. If you interested in all the sets, use value -2 for the parameter.

Note that the first set in the results is always the set of parameters which was used for the first solve, the baseline settings. This set serves as the base for any improvement. So if you are interested in the best tuned set of parameters you need to request at least 2 tune results. The first

one (with index 0) will be the baseline setting and the second one (with index 1) will be the best set found during tuning.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneTargetMIPGap

A target gap to be reached

Type: double
Default value: 0
Minimum value: 0
Maximum value: Infinity

A target gap to be reached. As soon as the tuner has found parameter settings that allow Gurobi to reach the target gap for the given model(s), it stops trying to improve parameter settings further. Instead, the tuner switches into the cleanup phase (see TuneCleanup parameter).

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneTargetTime

A target runtime in seconds to be reached

Type: double
Default value: 0.005
Minimum value: 0
Maximum value: Infinity

A target runtime in seconds to be reached. As soon as the tuner has found parameter settings that allow Gurobi to solve the model(s) within the target runtime, it stops trying to improve parameter settings further. Instead, the tuner switches into the cleanup phase (see TuneCleanup parameter).

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneTimeLimit

Tuning tool time limit

Type: double
Default value: Infinity
Minimum value: 0
Maximum value: Infinity

Limits total tuning runtime (in seconds). The default setting chooses a time limit automatically.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneTrials

Perform multiple runs on each parameter set to limit the effect of random noise

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	MAXINT

Performance on a MIP model can sometimes experience significant variations due to random effects. As a result, the tuning tool may return parameter sets that improve on the baseline only due to randomness. This parameter allows you to perform multiple solves for each parameter set, using different Seed values for each, in order to reduce the influence of randomness on the results. The default value of 0 indicates an automatic choice that depends on model characteristics.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

TuneUseFilename

Use model file names as model names

Type:	int
Default value:	0
Minimum value:	0
Maximum value:	1

During tuning, each model is referred to using a name (e.g., when displaying progress information for that model). By default, the model name comes from the contents of the model file. If this parameter is set to 1 before calling `grbtune`, tuning will refer to a model using the name of the model file instead. This can be helpful when several models being tuned have the same (or no) names.

Note: Command-line only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

UpdateMode

Changes the behavior of lazy updates

Type: int
Default value: 1
Minimum value: 0
Maximum value: 1

Determines how newly added variables and linear constraints are handled. The default setting (1) allows you to use new variables and constraints immediately for building or modifying the model. A setting of 0 requires you to call `update` before these can be used.

Since the vast majority of programs never query Gurobi for details about the optimization models they build, the default setting typically removes the need to call `update`, or even be aware of the details of our *lazy update* approach for handling model modifications. However, these details will show through when you try to query modified model information.

In the Gurobi interface, model modifications (bound changes, right-hand side changes, objective changes, etc.) are placed in a queue. These queued modifications are applied to the model at three times: when you call `update`, when you call `optimize`, or when you call `write` to write the model to disk. When you query information about the model, the result will depend on both *whether* that information was modified and *when* it was modified. In particular, no matter what setting of `UpdateMode` you use, if the modification is sitting in the queue, you'll get the result from before the modification.

To expand on this a bit, all attribute modifications are actually placed in a queue. This includes attributes that may not traditionally be viewed as being part of the model, including things like variable branching priorities, constraint basis statuses, etc. Querying the values of these attributes will return their previous values if subsequent modifications are still in the queue.

The only potential benefit to changing the parameter to 0 is that in unusual cases this setting may allow simplex to make more aggressive use of warm-start information after a model modification.

If you want to change this parameter, you need to set it as soon as you create your Gurobi environment.

Note that you still need to call `update` to modify an attribute on an SOS constraint, quadratic constraint, or general constraint.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

UserName

User name for Remote Services

Type: string
Default value: ""

Identify the user connecting to the Remote Services Manager.

You can provide either a username and password, or an access ID and a secret key, to authenticate your connection to a Cluster Manager.

You can set this parameter through either a `gurobi.lic` file (using `USERNAME=YOUR_USERNAME`) or an empty environment. Changing the parameter after your environment has been started will result in an error.

Note: Cluster Manager only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

VarBranch

Branch variable selection strategy

Type: int
Default value: -1
Minimum value: -1
Maximum value: 3

Controls the branch variable selection strategy. The default -1 setting makes an automatic choice, depending on problem characteristics. Available alternatives are Pseudo Reduced Cost Branching (0), Pseudo Shadow Price Branching (1), Maximum Infeasibility Branching (2), and Strong Branching (3).

Changing the value of this parameter rarely produces a significant benefit.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WLSAccessID

Web License Service access ID

Type: string
Default value: ""

When using a WLS license, set this parameter to the access ID for your license. You can retrieve this string from your account on the Gurobi Web License Manager site.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WLSSecret

Web License Service secret

Type: string
Default value: ""

When using a WLS license, set this parameter to the secret key for your license. You can retrieve this string from your account on the Gurobi Web License Manager site.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WLSToken

Web License Service token

Type: string
Default value: ""

If you are using a WLS license and have retrieved your own token through the WLS REST API, use this parameter to pass that token to the library. If you do this, you don't need to set any other WLS-related parameters.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WLSTokenDuration

Web License Service token duration

Type: int
Default value: 0
Minimum value: 0
Maximum value: MAXINT

When using a WLS license, this parameter can be used to adjust the lifespan (in minutes) of a token. A token enables Gurobi to run on that client for the life of the token. Gurobi will automatically request a new token if the current one expires, but it won't notify the WLS server if it completes its work before the current token expires. A shorter lifespan is better for short-lived usage. A longer lifespan is better for environments where the network connection to the WLS server is unreliable.

The default value of 0 means 'automatic', and is currently equal to 5 minutes. This value may change in the future.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WLSTokenRefresh

Web License Service token refresh interval

Type: double
Default value: 0.9
Minimum value: 0.0
Maximum value: 1.0

The value specifies the fraction of the token duration after which a token refresh is triggered. So, for example, if the token duration is 30 minutes and WLSTokenRefresh is set to 0.6, the token will be refreshed every 18 minutes. The minimum refresh interval is 4 minutes.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WorkerPassword

Distributed worker password

Type: string
Default value: ""

When using a distributed algorithm (distributed MIP, distributed concurrent, or distributed tuning), this parameter allows you to specify the password for the distributed worker cluster provided in the WorkerPool parameter.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WorkerPool

Distributed worker cluster (for distributed algorithms)

Type: string
Default value: ""

When using a distributed algorithm (distributed MIP, distributed concurrent, or distributed tuning), this parameter allows you to specify a Remote Services cluster that will provide distributed workers. You should also specify the access password for that cluster, if there is one, in the WorkerPassword parameter. Note that you don't need to set either of these parameters if your job is running on a Compute Server node and you want to use the same cluster for the distributed workers.

You can provide a comma-separated list of machines for added robustness. If the first node in the list is unavailable, the client will attempt to contact the second node, etc.

To give an example, if you have a Remote Services cluster that uses port 61000 on a pair of machines named `server1` and `server2`, you could set `WorkerPool` to `"server1:61000"` or `"server1:61000,server2:61000"`.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WorkLimit

Work limit

Type: double
Default value: Infinity
Minimum value: 0
Maximum value: Infinity

Limits the total work expended (in work units). Optimization returns with a `WORK_LIMIT` status if the limit is exceeded (see the Status Code section for further details).

In contrast to the TimeLimit, work limits are deterministic. This means that on the same hardware and with the same parameter and attribute settings, a work limit will stop the optimization of a given model at the exact same point every time. One work unit corresponds very roughly to one

second on a single thread, but this greatly depends on the hardware on which Gurobi is running and the model that is being solved.

Note that optimization may not stop immediately upon hitting the work limit. It will stop when the optimization is next in a deterministic state, and it will then perform the required additional computations of the attributes associated with the terminated optimization. As a result, the Work attribute may be larger than the specified WorkLimit upon completion, and repeating the optimization with a WorkLimit set to the Work attribute of the stopped optimization may result in additional computations and a larger attribute value.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ZeroHalfCuts

Zero-half cut generation

Type: int
Default value: -1
Minimum value: -1
Maximum value: 2

Controls zero-half cut generation. Use 0 to disable these cuts, 1 for moderate cut generation, or 2 for aggressive cut generation. The default -1 value chooses automatically. Overrides the Cuts parameter.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

ZeroObjNodes

Zero-objective heuristic

Type: int
Default value: -1
Minimum value: -1
Maximum value: MAXINT

Number of nodes to explore in the zero objective heuristic. Note that this heuristic is only applied at the end of the MIP root, and only when no other root heuristic finds a feasible solution.

This heuristic is quite expensive, and generally produces poor quality solutions. You should generally only use it if other means, including exploration of the tree with default settings, fail to produce a feasible solution.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

Note: Only affects mixed integer programming (MIP) models

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

13.3 Parameter Examples

Gurobi parameter handling is designed to be orthogonal, meaning that you only need to use a small number of routines to work with a large number of parameters. In particular:

- The names and meanings of the various Gurobi parameters remain constant across the different programming language APIs, although some decoration is required in each language.
- Given the type of a parameter (double, integer, etc.) and the programming language you wish to use it from, you simply need to identify the appropriate routine for that parameter type in that language in order to query or modify that parameter.

Please refer to the following sections for detailed examples of how to work with parameters from our various APIs:

- C
- C++
- C#
- Java
- MATLAB
- Python
- R
- Visual Basic

You can also browse our Examples to get a better sense of how to use our parameter interface.

One important note about integer-valued parameters: while the maximum value that can be stored in a signed integer is $2^{31} - 1$, we use a `MAXINT` value of 2,000,000,000. Attempting to set an integer parameter to a value larger than this maximum will produce an error.

C Parameter Examples

The C interface defines a symbolic constant for each parameter. The symbolic constant name is prefixed by `GRB_type_PAR_`, where `type` is either `INT`, `DBL`, or `STR`. This is followed by the capitalized parameter name. For example, the symbolic constant for the integer Threads parameter (found in C header file `gurobi_c.h`) is:

```
#define GRB_INT_PAR_THREADS "Threads"
```

The routine you use to modify a parameter value depends on the type of the parameter. For a double-valued parameter, you would use GRBsetdblparam.

Recall that each model gets its own copy of the environment when it is created. Parameter changes to the original environment therefore have no effect on existing models. You'll need to use GRBgetenv to retrieve the environment associated with a particular model if you want to change a parameter for that model.

To set the TimeLimit parameter for a model, you'd do:

```
error = GRBsetdblparam(GRBgetenv(model), GRB_DBL_PAR_TIMELIMIT, 100.0);
```

If you'd prefer to use a string for the parameter name, you can also do:

```
error = GRBsetdblparam(GRBgetenv(model), "TimeLimit", 100.0);
```

The case of the string is ignored, as are underscores. Thus, `TimeLimit` and `TIME_LIMIT` are equivalent.

Use GRBgetdblparam to query the current value of a (double) parameter:

```
double currentvalue;
error = GRBgetdblparam(modelenv, "TimeLimit", &currentvalue);
```

C++ Parameter Examples

In the C++ interface, parameters are grouped by datatype into three `enums`: `GRB_DoubleParam`, `GRB_IntParam`, and `GRB_StringParam`. You refer to a specific parameter by appending the parameter name to the enum name. For example, the Threads parameter is `GRB_IntParam_Threads`.

To modify a parameter, you use GRBModel::set. Recall that you can also set parameters on an environment, but changes to the environment won't affect models that have already been created using that environment. It is generally simpler to set parameters on the model itself.

To set the TimeLimit parameter for a model, you'd do:

```
GRBModel *m = ...;
m->set(GRB_DoubleParam_TimeLimit, 100.0);
```

You can also set the value of a parameter using strings for the parameter name and desired value. For example:

```
GRBModel *m = ...;
m->set("TimeLimit", "100.0");
```

Use GRBModel::get to query the current value of a parameter:

```
currentlimit = m.get(GRB_DoubleParam_TimeLimit);
```

C# Parameter Examples

In the C# interface, parameters are grouped by datatype into three `enums`: `GRB.DoubleParam`, `GRB.IntParam`, and `GRB.StringParam`. You would refer to the integer Threads parameter as `GRB.IntParam.Threads`.

To modify a parameter, set the corresponding .NET property from `Model.Parameters`. To set the TimeLimit parameter, for example:

```
GRBModel m = ...;
m.Parameters.TimeLimit = 100.0;
```

You can also use GRBModel.Set:

```
m.Set(GRB.DoubleParam.TimeLimit, 100.0);
```

You can also set the value of a parameter using strings for the parameter name and desired value. For example:

```
GRBModel m = ...;
m.Set("TimeLimit", "100.0");
```

To query the current value of a parameter, use:

```
currentlimit = m.Parameters.TimeLimit;
```

You can also use GRBModel.Get:

```
currentlimit = m.Get(GRB.DoubleParam.TimeLimit);
```

Java Parameter Examples

In the Java interface, parameters are grouped by datatype into three `enums`: `GRB.DoubleParam`, `GRB.IntParam`, and `GRB.StringParam`. You would refer to the integer Threads parameter as `GRB.IntParam.Threads`.

To modify a parameter, you use GRBModel.set. Recall that you can also set parameters on an environment, but changes to the environment won't affect models that have already been created using that environment. It is generally simpler to set parameters on the model itself.

To set the TimeLimit parameter for a model, you'd do:

```
GRBModel m = ...;
m.set(GRB.DoubleParam.TimeLimit, 100.0);
```

You can also set the value of a parameter using strings for the parameter name and desired value. For example:

```
GRBModel m = ...;
m.set("TimeLimit", "100.0");
```

Use GRBModel.get to query the current value of a parameter:

```
currentlimit = m.get(GRB.DoubleParam.TimeLimit);
```

MATLAB Parameter Examples

In the MATLAB interface, parameters are passed to Gurobi through a `struct`. To modify a parameter, you create a field in the `struct` with the appropriate name, and set it to the desired value. For example, to set the TimeLimit parameter to 100 you'd do:

```
params.timelimit = 100;
```

The case of the parameter name is ignored, as are underscores. Thus, you could also do:

```
params.timeLimit = 100;
```

...or...

```
params.TIME_LIMIT = 100;
```

All desired parameter changes should be stored in a single `struct`, which is passed as the second parameter to the gurobi function.

Python Parameter Examples

In the Python interface, parameters are listed as constants within the `GRB.Param` class. You would refer to the Threads parameter as `GRB.Param.Threads`.

To modify a parameter, you can set the appropriate member of `Model.Params`. To set the time limit for model `m`, you'd do:

```
m.Params.timeLimit = 100.0
```

The case of the parameter name is actually ignored, as are underscores, so you could also do:

```
m.Params.timelimit = 100.0
```

...or...

```
m.Params.TIME_LIMIT = 100.0
```

You can also use the Model.setParam method:

```
m.setParam(GRB.Param.TimeLimit, 100.0)
```

If you'd prefer to use a string for the parameter name, you can also do:

```
m.setParam("TimeLimit", 100.0);
```

To query the current value of a parameter, use:

```
currentlimit = m.Params.timeLimit
```

R Parameter Examples

In the R interface, parameters are passed to Gurobi through a `list`. To modify a parameter, you create a `named component` in the `list` with the appropriate name, and set it to the desired value. For example, to set the TimeLimit parameter to 100 you'd do:

```
params <- list(TimeLimit=100)
```

The case of the parameter name is ignored, as are underscores. Thus, you could also do:

```
params <- list(timeLimit = 100)
```

...or...

```
params <- list(TIME_LIMIT = 100)
```

All desired parameter changes should be stored in a single `list`, which is passed as the second parameter to the gurobi function.

Visual Basic Parameter Examples

In the Visual Basic interface, parameters are grouped by datatype into three `enums`: `GRB.DoubleParam`, `GRB.IntParam`, and `GRB.StringParam`. You would refer to the integer Threads parameter as `GRB.IntParam.Threads`.

To modify a parameter, set the corresponding .NET property from `Model.Parameters`. To set the TimeLimit parameter, for example:

```
GRBModel m = ...
m.Parameters.TimeLimit = 100.0
```

You can also use GRBModel.Set:

```
m.Set(GRB.DoubleParam.TimeLimit, 100.0)
```

You can also set the value of a parameter using strings for the parameter name and desired value. For example:

```
GRBModel m = ...
m.Set("TimeLimit", "100.0")
```

To query the current value of a parameter, use:

```
currentlimit = m.Parameters.TimeLimit
```

You can also use GRBModel.Get:

```
currentlimit = m.Get(GRB.DoubleParam.TimeLimit)
```

Optimization Status Codes

Once an optimize call has returned, the Gurobi Optimizer sets the Status attribute of the model to one of several possible values. The attribute takes an integer value, but we recommend that you use one of the predefined status constants to check the status in your program. Each code has a name, and each language requires a prefix on this name to obtain the appropriate constant. You would access status code `OPTIMAL` in the following ways from the available Gurobi interfaces:

Language	Status Code
C	`GRB_OPTIMAL`
C++	`GRB_OPTIMAL`
Java	`GRB.Status.OPTIMAL`
.NET	`GRB.Status.OPTIMAL`
Python	`GRB.OPTIMAL`

In MATLAB and R, status codes are returned as strings (e.g. `'OPTIMAL'`).

Possible status codes are as follows. Note that for statuses involving limits or other early termination of the optimization, if the feasibility of the reported solution is not evident from the status, you can assess the feasibility by accessing the appropriate solution quality attribute.

Status code	Value	Description
`LOADED`	1	Model is loaded, but no solution information is available.
`OPTIMAL`	2	Model was solved to optimality (subject to tolerances), and an optimal solution is available.
`INFEASIBLE`	3	Model was proven to be infeasible.
`INF_OR_UNBD`	4	Model was proven to be either infeasible or unbounded. To obtain a more definitive conclusion, set the DualReductions parameter to 0 and reoptimize.
`UNBOUNDED`	5	Model was proven to be unbounded. *Important note: an unbounded status indicates the presence of an unbounded ray that allows the objective to improve without limit. It says nothing about whether the model has a feasible solution. If you require information on feasibility, you should set the objective to zero and reoptimize.*
`CUTOFF`	6	Optimal objective for model was proven to be worse than the value specified in the `Cutoff` parameter. No solution information is available.

ITERATION_LIMIT	7	Optimization terminated because the total number of simplex iterations performed exceeded the value specified in the IterationLimit parameter, or because the total number of barrier iterations exceeded the value specified in the BarIterLimit parameter.
NODE_LIMIT	8	Optimization terminated because the total number of branch-and-cut nodes explored exceeded the value specified in the NodeLimit parameter.
TIME_LIMIT	9	Optimization terminated because the time expended exceeded the value specified in the TimeLimit parameter.
SOLUTION_LIMIT	10	Optimization terminated because the number of solutions found reached the value specified in the SolutionLimit parameter.
INTERRUPTED	11	Optimization was terminated by the user.
NUMERIC	12	Optimization was terminated due to unrecoverable numerical difficulties.
SUBOPTIMAL	13	Unable to satisfy optimality tolerances; a sub-optimal solution is available.
INPROGRESS	14	An asynchronous optimization call was made, but the associated optimization run is not yet complete.
USER_OBJ_LIMIT	15	User specified an objective limit (a bound on either the best objective or the best bound), and that limit has been reached.
WORK_LIMIT	16	Optimization terminated because the work expended exceeded the value specified in the WorkLimit parameter.
MEM_LIMIT	17	Optimization terminated because the total amount of allocated memory exceeded the value specified in the SoftMemLimit parameter.

Batch Status Codes

All batches have a current status, which is queried through the BatchStatus attribute on a Batch object. The attribute takes an integer value, but we recommend that you use one of the predefined status constants to check the status in your program. Each code has a name, and each language requires a prefix on this name to obtain the appropriate constant. You would access status code `COMPLETED` in the following ways from the available Gurobi interfaces:

Language	Status Code
C	`GRB_BATCH_COMPLETED`
C++	`GRB_BATCH_COMPLETED`
Java	`GRB.BatchStatus.COMPLETED`
.NET	`GRB.BatchStatus.COMPLETED`
Python	`GRB.BATCH_COMPLETED`

In MATLAB and R, batch status codes are returned as strings (e.g. `'BATCH_COMPLETED'`).

Possible batch status codes are as follows:

Status code	Value	Description
`CREATED`	1	Batch was created, but is not ready to be scheduled.
`SUBMITTED`	2	The batch has been completely specified, and is waiting for a job to finish processing the request.
`ABORTED`	3	Batch processing was aborted by the user.
`FAILED`	4	Batch processing failed.
`COMPLETED`	5	A Batch Job successfully processed the Batch request.

See the Batch Optimization section for more details.

Callback Codes

The Gurobi callback routines make use of a pair of arguments: `where` and `what`. When a user callback function is called, the `where` argument indicates from where in the Gurobi optimizer it is being called (presolve, simplex, barrier, MIP, etc.). When the user callback wishes to obtain more detailed information about the state of the optimization, the `what` argument can be passed to an appropriate `get` method for your language to obtain additional information (e.g., GRBcbget in C, GRBCallback::getIntInfo in C++, GRBCallback.getIntInfo in Java, GRBCallback.GetIntInfo in .NET, and Model.cbGet in Python).

More detailed information on how to use callbacks in your application can be found in the reference manuals for the different Gurobi language interfaces (C, C++, Java, .NET, and Python).

Note that changing parameters from within a callback is not supported, doing so may lead to undefined behavior.

Possible values for the `where` and `what` arguments are listed in the following tables. Note that these values are referred to in slightly different ways from the different Gurobi interfaces. Consider the `SIMPLEX` value as an example. You would refer to this constant as follows from the different Gurobi APIs:

Language	Callback constant
C	`GRB_CB_SIMPLEX`
C++	`GRB_CB_SIMPLEX`
Java	`GRB.Callback.SIMPLEX`
.NET	`GRB.Callback.SIMPLEX`
Python	`GRB.Callback.SIMPLEX`

Possible `where` values are:

where	Numeric value	Optimizer status
`POLLING`	0	Periodic polling callback
`PRESOLVE`	1	Currently performing presolve
`SIMPLEX`	2	Currently in simplex
`MIP`	3	Currently in MIP
`MIPSOL`	4	Found a new MIP incumbent
`MIPNODE`	5	Currently exploring a MIP node
`MESSAGE`	6	Printing a log message
`BARRIER`	7	Currently in barrier
`MULTIOBJ`	8	At the end of a multi-objective iteration
`IIS`	9	Currently computing an IIS

Allowable `what` values depend on the value of the `where` argument. Valid combinations are:

what	where	Result type	Description

RUNTIME	Any except POLLING	double	Elapsed solver runtime (seconds).
WORK	Any except POLLING	double	Elapsed solver work (work units).
PRE_COLDEL	PRESOLVE	int	The number of columns removed by presolve to this point.
PRE_ROWDEL	PRESOLVE	int	The number of rows removed by presolve to this point.
PRE_SENCHG	PRESOLVE	int	The number of constraint senses changed by presolve to this point.
PRE_BNDCHG	PRESOLVE	int	The number of variable bounds changed by presolve to this point.
PRE_COECHG	PRESOLVE	int	The number of coefficients changed by presolve to this point.
SPX_ITRCNT	SIMPLEX	double	Current simplex iteration count.
SPX_OBJVAL	SIMPLEX	double	Current simplex objective value.
SPX_PRIMINF	SIMPLEX	double	Current primal infeasibility.
SPX_DUALINF	SIMPLEX	double	Current dual infeasibility.
SPX_ISPERT	SIMPLEX	int	Is problem currently perturbed?
MIP_OBJBST	MIP	double	Current best objective.
MIP_OBJBND	MIP	double	Current best objective bound.
MIP_NODCNT	MIP	double	Current explored node count.
MIP_SOLCNT	MIP	int	Current count of feasible solutions found.
MIP_CUTCNT	MIP	int	Current count of cutting planes applied.
MIP_NODLFT	MIP	double	Current unexplored node count.
MIP_ITRCNT	MIP	double	Current simplex iteration count.
MIP_OPENSCENARIOS	MIP	int	Number of scenarios that are still open in a multi-scenario model.
MIP_PHASE	MIP	int	Current phase in the MIP solution process. Possible values are 0 (in the NoRel heuristic), 1 (in the standard MIP search), or 2 (performing MIP solution improvement). Predefined constants are available (e.g., GRB.PHASE_MIP_SEARCH).
MIPSOL_SOL	MIPSOL	double *	Solution vector for new solution (C only). The resultP argument to C routine GRBcbget should point to an array of doubles that is at least as long as the number of variables in the user model. Use the `getSolution` callback method in the object-oriented interfaces.

MIPSOL_OBJ	MIPSOL	double	Objective value for new solution.
MIPSOL_OBJBST	MIPSOL	double	Current best objective.
MIPSOL_OBJBND	MIPSOL	double	Current best objective bound.
MIPSOL_NODCNT	MIPSOL	double	Current explored node count.
MIPSOL_SOLCNT	MIPSOL	int	Current count of feasible solutions found.
MIPSOL_OPENSCENARIOS	MIPSOL	int	Number of scenarios that are still open in a multi-scenario model.
MIPSOL_PHASE	MIPSOL	int	Current phase in the MIP solution. Possible values are 0 (in the NoRel heuristic), 1 (in the standard MIP search), or 2 (performing MIP solution improvement). Predefined constants are available (e.g., GRB.PHASE_MIP_SEARCH).
MIPNODE_STATUS	MIPNODE	int	Optimization status of current MIP node (see the Status Code section for further information).
MIPNODE_OBJBST	MIPNODE	double	Current best objective.
MIPNODE_OBJBND	MIPNODE	double	Current best objective bound.
MIPNODE_NODCNT	MIPNODE	double	Current explored node count.
MIPNODE_SOLCNT	MIPNODE	int	Current count of feasible solutions found.
MIPNODE_OPENSCENARIOS	MIPNODE	int	Number of scenarios that are still open in a multi-scenario model.
MIPNODE_PHASE	MIPNODE	int	Current phase in the MIP solution process. Possible values are 0 (in the NoRel heuristic), 1 (in the standard MIP search), or 2 (performing MIP solution improvement). Predefined constants are available (e.g., GRB.PHASE_MIP_SEARCH).
MIPNODE_REL	MIPNODE	double *	Relaxation solution for the current node, when its optimization status is GRB_OPTIMAL (C only). The resultP argument to C routine GRBcbget should point to an array of doubles that is at least as long as the number of variables in the user model. Use the `getNodeRel` callback method in the object-oriented interfaces.
BARRIER_ITRCNT	BARRIER	int	Current barrier iteration count.
BARRIER_PRIMOBJ	BARRIER	double	Primal objective value for current barrier iterate.

BARRIER_DUALOBJ	BARRIER	double	Dual objective value for current barrier iterate.
BARRIER_PRIMINF	BARRIER	double	Primal infeasibility for current barrier iterate.
BARRIER_DUALINF	BARRIER	double	Dual infeasibility for current barrier iterate.
BARRIER_COMPL	BARRIER	double	Complementarity violation for current barrier iterate.
MULTIOBJ_OBJCNT	MULTIOBJ	int	Current count of objectives already optimized.
MULTIOBJ_SOLCNT	MULTIOBJ	int	Current count of feasible solutions found.
MULTIOBJ_SOL	MULTIOBJ	double *	Solution vector for new solution (C only). The resultP argument to C routine
IIS_CONSTRMIN	IIS	int	Minimum number of constraints in the IIS.
IIS_CONSTRMAX	IIS	int	Maximum number of constraints in the IIS.
IIS_CONSTRGUESS	IIS	int	Estimated number of constraints in the IIS.
IIS_BOUNDMIN	IIS	int	Minimum number of variable bounds in the IIS.
IIS_BOUNDMAX	IIS	int	Maximum number of variable bounds in the IIS.
IIS_BOUNDGUESS	IIS	int	Estimated number of variable bounds in the IIS.
MSG_STRING	MESSAGE	char *	The message that is being printed.

Remember that the appropriate prefix must be added to the `what` or `where` name listed above, depending on the language you are using.

Callback notes

Note that the `POLLING` callback is an optional callback that is only invoked if other callbacks have not been called in a while. It does not allow any progress information to be retrieved. It is simply provided to allow interactive applications to regain control frequently, so that they can maintain application responsiveness.

The object-oriented interfaces have specialized methods for obtaining the incumbent or relaxation solution. While in C you would use GRBcbget, you use `getSolution` or `getNodeRel` in the object-oriented interfaces. Please consult the callback descriptions for C++, Java, .NET, or Python for further details.

Note that the `MIPSOL` callback might also be called for solutions that do not improve the incumbent. This has technical reasons and should only happen in rare cases. You might want to check its objective value if your code relies on stricly improving solutions.

Note that the `MIPNODE` callback will be called once for each cut pass during the root node solve.

The `MIPNODE_NODCNT` value will remain at 0 until the root node is complete. If you query relaxation values from during the root node, the first `MIPNODE` callback will give the relaxation with no cutting planes, and the last will give the relaxation after all root cuts have been applied.

Note that the multi-objective optimization algorithm solves a sequence of optimization problems. In each solve the `MULTIOBJ` callback will be called. Also, MIP-related callbacks will be called if the original model is a MIP, and LP-related callbacks will be called if the original model is an LP.

Note that there are certain restrictions concerning the available callbacks when using the Gurobi Remote Services (Compute Server, Instant Cloud, etc.). Please refer to the `Callbacks` section of the Gurobi Remote Services Manual for more information.

Errors can arise in most of the Gurobi library routines. In the C interface, library routines return an integer error code. In the C++, Java, .NET, and Python interfaces, Gurobi methods can throw an exception (a C++ exception, a Java exception, a .NET exception, or a Python exception).

Underlying all Gurobi error reporting is a set of error codes. These are integer values, but we recommend that you use one of the predefined error code constants to check the error status in your program. Each error code has a name, and each language requires a prefix on this name to obtain the appropriate constant. You would access error code `OUT_OF_MEMORY` in the following ways from the available Gurobi interfaces:

Language	Error Code
C	`GRB_ERROR_OUT_OF_MEMORY`
C++	`GRB_ERROR_OUT_OF_MEMORY`
Java	`GRB.Error.OUT_OF_MEMORY`
.NET	`GRB.Error.OUT_OF_MEMORY`
Python	`GRB.Error.OUT_OF_MEMORY`

Note that when an error occurs, it produces both an error code and an error message. The message can be obtained through GRBgeterrormessage in C, through GRBException::getMessage() in C++, through the inherited `getMessage()` method on the GRBException class in Java, through the inherited `Message` property on the GRBException class in .NET, or through the `e.message` attribute on the GurobiError object in Python.

In MATLAB and R, the error codes and corresponding messages are printed on the screen whenever an error occurs.

Possible error codes are:

Error code	Error number	Description
`OUT_OF_MEMORY`	10001	Available memory was exhausted
`NULL_ARGUMENT`	10002	`NULL` input value provided for a required argument
`INVALID_ARGUMENT`	10003	An invalid value was provided for a routine argument
`UNKNOWN_ATTRIBUTE`	10004	Tried to query or set an unknown attribute
`DATA_NOT_AVAILABLE`	10005	Attempted to query or set an attribute that could not be accessed at that time
`INDEX_OUT_OF_RANGE`	10006	Tried to query or set an attribute, but one or more of the provided indices (e.g., constraint index, variable index) was outside the range of valid values
`UNKNOWN_PARAMETER`	10007	Tried to query or set an unknown parameter

VALUE_OUT_OF_RANGE	10008	Tried to set a parameter to a value that is outside the parameter's valid range
NO_LICENSE	10009	Failed to obtain a valid license
SIZE_LIMIT_EXCEEDED	10010	Attempted to solve a model that is larger than the limit for a demo license
CALLBACK	10011	Problem in callback
FILE_READ	10012	Failed to read the requested file
FILE_WRITE	10013	Failed to write the requested file
NUMERIC	10014	Numerical error during requested operation
IIS_NOT_INFEASIBLE	10015	Attempted to perform infeasibility analysis on a feasible model
NOT_FOR_MIP	10016	Requested operation not valid for a MIP model
OPTIMIZATION_IN_PROGRESS	10017	Tried to query or modify a model while optimization was in progress
DUPLICATES	10018	Constraint, variable, or SOS contained duplicated indices
NODEFILE	10019	Error in reading or writing a node file during MIP optimization
Q_NOT_PSD	10020	Q matrix in QP model is not positive semi-definite
QCP_EQUALITY_CONSTRAINT	10021	QCP equality constraint specified (only inequalities are supported if the NonConvex parameter is set to 0 or 1)
NETWORK	10022	Problem communicating with the Gurobi Compute Server
JOB_REJECTED	10023	Gurobi Compute Server responded, but was unable to process the job (typically because the queuing time exceeded the user-specified timeout or because the queue has exceeded its maximum capacity)
NOT_SUPPORTED	10024	Indicates that a Gurobi feature is not supported under your usage environment (for example, some advanced features are not supported in a Compute Server environment)
EXCEED_2B_NONZEROS	10025	Indicates that the user has called a query routine on a model with more than 2 billion non-zero entries, and the result would exceed the maximum size that can be returned by that query routine. The solution is typically to move to the GRBX version of that query routine, e.g. GRBXgetvars.
INVALID_PIECEWISE_OBJ	10026	Piecewise-linear objectives must have certain properties (as described in the documentation for the various setPWLObj methods). This error indicates that one of those properties was violated.
UPDATEMODE_CHANGE	10027	The UpdateMode parameter can not be modified once a model has been created.

CLOUD	10028	Problems launching a Gurobi Instant Cloud job.
MODEL_MODIFICATION	10029	Indicates that the user has modified the model in such a way that the model became invalid. For example, this happens when a general constraint exists in the model and the user deletes the resultant variable of this constraint. In such a case, the general constraint does not have any meaningful interpretation anymore. The solution is to also delete the general constraint when a resultant variable is deleted.
CSWORKER	10030	When you are using a client-server feature, this error indicates that there was an application problem.
TUNE_MODEL_TYPES	10031	Indicates that tuning was invoked on a set of models, but the models were of different types (e.g., one an LP, another a MIP).
SECURITY	10032	Indicates that an authentication step failed or that an operation was attempted for which the current credentials do not warrant permission.
NOT_IN_MODEL	20001	Tried to use a constraint or variable that is not in the model, either because it was removed or because it has not yet been added.
FAILED_TO_CREATE_MODEL	20002	Failed to create the requested model
INTERNAL	20003	Internal Gurobi error

The Gurobi Optimizer works with a variety of text file formats:

- The MPS, REW, LP, RLP, ILP, OPB, DLP, and DUA formats are used to hold optimization models.
- The MST format is used to hold MIP start data. Importing this data into a MIP model allows optimization to start with a known feasible solution.
- The HNT format is used to hold MIP hints. Importing this data into a MIP model guides the MIP search towards a guess at a high-quality feasible solution.
- The ORD format is used to hold MIP variable branching priorities. Importing this data into a MIP model affects the search strategy.
- The BAS format holds simplex basis information. Importing this data into a continuous models allows the simplex algorithm to start from the given simplex basis.
- The SOL and JSON solution formats are used to hold solution information. The latter includes additional information about the results of the optimization. Both formats can only be written after the model has been optimized.
- The ATTR format stores a collection of attributes of a model, including (multiple) MIP starts, solution, basis information, partitions, variable hints and branching priorities.
- The PRM format holds parameter values. Importing this data into a model changes the values of the referenced parameters.

Note that all of the Gurobi file I/O routines can work with compressed versions of these files. Specifically, we can read or write files with the following extensions: `.zip`, `.gz`, `.bz2`, and `.7z` (assuming that the associated compression tool, e.g., `7zip` for `.7z`, is installed on your machine and a corresponding entry is part of your PATH environment variable).

18.1 MPS format

MPS format is the oldest and most widely used format for storing math programming models. There are actually two variants of this format in wide use. In fixed format, the various fields must always start at fixed columns in the file. Free format is very similar, but the fields are separated by whitespace characters instead of appearing in specific columns. One important practical difference between the two formats is in name length. In fixed format, row and column names are exactly 8 characters, and spaces are part of the name. In free format, names can be arbitrarily long (although the Gurobi reader places a 255 character limit on name length), and names may not contain spaces. The Gurobi MPS reader reads both MPS types, and recognizes the format automatically.

Note that any line that begins with the `*` character is a comment. The contents of that line are ignored.

NAME section

The first section in an MPS format file is the `NAME` section. It gives the name of the model:

```
NAME          AFIRO
```

In fixed format, the model name starts in column 15.

OBJSENSE section

The `OBJSENSE` section is an optional section for maximizing the objective function. By default, Gurobi assumes the objective function of an MPS file should be minimized, in which case the `OBJSENSE` section can be omitted.

To instruct Gurobi to maximize the objective function, add the line

```
OBJSENSE    MAX
```

after the `NAME` section.

ROWS section

The next section is the `ROWS` section. It begins with the word `ROWS` on its own line, and continues with one line for each row in the model. These lines indicate the constraint type (`E` for equality, `L` for less-than-or-equal, or `G` for greater-than-or-equal), and the constraint name. In fixed format, the type appears in column 2 and the row name starts in column 5. Here's a simple example:

```
ROWS
 E  R09
 E  R10
 L  X05
 N  COST
```

Note that an `N` in the type field indicates that the row is a *free row*. The first free row is used as the objective function.

If the file includes multiple `N` rows, each including a priority, weight, relative, and absolute tolerance field, then each such row is treated as an objective in a multi-objective model. The additional fields must appear after the name, separated by spaces. For example, the following would capture a pair of objectives, where the first has priority 2 and the second has priority 1 (and both have identical weights, and relative and absolute tolerances):

```
 N  OBJ0 2 1 0 0
 N  OBJ1 1 1 0 0
```

Please refer to the multi-objective, ObjNPriority, ObjNWeight, ObjNAbsTol, and ObjNRelTol sections for information on the meanings of these fields. Note that all objectives of a multi-objective optimization problem have to be linear.

LAZYCONS section

The next section is the `LAZY CONSTRAINTS` section. It begins with the line `LAZYCONS`, optionally followed by a space and a laziness level 1-3 (if no laziness level is specified 1 is assumed), and continues with one line for each lazy constraint. The format is the same as that of the `ROWS` section: each line indicates the constraint type (`E` for equality, `L` for less-than-or-equal, or `G` for greater-than-or-equal), and the constraint name. In fixed format, the type appears in column 2 and the row name starts in column 5. For example:

```
LAZYCONS
 E  R01
 G  R07
 L  S01
LAZYCONS 2
 E  R02
 G  R03
 L  S11
```

Lazy constraints are linear constraints, and they are semantically equivalent to standard linear constraints (i.e., entries in the `ROWS` section). Depending on their laziness level they are enforced differently by the MIP solver. Please refer to the description of the Lazy attribute for details.

This section is optional.

USERCUTS section

The next section is the `USER CUTS` section. It begins with the line `USERCUTS`, on its own line, and continues with one line for each user cut. The format is the same as that of the `ROWS` section: each line indicates the constraint type (`E` for equality, `L` for less-than-or-equal, or `G` for greater-than-or-equal), and the constraint name. In fixed format, the type appears in column 2 and the row name starts in column 5. For example:

```
USERCUTS
 E  R01
 G  R07
 L  S01
```

User cuts are linear constraints, and they are semantically equivalent to standard linear constraints (i.e., entries in the `ROWS` section). Please refer to the description of the Lazy attribute for details.

This section is optional.

COLUMNS section

The next and typically largest section of an MPS file is the `COLUMNS` section, which lists the columns in the model and the non-zero coefficients associated with each. Each line in the columns section provides a column name, followed by either zero, one, or two non-zero coefficients from that column. Coefficients are specified using a row name first, followed by a floating-point value. Consider the following example:

```
COLUMNS
    X01          X48            .301   R09          -1.
    X01          R10          -1.06    X05           1.
    X02          X21            -1.    R09           1.
    X02          COST           -4.
```

The first line indicates that column `X01` has a non-zero in row `X48` with coefficient `.301`, and a non-zero in row `R09` with coefficient `-1.0`. Note that multiple lines associated with the same column must be contiguous in the file.

In fixed format, the column name starts in column 5, the row name for the first non-zero starts in column 15, and the value for the first non-zero starts in column 25. If a second non-zero is present, the row name starts in column 40 and the value starts in column 50.

Integrality markers

The COLUMNS section can optionally include integrality markers. The variables introduced between a pair of markers must take integer values. All variables within markers will have a default lower bound of 0 and a default upper bound of 1 (other bounds can be specified in the BOUNDS section). The beginning of an integer section is marked by an INTORG marker:

```
    MARK0000  'MARKER'                 'INTORG'
```

The end of the section is marked by an INTEND marker:

```
    MARK0000  'MARKER'                 'INTEND'
```

The first field (beginning in column 5 in fixed format) is the name of the marker (which is ignored). The second field (in column 15 in fixed format) must be equal to the string 'MARKER' (including the single quotes). The third field (in column 40 in fixed format) is 'INTORG' at the start and 'INTEND' at the end of the integer section.

The COLUMNS section can contain an arbitrary number of such marker pairs.

RHS section

The next section of an MPS file is the RHS section, which specifies right-hand side values. Each line in this section may contain one or two right-hand side values.

```
RHS
    B           X50           310.   X51           300.
    B           X05            80.   X17            80.
```

The first line above indicates that row X50 has a right-hand side value of 310, and X51 has a right-hand side value of 300. In fixed format, the variable name for the first bound starts in column 15, and the first bound value starts in column 25. For the second bound, the variable name starts in column 40 and the value starts in column 50. The name of the RHS is specified in the first field (column 5 in fixed format), but this name is ignored by the Gurobi reader. If a row is not mentioned anywhere in the RHS section, that row takes a right-hand side value of 0. You may define an objective offset by setting the negative offset as right-hand side of the objective row. For example, if the linear objective row in the problem is called COST and you want to add an offset of 1000 to your objective function, you can add the following to the RHS section:

```
RHS
    RHS1      COST      -1000
```

BOUNDS section

The next section in an MPS file is the optional BOUNDS section. By default, each variable takes a lower bound of 0 and an infinite upper bound. Each line in this section can modify the lower bound of a variable, the upper bound, or both. Each line indicates a bound type (in column 2 in fixed format), a bound name (ignored), a variable name (in column 15 in fixed format), and a bound value (in columns 25 in fixed format). The different bound types, and the meaning of the associate bound value, are as follows:

Bound type	Meaning
LO	lower bound
UP	upper bound
FX	variable is fixed at the specified value
FR	free variable (no lower or upper bound)
MI	infinite lower bound
PL	infinite upper bound
BV	variable is binary (equal 0 or 1)
LI	lower bound for integer variable
UI	upper bound for integer variable
SC	upper bound for semi-continuous variable
SI	upper bound for semi-integer variable

Consider the following example:

```
BOUNDS
 FR BND          X49
 UP BND          X50             80.
 LO BND          X51             20.
 FX BND          X52             30.
```

In this `BOUNDS` section, variable `X49` gets a lower bound of `-infinity` (infinite upper bound is unchanged), variable `X50` gets a upper bound of `80` (lower bound is unchanged at `0`, `X51` gets a lower bound of `20` (infinite upper bound is unchanged), and `X52` is fixed at `30`.

QUADOBJ section

The next section in an MPS file is the optional `QUADOBJ` section, which contains quadratic objective terms. Each line in this section represents a single non-zero value in the lower triangle of the Q matrix. The names of the two variable that participate in the quadratic term are found first (starting in columns 5 and 15 in fixed format), followed by the numerical value of the coefficient (in column 25 in fixed format). By convention, the Q matrix has an implicit one-half multiplier associated with it. Here's an example containing three quadratic terms:

```
QUADOBJ
    X01        X01        10.0
    X01        X02        2.0
    X02        X02        2.0
```

These three terms would represent the quadratic function $(10X01^2 + 2X01 * X02 + 2X02 * X01 + 2X02^2)/2$ (recall that the single off-diagonal term actually represents a pair of non-zero values in the symmetric Q matrix).

QCMATRIX section

The next section in an MPS file contains zero or more `QCMATRIX` blocks. These blocks contain the quadratic terms associated with the quadratic constraints. There should be one block for each quadratic constraint in the model.

Each `QCMATRIX` block starts with a line that indicates the name of the associated quadratic constraint (starting in column 12 in fixed format). This is followed by one of more quadratic terms.

Each term is described on one line, which gives the names of the two involved variables (starting in columns 5 and 15 in fixed format), followed by the coefficient (in column 25 in fixed format). For example:

```
QCMATRIX   QC0
    X01        X01        10.0
    X01        X02        2.0
    X02        X01        2.0
    X02        X02        2.0
```

These four lines describe three quadratic terms: quadratic constraint `QC0` contains terms $10X01^2$, $4X01 * X02$, and $2X02^2$. Note that a `QCMATRIX` block must contain a symmetric matrix, so for example an `X01*X02` term must be accompanied by a matching `X02*X01` term.

Linear terms for quadratic constraint `QC0` appear in the `COLUMNS` section. The sense and right-hand side value appear in the `ROWS` and `RHS` sections, respectively.

PWLOBJ section

The next section in an MPS file is the optional `PWLOBJ` section, which contains piecewise-linear objective functions. Each line in this section represents a single point in a piecewise-linear objective function. The name of the associated variable appears first (starting in column 4), followed by the x and y coordinates of the point (starting in columns 14 and 17). Here's an example containing two piecewise-linear expressions, for variables `X01` and `X02`, each with three points:

```
   X01        1  1
   X01        2  2
   X01        3  4
   X02        1  1
   X02        3  5
   X02        7  10
```

SOS section

The next section in an MPS file is the optional `SOS` section. The representation for a single SOS constraint contains one line that provides the type of the SOS set (`S1` for SOS type 1 or `S2` for SOS type 2, found in column 2 in fixed format) and the name of the SOS set (column 5 in fixed format) of the SOS set. This is followed by one line for each SOS member. The member line gives the name of the member (column 5 in fixed format) and the associated weight (column 15 in fixed format). Here's an example containing two SOS2 sets.

```
SOS
 S2 sos1
    x1            1
    x2            2
    x3            3
 S2 sos2
    x3            1
    x4            2
    x5            3
```

Indicator Constraint section

The indicator constraint section is optional in the MPS format. It starts with the keyword `INDICATORS`. Each subsequent line of the indicator section starts with the keyword `IF` (placed at column 2 in fixed format) followed by a space and a row name (the row must have already been defined in the `ROWS` section). The line continues with a binary variable (placed at column 15 in fixed format) and finally a value `0` or `1` (placed at column 25 in fixed format).

Here a simple example:

```
INDICATORS
 IF row1      x1        0
 IF row2      y1        1
```

The first indicator constraint in this example states that `row1` has to be fulfilled if the variable `x1` takes a value of `0`.

General Constraint section

An MPS file may contain an optional section that captures *general constraints*. This section starts with the keyword `GENCONS`.

General constraints can be of two basic types: *simple* general constraints - *MIN*, *MAX*, *OR*, *AND*, *NORM*, *ABS* or *PWL*, or *function* constraints - polynomial (*POLY*), power (*POW*), exponential (*EXP* or *EXPA*), logarithmic (*LOG*, *LOGA*), or logistic (*LOGISTIC*), and trigonometric functions (*SIN*, *COS*, or *TAN*).

Each general constraint starts with a general constraint type specifier (*MIN*, *MAX*, *OR*, *AND*, *NORM*, *ABS*, *PWL*, *POLY*, *POW*, *EXP*, *EXPA*, *LOG*, *LOGA*, *LOGISTIC*, *SIN*, *COS*, or *TAN*), found in column 2 in fixed format. Optionally a space and a constraint name may follow. For a *NORM* constraint, the norm type (0, 1, 2, or INF) follows the type specifier (and is optionally followed by a constraint name).

For function constraints, the next line defines a few attributes used to perform the piecewise-linear approximation. The line starts with the keyword *Options* (found in column 5 in fixed format), followed by two spaces, followed by five values (separated by two spaces) that define the `FuncPieces`, `FuncPieceLength`, `FuncPieceError`, `FuncPieceRatio`, and `FuncNonlinear` attribute values (in that order).

What follows depends on the general constraint type. Simple general constraints start with the name of the so-called resultant variable, placed on it's own line (starting at column 5 in fixed format). For *MIN* or *MAX* constraints, a non empty list of variables or values follows (with each variable name on its own line). For *OR*, *AND*, and *NORM* constraints, a list of variables follows (each on its own line). The variables must be binary for *OR* and *AND* constraints. For *ABS* constraints, only one additional variable follows (on its own line). In fixed format, all of these variables or values begin in column 5.

Piecewise-linear constraints start with the name of the so-called operand variable (starting at column 5 in fixed format), followed by the so-called resultant variable. The next lines contain the piecewise-linear function breakpoints, each represented as pair of x and y values. The x values must be non-decreasing.

Function constraints also start with the name of the operand variable (starting at column 5 in fixed format), followed by two spaces, followed by the name of the resultant variable. This is sufficient to define *EXP LOG*, *LOGISTIC*, *SIN*, *COS*, and *TAN* functions. The *POW*, *EXPA*

and *LOGA* functions require an exponent or base, respectively, which is defined on the next line (starting in column 5 in fixed format). For the polynomial function, the following lines contain a coefficient (at column 5 in fixed format), followed by two spaces, followed by the associated power (natural numbers only). Note that powers must be decreasing.

The other general constraint type, the *INDICATOR* constraint, appears in a separate `Indicator` section, which is described above.

The following shows an example of a general constraint section:

```
GENCONS
 MAX gc0
    r1
    x1
    x2
    x10
    0.7
 MIN gencons1
    r2
    y0
    10
    y1
    r1
 AND and1
    r
    b1
    b2
 OR or1
    r
    b3
    b4
 NORM 2 norm2
    r3
    x1
    y1
    z1
 ABS GC14
    xabs
    x
 PWL GC0
    x[0]  y[0]
    -1  2
    0  1
    0  0
    0  1
    1  2
 POLY GC2
    Options  0  0.01  0.001  -1
```

```
   x  y
   4  7
   2  3
SIN gc1
   Options  0  0.01  1e-05  0.5
   y  z
LOGA gc6
   Options  0  0.01  0.001  -1
   x  y
   10
EXPA gc4
   Options  0  0.01  0.001  -1
   y  z
   3
```

For more information, consult the general constraint discussion.

Scenario section

An MPS file may contain an optional section that captures scenario data. A model can have multiple scenarios, where each defines a set of changes to the original model (which we refer to as the *base model*).

This section starts with the keyword `SCENARIOS`, followed by the number of scenarios. Scenarios are described as a set of changes to the objective function, the right-hand sides of linear constraints, and the bounds of variables. Objective changes are stated first, followed by right-hand side changes, then bound changes. A scenario can be empty (i.e., identical to the base model).

Each scenario starts with the keyword `NAME` (starting at column 2 in fixed format), followed by a scenario name.

Changes to the objective function are defined in the `COLUMNS` subsection (starting at column 2 in fixed format). Each objective change is on its own line; that line contains the variable name (starting at column 5 in fixed format), the objective name (starting at column 15 in fixed format), and the modified objective value (starting at column 25 in fixed format). The format is similar to the columns section above.

Changes to the right-hand sides of linear constraints are defined in the `RHS` subsection (starting at column 2 in fixed format). Each right-hand side change is on its own line; that line contains a right-hand side specifier (starting at column 5 in fixed format), the constraint name (starting at column 15 in fixed format), and the right-hand side value (starting at column 25 in fixed format). The format is similar to the right-hand side section above.

Changes to variable bounds are defined in the `BOUNDS` subsection. Each changed variable bound is on its own line. The format is similar to the bounds section above (with a small difference that the first and second column in fixed format are 5 and 8, respectively).

The following example shows three scenarios in MPS format:

```
SCENARIOS 3
 NAME scenario0
 NAME scenario1
 COLUMNS
   x1        OBJ       0
```

```
    x2        OBJ       1
RHS
    RHS1      c1        2
    RHS1      c2        2
BOUNDS
    FR BND1   x1
    LO BND1   x3        0.5
    UP BND1   x3        1.5
    FX BND1   x2        0
NAME scenario2
BOUNDS
    FX BND1   x3        3
```

For more information, consult the multiple scenario discussion.

ENDATA

The final line in an MPS file must be an `ENDATA` statement.

Additional notes

Note that in the Gurobi Optimizer, MPS models are always written in full precision. That means that if you write a model and then read it back, the data associated with the resulting model will be bit-for-bit identical to the original data.

18.2 REW format

The REW format is identical to the MPS format, except in how objects are named when files are written. When writing an MPS format file, the Gurobi Optimizer refers to constraints and variables using their given names. When writing an REW format file, the Gurobi Optimizer ignores the given names and instead refers to the variables using a set of default names that are based on row and column numbers. The constraint name depends solely on the associated row number: row `i` gets name `ci`. The variable name depends on the type of the variable, the column number of the variable in the constraint matrix, and the number of non-zero coefficients in the associated column. A continuous variable in column 7 with column length 2 would get name `C7(2)`, for example. A binary variable with the same characteristics would get name `B7(2)`.

18.3 DUA format

The DUA format is identical to the MPS format. The only difference is in how they are used. Writing a DUA file will generate and write the dual formulation of a pure LP model.

18.4 LP format

The LP format captures an optimization model in a way that is easier for humans to read than MPS format, and can often be more natural to produce. One limitation of the LP format is that it doesn't preserve several model properties. In particular, LP files do not preserve column order when read, and they typically don't preserve the exact numerical values of the coefficients (although this isn't inherent to the format).

Unlike MPS files, LP files do not rely on fixed field widths. Line breaks and whitespace characters are used to separate objects. Here is a simple example:

```
\ LP format example

Maximize
  x + y + z
Subject To
  c0: x + y = 1
  c1: x + 5 y + 2 z <= 10
  qc0: x + y + [ x ^ 2 - 2 x * y + 3 y ^ 2 ] <= 5
Bounds
  0 <= x <= 5
  z >= 2
Generals
  x y z
End
```

The backslash symbol starts a comment; the remainder of that line is ignored.

Variable names play a major role in LP files. Each variable must have its own unique name. A name should be no longer than 255 characters, and to avoid confusing the LP parser, it can not begin with a number or any of the characters `+, -, *, ^, <, >, =, (, ), [, ], ,, or :`. For similar reasons, a name should not contain any of the characters `+, -, *, ^, or :`. Also, variable names should not be equal (case insensitive) to any of the LP file format keywords, e.g., `st, bounds, min, max, binary, or end`. Names must be preceded and followed by whitespace.

The same rules apply to any other type of names in the LP format, e.g., constraint names or the objective name.

Note that whitespace characters are not optional in the Gurobi LP format. Thus, for example, the text `x+y+z` would be treated as a single variable name, while `x + y + z` would be treated as a three term expression.

LP files are structured as a list of sections, where each section captures a logical piece of the whole optimization model. Sections begin with particular keywords, and must generally come in a fixed order, although a few are allowed to be interchanged.

Objective Section

The first section in an LP file is the objective section. This section begins with one of the following six keywords: *minimize*, *maximize*, *minimum*, *maximum*, *min*, or *max*. Capitalization is ignored. This keyword may appear alone, or it may be immediately followed by *multi-objectives*, which indicates that the model contains multiple objective functions.

Single-Objective Case

Let us consider single-objective models first, where this header is followed by a single linear or quadratic expression that captures the objective function.

The objective optionally begins with a label. A label consists of a name, followed by a colon character, following by a space. A space is allowed between the name and the colon, but not required.

The objective then continues with a list of linear terms, separated by the + or - operators. A term can contain a coefficient and a variable (e.g., `4.5 x`), or just a variable (e.g., `x`). The objective can be spread over many lines, or it may be listed on a single line. Line breaks can come between tokens, but never within tokens.

The objective may optionally continue with a list of quadratic terms. The quadratic portion of the objective expression begins with a `[` symbol and ends with a `]` symbol, followed by `/ 2`. These brackets should enclose one or more quadratic terms. Either squared terms (e.g., `2 x ^ 2`) or product terms (e.g., `3 x * y`) are accepted. Coefficients on the quadratic terms are optional.

For variables with piecewise-linear objective functions, the objective section will include a `__pwl(x)` term, where `x` is the name of the variable. The actual piecewise-linear expressions are pulled from the later `PWLObj` section.

The objective expression must always end with a line break.

An objective section might look like the following:

```
Minimize
  obj: 3.1 x + 4.5 y + 10 z + [ x ^ 2 + 2 x * y + 3 y ^ 2 ] / 2
```

Multi-Objective Case

In the multi-objective case, the header is followed by one or more linear objective functions, where each starts with its own sub-header. The sub-header gives the name of the objective, followed by a number of fields that provide a *Priority*, *Weight*, absolute tolerance (*AbsTol*) and relative tolerance (*RelTol*) for that objective (see ObjNPriority, ObjNWeight, ObjNAbsTol, and ObjNRelTol for details on the meanings of these fields). The fields start with the field name, followed by a `=`, followed by the value. For example:

```
  OBJ0: Priority=2 Weight=1 AbsTol=0 RelTol=0
```

Please refer to the multi-objective section for additional details.

Each sub-header is followed by a linear expression that captures that objective.

A complete multi-objective section might look like the following:

```
Minimize multi-objectives
  OBJ0: Priority=2 Weight=1 AbsTol=0 RelTol=0
    3.1 x + 4.5 y + 10 z
  OBJ1: Priority=1 Weight=1 AbsTol=0 RelTol=0
    10 x + 0.1 y
```

The objective section is optional. The objective is set to 0 when it is not present.

Constraints Section

The next section is the constraints section. It begins with one of the following headers, on its own line: *subject to*, *such that*, *st*, or *s.t.*. Capitalization is ignored.

The constraint section can have an arbitrary number of constraints. Each constraint starts with an optional label (constraint name, followed by a colon, followed by a space), continues with a linear expression, followed by an optional quadratic expression (enclosed in square brackets), and ends with a comparison operator, followed by a numerical value, followed by a line break. Valid comparison operators are `=`, `<=`, `<`, `>=`, or `>`. Note that LP format does not distinguish between strict and non-strict inequalities, so for example `<` and `<=` are equivalent.

Note that the left-hand side of a constraint may not contain a constant term; the constant must appear on the right-hand side.

The following is a simple example of a valid linear constraint:

```
c0: 2.5 x + 2.3 y + 5.3 z <= 8.1
```

The following is a valid quadratic constraint:

```
qc0: 3.1 x + 4.5 y + 10 z + [ x ^ 2 + 2 x * y + 3 y ^ 2 ] <= 10
```

The constraint section may also contain another constraint type: the so-called indicator constraint. Indicator constraints start with an optional label (constraint name, followed by a colon, followed by a space), followed by a binary variable, a space, a `=`, again a space and a value, either `0` or `1`. They continue with a space, followed by `->`, and again a space and finally a linear constraint (without a label).

For example:

```
c0: b1 = 1 -> 2.5 x + 2.3 y + 5.3 z <= 8.1
```

This example constraint requires the given linear constraint to be satisfied if the variable `b1` takes a value of `1`.

Every LP format file must have a constraints section.

Lazy Constraints Section

The next section is the lazy constraints section. It begins with the line `Lazy Constraints`, optionally followed by a space and a laziness level 1-3 (if no laziness level is specified 1 is assumed), and continues with a list of linear constraints in the exact same format as the linear constraints in the constraints section. For example:

```
Lazy Constraints
  c0: 2.5 x + 2.3 y + 5.3 z <= 8.1
Lazy Constraints 2
  c1: 1.5 x + 3.3 y + 4.3 z <= 8.1
```

Lazy constraints are linear constraints, and they are semantically equivalent to standard linear constraints. Depending on their laziness level they are enforced differently by the MIP solver. Please refer to the description of the Lazy attribute for details.

This section is optional.

User Cuts Section

The next section is the user cuts section. It begins with the line `User Cuts`, on its own line, and is followed by a list of linear constraints in the exact same format as the linear constraints in the constraints section. For example:

```
User Cuts
  c0: 2.5 x + 2.3 y + 5.3 z <= 8.1
```

User cuts are linear constraints, and they are semantically equivalent to standard linear constraints. Please refer to the description of the Lazy attribute for details.

This section is optional.

Bounds Section

The next section is the bounds section. It begins with the word `Bounds`, on its own line, and is followed by a list of variable bounds. Each line specifies the lower bound, the upper bound, or both for a single variable. The keywords `inf` or `infinity` can be used in the bounds section to specify infinite bounds. A bound line can also indicate that a variable is `free`, meaning that it is unbounded in either direction.

Here are examples of valid bound lines:

```
Bounds
  0 <= x0 <= 1
  x1 <= 1.2
  x2 >= 3
  x3 free
  x2 >= -Inf
```

It is not necessary to specify bounds for all variables; by default, each variable has a lower bound of 0 and an infinite upper bound. In fact, the entire bounds section is optional.

Variable Type Section

The next section is the variable types section. Variables can be designated as being either binary, general integer, or semi-continuous. In all cases, the designation is applied by first providing the appropriate header (on its own line), and then listing the variables that have the associated type. For example:

```
Binary
  x y z
```

Variable type designations don't need to appear in any particular order (e.g., general integers can either precede or follow binaries). If a variable is included in multiple sections, the last one determines the variable type.

Valid keywords for variable type headers are: *binary*, *binaries*, *bin*, *general*, *generals*, *gen*, *semi-continuous*, *semis*, or *semi*.

The variable types section is optional. By default, variables are assumed to be continuous.

SOS Section

An LP file can contain a section that captures SOS constraints of type 1 or type 2. The SOS section begins with the `SOS` header on its own line (capitalization isn't important). An arbitrary number of SOS constraints can follow. An SOS constraint starts with a name, followed by a colon (unlike linear constraints, the name is not optional here). Next comes the SOS type, which can be either `S1` or `S2`. The type is followed by a pair of colons.

Next come the members of the SOS set, along with their weights. Each member is captured using the variable name, followed by a colon, followed by the associated weight. Spaces can optionally be placed before and after the colon. An SOS constraint must end with a line break.

Here's an example of an SOS section containing two SOS constraints:

```
SOS
  sos1: S1 :: x1 : 1  x2 : 2  x3 : 3
  sos2: S2 :: x4:8.5  x5:10.2  x6:18.3
```

The SOS section is optional.

PWLObj Section

An LP file can contain a section that captures piecewise-linear objective functions. The PWL section begins with the `PWLObj` header on its own line (capitalization isn't important). Each piecewise-linear objective function is associated with a model variable. A PWL function starts with the corresponding variable name, followed immediately by a colon (the name is not optional). Next come the points that define the piecewise-linear function. These points are represented as `(x, y)` pairs, with parenthesis surrounding the two values and a comma separating them. A PWL function must end with a line break.

Here's an example of a PWLObj section containing two simple piecewise-linear functions:

```
PWLObj
  x1: (1, 1) (2, 2) (3, 4)
  x2: (1, 3) (3, 5) (100, 300)
```

The PWLObj section is optional.

General Constraint Section

An LP file may contain an optional section that captures *general constraints*. This section starts with one of the following keywords *general constraints*, *general constraint*, *gencons*, or *g.c.* (capitalization is ignored).

General constraints can be of two basic types: *simple* general constraints - *MIN*, *MAX*, *OR*, *AND*, *NORM*, *ABS*, or *PWL*, or *function* constraints - polynomial (`POLY`), power (`POW`), exponential (`EXP` or `EXPA`), logarithmic (`LOG`, `LOGA`), logistic (`LOGISTIC`), or trigonometric (`SIN`, `COS`, or `TAN`).

A simple general constraint starts with an optional label (constraint name, followed by a colon), followed by a variable name (the so-called *resultant*), then an equals sign `=`. The line continues with a general constraint type specifier (*MIN*, *MAX*, *OR*, *AND*, *NORM*, or *ABS*), then a `(`. All tokens must be separated using spaces. Capitalization is ignored.

What follows depends on the general constraint type. *MIN* or *MAX* constraints expect a non-empty, comma-separated list of variables or values. *OR* and *AND* constraints expect a comma-separated list of binary variables. *NORM* expects a norm type (0, 1, 2, or INF), in parenthesis, followed by a comma-separated list of variables. *ABS* constraints expect only one variable name. Again, all tokens (including commas) must be separated using spaces.

All of these simple general constraints end with a `)` and a line break.

Here are a few examples:

```
gc0: r1 = MAX ( x1 , x2 , x10 , 0.7 )
gencons1: r2 = MIN ( y0 , 10 , y1 , r1 )
and1: r = AND ( b1 , b2 )
or1: r = OR ( b3 , b4 )
norm2: r = NORM ( 2 ) ( x1 , y1, z1 )
GC14: xabs = ABS ( x )
```

Piecewise-linear constraints also start with an optional label (constraint name, followed by a colon). The line continues with a variable name (the so-called *resultant*) and an equal sign `=`. Next comes the keyword *PWL* that indicates that the constraint is of type piecewise-linear. This is followed by a `(`, and then by a variable name (the so-called *operand*) followed by a `)`. The line

continues with a `:` and then the list of piecewise-linear breakpoints in parentheses (e.g., (x0, y0) (x1, y1)) with non-decreasing values on x. Recall that spaces are required between tokens.

Here an example:

```
GC0: y[0] = PWL ( x[0] ) : (-1, 2) (0, 1) (0, 0) (0, 1) (1, 2)
```

There is one other type of simple constraint, the *INDICATOR* constraint. Those appear in the regular constraints section (described above), not in the general constraint section.

Function constraints also start with an optional label (constraint name, followed by a colon). An optional list of attribute assignments follows. These start with a `(`, then a space-separated list of `Name=Value` strings (no spaces before or after the `=`), closed with a `)`. An example is shown below. Default values are used if no attributes are specified.

The line continues with a variable name (the so-called *resultant*) and an equal sign `=`. Next comes a keyword that indicates the type of function being defined (*POLY*, *POW*, *EXP*, *EXPA*, *LOG*, *LOG_A*, *LOGISTIC*, *SIN*, *COS*, or *TAN*). For a *LOG*, use *LOG_A* if it isn't a natural log, where *A* is the base. This is followed by a `(`, and then by the expression that defines the actual function. The line closes with a `)`. Recall that spaces are required between tokens.

Polynomials and powers are described in what is hopefully the natural way, with exponents preceded by the `^` symbol.

The following give examples of a few function constraints:

```
gc1: ( FuncPieceError=1e-05 FuncPieceRatio=0.5 ) z = SIN ( y )
GC2: ( FuncPieceLength=0.001 ) y = POLY ( 5 x ^ 3 + 2 x + 5 )
gc3: z = EXPA ( 3.5 ^ y )
gc4: z = LOG_10 ( y )
logytoz: z = LOG ( y )
```

For more information, consult the general constraint discussion.

Scenario Section

An LP file may contain an optional section that captures scenario data. A model can have multiple scenarios, where each defines a set of changes to the original model (which we refer to as the *base model*).

This section starts with the *Scenario* keyword (capitalization is ignored), followed by a scenario name. Scenarios are described as a set of changes to the objective function, the right-hand sides of linear constraints, and the bounds of variables. Objective changes are stated first, followed by right-hand side changes, then bound changes. A scenario can be empty (i.e., identical to the base model).

Changes to the objective function start with one of the allowed objective keywords (`Minimize`, `Maximize`, etc.; see objective section above for additional information). Note that the keyword needs to match the objective sense of the base model. This is followed by a line for each changed objective coefficient that contains the variable name and its modified value (separated by a space).

Changes to the right-hand sides of linear constraints start with one of the allowed constraint section keywords (`Subject To`, etc.; see the constraints section above for additional information). This is followed by a line for each changed right-hand side value that contains the constraint name followed by a colon, then a space, the constraint sense, a space, and the scenario right-hand side value.

Changes to variable bounds start with the `Bounds` keyword. This is followed by a line for each variable with changed scenario bounds; the format of each such line is the same as in the bounds section above.

The following example shows three scenarios in LP format:

```
Scenario scenario0
Scenario scenario1
Maximize
 x1 0
 x2 1
Subject To
 c1: <= 2
 c2: >= 2
Bounds
 x3 <= 1.5
 x1 free
 0 <= x2 <= 0
 x3 >= 0.5
Scenario scenario2
Bounds
 x3 = 3
```

For more information, consult the multiple scenario discussion.

End statement

The last line in an LP format file should be an `End` statement.

18.5 RLP format

The RLP format is identical to the LP format, except in how objects are named when files are written. When writing an LP format file, the Gurobi Optimizer refers to constraints and variables using their given names. When writing an RLP format file, the Gurobi Optimizer ignores the given names and instead refers to the variables using names that are based on variable or constraint characteristics. The constraint name depends solely on the associated row number: row `i` gets name `ci`. The variable name depends on the type of the variable, the column number of the variable in the constraint matrix, and the number of non-zero coefficients in the associated column. A continuous variable in column 7 with column length 2 would get name `C7(2)`, for example. A binary variable with the same characteristics would get name `B7(2)`.

18.6 DLP format

The DLP file format is identical to the LP format. The only difference is in how they are used. Writing a DLP file will generate and write the dual formulation of a pure LP model.

18.7 ILP format

The ILP file format is identical to the LP format. The only difference is in how they are used. ILP files are specifically used to write computed Irreducible Inconsistent Subsystem (IIS) models.

18.8 OPB format

The OPB file format is used to store pseudo-boolean satisfaction and pseudo-boolean optimization models. These models may only contain binary variables, but these variables may be complemented and multiplied together in constraints and objectives. Pseudo-boolean models in OPB files are translated into a MIP representation by Gurobi. The syntax of the OPB format is described in detail by Roussel and Manquinho. However, the OPB format supported by Gurobi is less restrictive, e.g., fractional coefficients are allowed.

The following is an example of a pseudo-boolean optimization model

$$\begin{array}{ll} \text{minimize} & y - 1.3x(1-z) + (1-z) \\ \text{subject to} & 2y - 3x + 1.7w = 1.7 \\ & -y + x + xz(1-v) \geq 0 \\ & -y \leq 0, \\ & v, w, x, y, z \in \{0, 1\}. \end{array} \tag{1}$$

The corresponding OPB file for this example is given by

```
* This is a dummy pseudo-boolean optimization model
min: y - 1.3 x ~z + ~z;
2 y - 3 x + 1.7 w = 1.7;
-1 y + x + x z ~v >= 0;
-1 y <= 0;
```

Lines starting with `*` are treated as comments and ignored. Non-comment lines must end with a semicolon `;`. Whitespace characters must be used to separate variables. The complement of a variable may be specified with a tilde `~`.

Only minimization models are supported. These models must be specified with the `min:` objective keyword. This keyword must appear before other constraints. Satisfiability models may be defined by omitting the objective.

Constraint senses `>=`, `=`, and `<=` are supported.

18.9 MST format

A MIP start (MST) file is used to specify an initial solution for a mixed integer programming model. The file lists values to assign to the variables in the model. If a MIP start has been imported into a MIP model before optimization begins (using GRBread, for example), the Gurobi optimizer will attempt to build a feasible solution from the specified start values. A good initial solution often speeds the solution of the MIP model, since it provides an early bound on the optimal value, and also since the specified solution can be used to seed the local search heuristics employed by the MIP solver.

A MIP start file consists of variable-value pairs, each on its own line. Any line that begins with the hash sign (#) is a comment line and is ignored. The following is a simple example:

```
# MIP start
x1  1
x2  0
x3  1
```

Importing a MIP start into a model is equivalent to setting the `Start` attribute for each listed variable to the associated value. If the same variable appears more than once in a start file, the last assignment is used. Importing multiple start files is equivalent to reading the concatenation of the imported files.

Note that start files don't need to specify values for all variables. When variable values are left unspecified, the Gurobi solver will try to extend the specified values into a feasible solution for the full model.

It is important to mention that when saving an MST file, Gurobi will not save the values of continuous variables. If you want to save a complete description of the best solution found, we recommend to save it as a solution file (SOL format). This will guarantee that you will save the values for each variable present in your model.

18.10 HNT format

A MIP hint (HNT) file is used to provide hints for the values of the variables in a mixed integer programming model (typically obtained from a solution to a related model). The file lists values for variables in the model, and priorities for those hints. When MIP hints are imported into a MIP model before optimization begins (using GRBread, for example), the MIP search is guided towards the values captured in those hints. Good hints often allow the MIP solver to find high-quality solutions much more quickly.

A MIP hint file consists of variable-value-priority triples, each on its own line. Any line that begins with the hash sign (#) is a comment line and is ignored. The following is a simple example:

```
# MIP hints
x1  1 2
x2  0 1
x3  1 1
```

Importing hints into a model is equivalent to setting the `VarHintVal` and `VarHintPri` attributes for each listed variable to the associated values. If the same variable appears more than once in a hint file, the last assignment is used. Importing multiple hint files is equivalent to reading the concatenation of the imported files.

Note that hint files don't need to specify values for all variables. When values are left unspecified, the Gurobi MIP solver won't attempt to adjust the search strategy for those variables.

Please refer to the VarHintVal discussion for more details on the role of variable hints.

18.11 ORD format

A priority ordering (ORD) file is used to input a set of variable priority orders. Reading a priority file (using GRBread, for example) modifies the MIP branch variable selection. When choosing a branching variable from among a set of fractional variables, the Gurobi MIP solver will always choose a variable with higher priority over one with a lower priority.

The file consists of variable-value pairs, each on its own line. The file contains at most one line for each variable in the model. Any line that starts with the hash sign (#) is treated as a comment line and is ignored. The following is a simple example:

```
# Branch priority file
x 1
y 1
z -1
```

Variables have a default branch priority value of 0, so it is not necessary to specify values for all variables.

Importing a priority order file is equivalent to replacing the BranchPriority attribute value for each variable in the model. Note that you can still modify the `BranchPriority` attribute after importing an ordering file.

18.12 BAS format

An LP basis (BAS) file is used to specify an initial basis for a continuous model. The file provides basis status information for each variable and constraint in the model. If written by Gurobi, to reduce the size of the file, it only includes the difference to the slack basis. In a slack basis for each row the corresponding slack variable is basic while all other problem variables are at their lower bound. If a basis has been imported into a continuous model before optimization begins (using GRBread, for example), and if a simplex optimizer has been selected (through the Method parameter), the Gurobi simplex optimizer begins from the specified basis.

A BAS file begins with a `NAME` line, and ends with an `ENDATA` statement. No information is retrieved from these lines, but they are required by the format. Between these two lines are basis status lines, each consisting of two or three fields and starting with a white space character. If the first field is `LL`, `UL`, or `BS`, the variable named (slack variables are not allowed) in the second field is non-basic at its lower bound, non-basic at its upper bound, or basic, respectively. Any additional fields are ignored. If the first field is `XL` or `XU`, the variable named in the second field is basic, while the row named in the third field states that the corresponding slack variable is non-basic at its lower or upper bound, respectively.

The following is a simple example:

```
NAME  example.bas
 XL x1 c1
 XU x2 c2
 UL x3
 LL x4
ENDATA
```

Importing a basis into a model is equivalent to setting the VBasis and CBasis attributes for each listed variable and constraint to the specified basis status.

A near-optimal basis can speed the solution of a difficult LP model. However, specifying a start basis that is not extremely close to an optimal solution will often slow down the solution process. Exercise caution when providing start bases.

18.13 SOL format

A Gurobi solution (SOL) file is used to output a solution vector. It can be written (using GRBwrite, for example) whenever a solution is available.

The file consists of variable-value pairs, each on its own line. The file contains one line for each variable in the model. The following is a simple example:

```
# Solution file
x  1.0
y  0.5
z  0.2
```

18.14 JSON solution format

JSON (or JavaScript Object Notation) is a lightweight, text-based, language-independent data interchange format. It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data.

Gurobi JSON solution format is meant to be a simple and standard way to capture and share optimization results. It conforms to the RFC-8259 standard. JSON solutions can be written to a file or captured in a string.

The JSON solution captures the values of various Gurobi attributes associated with the solution to the model. Some are related to the model overall, some to individual variables, and some to individual constraints. The exact contents of a JSON solution string will depend on a few factors:

- The type of model being solved (linear, quadratic, mixed-integer, multi-objective, etc.). Some solution information is simply not available for certain problem types (e.g., dual variable values for MIP models).

- The set of tagged elements in the model. By default the JSON solution will contain only model attributes and variable data. Users can tag variables (using the VTag attribute), linear constraints (using the CTag attribute), and quadratic constraints (using the QCTag attribute) to request data on a more selective basis. If any such attribute is used, only tagged elements will have solution information in the JSON solution.

- The JSONSolDetail parameter, which controls how much detail is included in the JSON solution.

- Parameter settings such as InfUnbdInfo or QCPDual, which cause the optimization process to generate more solution information.

JSON solutions aren't generally meant to be interpreted directly by humans. Instead, you typically feed them into a JSON parser, which provides tools for extracting the desired information from the string. JSON is a widely-used format, and nearly all modern program languages have libraries available for helping to parse JSON strings and files. And if you are determined to examine the string directly, JSON parsers typically also include pretty-printing utilities that make it easier to do so.

Basic Structure

A JSON solution string consists of a collection of named components. In its simplest form, it might look like the following:

```
{ "SolutionInfo": { "Status": 3,
                    "Runtime": 3.4289908409118652e-01,
                    "BoundVio": 0,
                    "ConstrVio": 0,
                    "IterCount": 0,
                    "BarIterCount": 0}}
```

A JSON parser makes it relatively easy to extract the various components from this string. In Python, for example, you would be able to retrieve the optimization status by accessing

```
result['SolutionInfo']['Status']
```

after parsing.

Before discussing the specific information that is available in this format, let us first say a word about how data is represented. The type of each data item follows from the attribute type. For example, Status is an integer attribute, so the corresponding value is stored as an integer. Runtime is a double attribute, which is represented as a string, and that string always captures the exact, double-precision (IEEE754) value of the attribute.

Named Components

A JSON solution will always have at least one named object: `SolutionInfo`. It may contain up to three optional named arrays: `Vars, Constrs, QConstrs`. A JSON solution string may look like:

```
{"SolutionInfo": {...},
 "Vars": [...],
 "Constrs": [...],
 "QConstrs": [...]}
```

The exact contents of the three optional sections will depend on what model components have been tagged and on what solution information is available. If no element is tagged, for example, then the `Vars` array will be present and contain names and solution values for all variables with non-zero solution values. For a MIP model, the `Constrs` array will not be present, since MIP solutions don't contain any constraint information.

SolutionInfo Object

The `SolutionInfo` object contains high-level information about the solution that was computed for this model. Some entries will always be present, while others depend on the problem type or the results of the optimization. This component may include the following model attributes:

`Status (always present)`: The optimization status (optimal, infeasible, hit the time limit, etc.).

`Runtime (always present)`: Runtime for the optimization (in seconds).

`ObjVal`: The solution objective value.

`ObjBound`: The best known bound on the objective value.

ObjBoundC: The best known bound on the objective value (before using integrality information to strengthen the bound).

MIPGap: The optimality gap.

IntVio: The maximum integrality violation.

BoundVio: The maximum bound violation.

ConstrVio: The maximum constraint violation.

ObjNVal (multi-objective only): An array of objective values, one for each objective.

ScenNObjVal (multi-scenario only): An array of objective values, one for each scenario.

ScenNObjBound (multi-scenario only): An array of objective bounds, one for each scenario.

IterCount: Number of simplex iterations.

BarIterCount: Number of barrier iterations.

NodeCount: Number of branch-and-cut nodes explored for MIP models.

FarkasProof: Part of the infeasibility certificate for infeasible models. Note that you have to set the InfUnbdInfo parameter before the optimization call for this information to be available.

SolCount: Number of stored solutions (only for MIP models).

PoolObjBound: Bound on the objective of undiscovered MIP solutions.

PoolObjVal: Only for MIP models with at least one solution. For single-objective models, this is an array containing the objective value for each stored solution (starting with the incumbent). For multi-objective models, this is an array containing SolCount arrays of values, each array contains the objective value for each multi-objective for the particular solution.

Here's an example of a SolutionInfo object for a MIP model:

```
{ "SolutionInfo": { "Status": 2,
                    "Runtime": 5.8451418876647949e+00,
                    "ObjVal": 3089,
                    "ObjBound": 3089,
                    "ObjBoundC": 3089,
                    "MIPGap": 0,
                    "IntVio": 0,
                    "BoundVio": 0,
                    "ConstrVio": 0,
                    "IterCount": 32,
                    "BarIterCount": 0,
                    "NodeCount": 1,
                    "SolCount": 1,
                    "PoolObjBound": 3089,
                    "PoolObjVal": [ 3089]}}
```

Vars Array

The `Vars` component is an array (possibly empty) of objects containing information about variables. If no explicit tags (VTag, CTag, or QCTag) have been set at all, all variables with a non-zero objective value will be included, along with their names. Otherwise only variables wih a set VTag will be included, and this tag will be part of the object. Some data will always be present, while other data will depend on the problem type or the results of the optimization. This component may include the following variable attributes:

`VarName`: The variable's name in the model. Present only if no tags have been set.

`VTag`: Array containing the variable tag. Note that this is stored as an array, but the array will currently only ever contain a single string.

`X (always present)`: Value for the variable corresponding to the VarName or VTag in the current solution. Note objects with a zero variable value will be omitted from the Vars array unless `JSONSolDetail` is greater than zero.

`Xn`: Values for all stored solutions including the incumbent solution (only for MIP).

`ScenNX`: For multiple scenarios, values for all scenario solutions.

`RC`: For continuous models with dual information, the reduced cost for the variable.

`VBasis`: For continuous models whose solution is basic, the basis status for the variable.

`UnbdRay`: For unbounded models with InfUnbdInfo enabled, the unbounded ray component associated with the variable.

The following attributes are only included if `JSONSolDetail` is greater than 0: `RC`, `UnbdRay`, `VBasis`, `Xn`.

These objects may look like:

```
{ "VTag": ["VTag7"], "X": 1}
{ "VTag": ["VTag12"], "X": 3.6444895037909271e-02, "RC": 0}
{ "VTag": ["VTag2747"],
  "X": 0,
  "Xn": [ 0, 1, 1, 1, 0, 1, 1, 0, 0, 0]}
```

Constrs Array

The `Constrs` component is an array (possibly empty) of objects containing information about tagged linear constraints. Some entries will always be present, while others depend on the problem type or the results of the optimization. This component may include the following constraint attributes:

`CTag (always present)`: Array containing the linear constraint tag. Note that this is stored as an array, but the array will currently only ever contain a single string.

`Slack (always present)`: Value for the slack variable in the current solution.

`Pi`: For continuous models with dual information, the dual value for the corresponding constraint.

FarkasDual: For infeasible models with InfUnbdInfo enabled, the Farkas dual component associated with the constraint.

This component will always be empty for MIP models.

The following attributes are only included if JSONSolDetail is greater than 0: CBasis, FarkasDual, Pi, Slack.

These objects may look like:

```
{ "CTag": ["CTag72"],
  "Slack": -1.3877787807814457e-17,
  "Pi": -5.6530866311690423e-02}
```

QConstrs Array

The QConstrs component is an array (possibly empty) of objects containing information about tagged quadratic constraints. Some entries will always be present, while others depend on the problem type or the results of the optimization. This component may include the following quadratic constraint attributes:

QCTag (always present): Array containing the quadratic constraint tag. Note that this is stored as an array, but the array will currently only ever contain a single string.

QCSlack (always present): Value for the slack variable in the current solution.

QCPi: For continuous models with dual information, the dual value for the corresponding constraint.

This component will always be empty for MIP models.

The following attributes are only included if JSONSolDetail is greater than 0: QCPi, QCSlack.

JSON Solution Examples

For a continuous model, the JSON solution string may look like

```
{ "SolutionInfo": {
     "Status": 2,
     "Runtime": 9.9294495582580566e-01,
     "ObjVal": 5.2045497375374854e-07,
     "BoundVio": 0,
     "ConstrVio": 1.002e-07,
     "IterCount": 0,
     "BarIterCount": 3},
  "Vars": [
     {"VTag": ["VTag7"], "X": -3.0187172916263982e-09, "RC": 0},
     {"VTag": ["VTag1340"], "X": -3.0696132844593768e-09, "RC": 0},
     {"VTag": ["VTag2673"], "X": -4.8134359014615295e-09, "RC": 0},
     {"VTag": ["VTag4006"], "X": -7.1652420015125937e-02, "RC": 0},
     {"VTag": ["VTag5339"], "X": -1.5815441619302997e-02, "RC": 0},
     {"VTag": ["VTag6672"], "X": 1.4945278866946186e-02, "RC": 0}],
  "Constrs": [
     {"CTag": ["CTag7"], "Slack": 4.85722506e-17, "Pi": 2.3140310696e-06},
```

```
     {"CTag": ["CTag673"], "Slack": 0, "Pi": -1.4475853138350041e-06},
     {"CTag": ["CTag1339"], "Slack": -2.7758914e-17, "Pi": -3.7443785e-06},
     {"CTag": ["CTag2005"], "Slack": 4.3420177e-18, "Pi": -1.0277524e-06},
     {"CTag": ["CTag2671"], "Slack": -1.3895245e-17, "Pi": 8.0012944e-07},
     {"CTag": ["CTag3337"], "Slack": 6.39465e-16, "Pi": -5.3368958e-06}]}
```


For a multi-objective LP, the JSON solution string may look like


```
{ "SolutionInfo": {
     "Status": 2,
     "Runtime": 2.2838807106018066e-01,
     "ObjNVal": [ 10, 339],
     "IterCount": 112,
     "BarIterCount": 0,
     "NodeCount": 0},
  "Vars": [
     {"VTag": ["VTag7"], "X": 0},
     {"VTag": ["VTag569"], "X": 0},
     {"VTag": ["VTag1131"], "X": 0},
     {"VTag": ["VTag1693"], "X": 0},
     {"VTag": ["VTag2255"], "X": 0},
     {"VTag": ["VTag2817"], "X": 0},
     {"VTag": ["VTag3379"], "X": 0},
     {"VTag": ["VTag3941"], "X": 0},
     {"VTag": ["VTag4503"], "X": 0},
     {"VTag": ["VTag5065"], "X": 1},
     {"VTag": ["VTag5627"], "X": 1},
     {"VTag": ["VTag6189"], "X": 1}]}
```


For a regular MIP problem, the JSON solution string may look like


```
{ "SolutionInfo": {
     "Status": 2,
     "Runtime": 2.4669170379638672e-03,
     "ObjVal": 3124,
     "ObjBound": 3124,
     "ObjBoundC": 3124,
     "MIPGap": 0,
     "IntVio": 1.958742e-08,
     "BoundVio": 0,
     "ConstrVio": 1.002e-07,
     "IterCount": 465,
     "BarIterCount": 0,
     "NodeCount": 1,
     "SolCount": 4,
     "PoolObjBound": 3124,
     "PoolObjVal": [ 3124, 3124, 3124, 3124]},
```

```
  "Vars": [
     {"VTag": ["VTag7"], "X": 1, "Xn": [ 1, 1, 1, 1]},
     {"VTag": ["VTag466"], "X": 0, "Xn": [ 0, 1, 1, 0]},
     {"VTag": ["VTag925"], "X": 0, "Xn": [ 0, 0, 0, 0]},
     {"VTag": ["VTag1384"], "X": 0, "Xn": [ 0, 0, 1, 1]},
     {"VTag": ["VTag1843"], "X": 0, "Xn": [ 0, 1, 0, 0]},
     {"VTag": ["VTag2302"], "X": 0, "Xn": [ 0, 1, 1, 0]}]}
```

For a multi-objective MIP, the JSON solution string may look like

```
{ "SolutionInfo": {
     "Status": 2,
     "Runtime": 3.5403838157653809e+00,
     "ObjNVal": [ 2763, 704],
     "IterCount": 595,
     "BarIterCount": 0,
     "NodeCount": 1,
     "SolCount": 6,
     "PoolObjVal": [ [ 2763, 704 ], [ 2763, 705 ],
                     [ 2763, 716 ], [ 2763, 718 ],
                     [ 2763, 769 ], [ 2763, 1060 ]]},
  "Vars": [
     {"VTag": ["VTag7"], "X": 1, "Xn": [ 1, 1, 1, 1, 1, 1]},
     {"VTag": ["VTag466"], "X": 0, "Xn": [ 0, 1, 0, 0, 0, 0]},
     {"VTag": ["VTag925"], "X": 0, "Xn": [ 0, 0, 0, 0, 1, 1]},
     {"VTag": ["VTag1384"], "X": 0, "Xn": [ 0, 0, 0, 0, 0, 0]},
     {"VTag": ["VTag1843"], "X": 0, "Xn": [ 0, 0, 1, 1, 0, 0]},
     {"VTag": ["VTag2302"], "X": 0, "Xn": [ 0, 1, 0, 0, 0, 0]}]}
```

For a multi-scenario model, the JSON solution string may look like

```
{ "SolutionInfo": {
     "Status": 2,
     "Runtime": 3.5403838157653809e+00,
     "ObjVal": 2763,
     "ObjBound": 2763,
     "ObjBoundC": 1324,
     "IntVio": 0,
     "BoundVio": 0,
     "ConstrVio": 0,
     "ScenNObjVal": [2763, 3413, 1e+100],
     "ScenNObjBound": [2763, 3413, 1e+100],
     "IterCount": 595,
     "BarIterCount": 0,
     "NodeCount": 1,
     "SolCount": 3,
     "PoolObjBound": 2763,
```

```
    "PoolObjVal": [ 2763, 2763, 2763]},
  "Vars": [
    {"VTag": ["VTag7"], "X": 1, "ScenNX": [1, 0, 1e+101], "Xn": [ 1, 0, 1]},
    {"VTag": ["VTag466"], "X": 0, "ScenNX": [1, 1, 1e+101], "Xn": [ 1, 1, 1]},
    {"VTag": ["VTag925"], "X": 0, "ScenNX": [0, 0, 1e+101], "Xn": [ 0, 0, 0]},
    {"VTag": ["VTag1384"], "X": 0, "ScenNX": [2, 1, 1e+101], "Xn": [ 2, 1, 0]},
    {"VTag": ["VTag1843"], "X": 0, "ScenNX": [0, 2, 1e+101], "Xn": [ 0, 2, 1]},
    {"VTag": ["VTag2302"], "X": 0, "ScenNX": [0, 1, 1e+101], "Xn": [ 0, 1, 0]}]}
```

If the scenario objective value ScenNObjVal is infinite (`GRB_INFINITY = 1e+100` for minimization, `-GRB_INFINITY = -1e+100` for maximization), then no feasible solution has been found for this scenario. The corresponding ScenNX value for each variable will be `GRB_UNDEFINED = 1e+101`. Moreover, if the ScenNObjBound value for the scenario is also infinite, it means that the scenario has been proven to be infeasible.

18.15 ATTR format

A Gurobi attribute (ATTR) file is used to store, and read, attribute information of a model -- provided by the user or generated during a previous solution call -- that affects the optimization process.

More precisely, this file stores -- if available -- the following attributes: X, the primal solution to the last optimization call. Pi, the dual solution to the last optimization call. Start, all the stored MIP start vectors. Partition, the variable partition. VarHintVal and VarHintPri, the variable hint information. BranchPriority, the variable branch priority vector. Lazy, the lazy attribute for each constraint. VTag, CTag and QCTag, the tagged elements of the model. VBasis and CBasis, the basis information for variables and constraints. PStart and DStart, the simplex-start information for variables and constraints.

Any line starting with the character '#', or empty lines, are considered as comments, and will be discarded at reading time. The file should start by a line of the form
`GRB_ATTR_FILE_VERSION 0110000`
which indicate the version of the file. Older versions are backward-compatible. Then follows a number of sections, each storing one set of attributes. Each section starts with one of the following:

- `SECTION SOLUTION`
- `SECTION MIPSTART`
- `SECTION PARTITION`
- `SECTION VARHINTS`
- `SECTION BRANCHPRIORITY`
- `SECTION LAZYCONSTRS`
- `SECTION BASIS`
- `SECTION PSTART`

- SECTION DSTART
- SECTION VTAG
- SECTION CTAG
- SECTION QCTAG

For those attributes for which there is a dedicated file extension, the following format is exactly the same as described in the corresponding file format description.

For sections involving other variable attributes, each line is a tuple describing the variable name, and the attribute(s) value related to it. Variables with attribute values at default may be omitted.

For sections involving other constraint attributes, each line is a tuple describing the constraint name, and the attribute(s) value related to it. Constraints with attribute values at default may be omitted.

If a model has multiple MIP starts, each of them will be saved in a different `SECTION MIPSTART`. Whenever an attribute file is loaded into a model, each `SECTION MIPSTART` will be loaded into a new MIP start vector.

The tags of variables and constraints must be enclosed in double quotes. If the tag itself contains a double quote, this needs to be escaped by a backslash. Moreover, if the tag contains a backslash, this also needs to be escaped, which yields two consecutive backslashes. For example, if a variable named `"V01"` has as a tag the string: `"My Tag"\"`, the corresponding line in the attribute file will contain:

```
V01 "My Tag \"\\"
```

Note that tags can only consist of US-ASCII printable characters.

18.16 PRM format

A Gurobi parameter (PRM) file is used to specify parameter settings. Reading a parameter file (using GRBread, for example) causes the parameters specified in the file to take the specified values.

The file consists of parameter-value pairs, each on its own line. Any line that begins with the hash sign (#) is a comment line and is ignored. The following is a simple example:

```
# Parameter settings
Cuts       2
Heuristics 0.5
```

If an unknown parameter name is listed in the file, a warning is printed and the associated line is ignored.

Note that when you write a Gurobi parameter file (using GRBwrite in **C**, Model.write in Python, or in any of the other APIs), both integer or double parameters not at their default value will be saved, but no string parameter will be saved into the file.

The Gurobi Optimizer produces a log that allows you to track the progress of the optimization. The Gurobi command-line tool and the Gurobi Interactive Shell put the log to both the screen and to a file named `gurobi.log` by default. The other interfaces just put the log to the screen. Screen output can be controlled using the OutputFlag parameter, and file output can be controlled using the LogFile parameter.

19.1 Header

When you solve a model, the first part of the resulting log contains basic information about the model you are solving and the machine you are solving it on. You will first see a line that looks like this:

```
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
```

This shows the number of physical cores and logical processors in the machine. The latter may be larger than the former due to Simultaneous Multi Threading (SMT), which allows one physical core to appear as multiple logical processors. This log line also shows the maximum number of threads that the solver will use in this instance. This depends on the number of (logical) processors in the machine, but also on the number requested (through the Threads parameter), and for some license types on the number enabled by your license.

Next you will see summary information about the model you are solving:

```
Optimize a model with 755 rows, 2756 columns and 8937 nonzeros
Model fingerprint: 0x22935346
Variable types: 0 continuous, 2756 integer (0 binary)
```

The first line shows the size of the model. The second shows the model *fingerprint*, which is a hash value that takes into account the values of all attributes of all components of the model. The intent is that models that differ in any way will most always have different fingerprints. The last log line above provides additional information about variable integrality (for MIP models).

The final portion of the header contains statistics about the constraint matrix:

```
Coefficient statistics:
  Matrix range     [1e+00, 1e+04]
  Objective range  [1e+00, 1e+04]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+04]
```

This information gives a very rough indication of whether you can expect numerical issues when solving your model. Please consult the numerical guidelines section to learn more.

The format of the remainder of the log depends on the algorithm that is invoked (simplex, barrier, sifting, branch-and-cut, or IIS. We now describe the contents of the log for each.

19.2 Simplex Logging

The simplex log can be divided into three sections: the presolve section, the simplex progress section, and the summary section.

Presolve Section

The first thing the Gurobi Optimizer does when optimizing a model is to apply a *presolve* algorithm in order to simplify the model. The first section of the Gurobi log provides information on the extent to which presolve succeeds in this effort. Consider the following example output from NETLIB model `dfl001`:

```
Presolve removed 2349 rows and 3378 columns
Presolve time: 0.04s
Presolved: 3722 rows, 8852 columns, 30908 nonzeros
```

The example output shows that presolve was able to remove 2349 rows and 3378 columns, and it required 0.04 seconds. The final line in the presolve section shows the size of the model after presolve. This is size of the model that is passed to the simplex optimizer. Note that the solution that is computed for this model is automatically transformed into a solution for the original problem once simplex finishes (in a process often called *uncrushing*), but this uncrush step is transparent and produces no log output.

Progress Section

The second section of the Gurobi simplex output provides information on the progress of the simplex method:

```
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -2.4571000e+32   4.000000e+30   2.457100e+02      0s
   18420    1.1265574e+07   1.655263e+03   0.000000e+00      5s
   20370    1.1266393e+07   0.000000e+00   0.000000e+00      6s
```

The five columns in each output row show the number of simplex iterations performed to that point, the objective value for the current basis, the magnitude of the primal infeasibility for the current basis (computed as the sum of the absolute values of all constraint and bound violations), the magnitude of the dual infeasibility (computed as the sum of the absolute values of all dual constraint violations), and the amount of time expended to that point (measured using wall clock time). The default simplex algorithm in the Gurobi solver is dual simplex, which tries to maintain dual feasibility while performing simplex pivots to improve the objective. Thus, once the dual simplex algorithm has found a dual feasible basis, you will generally see a dual infeasibility value of zero. When the primal and dual infeasibilities both reach zero, the basis is optimal and optimization is complete.

By default, the Gurobi Optimizer produces a log line every 5 seconds. The frequency of log lines can be changed by modifying the DisplayInterval parameter (see the Parameter section of this document for more information).

Summary Section

The third section of the simplex log provides summary information. It provides a summary of the work that the simplex algorithm performed, including the iteration count and the runtime, and it

provides information on outcome of the optimization. The summary for a model that is solved to optimality would look like this:

```
Solved in 20370 iterations and 5.75 seconds
Optimal objective  1.126639304e+07
```

Other termination states produce different summaries. For example, a user interrupt would produce a summary that looks like:

```
Stopped in 7482 iterations and 3.41 seconds
Solve interrupted
```

Hitting a time limit would produce a summary that looks like:

```
Stopped in 18082 iterations and 5.00 seconds
Time limit reached
```

19.3 Barrier Logging

The barrier log can be divided into five sections: the presolve section, the barrier preprocessing section, the barrier progress section, the crossover progress section, and the summary section.

Presolve Section

As mentioned earlier, the first thing the Gurobi Optimizer does when optimizing a model is to apply a *presolve* algorithm in order to simplify the model. The first section of the Gurobi log provides information on the extent to which presolve succeeds in this effort. Consider the following example output from NETLIB model `dfl001`:

```
Presolve removed 2349 rows and 3378 columns
Presolve time: 0.04s
Presolved: 3722 rows, 8852 columns, 30908 nonzeros
```

The example output shows that presolve was able to remove 2349 rows and 3378 columns, and it required 0.04 seconds. The final line in the presolve section shows the size of the model after presolve. This is size of the model that is passed to the barrier optimizer. Note that the solution that is computed for this model is automatically transformed into a solution for the original problem once barrier finishes (in a process often called *uncrushing*), but this uncrush step is transparent and produces no log output.

Barrier Preprocessing Section

The factor matrix for the linear system solved in each iteration of the barrier method can be quite large and quite expensive to compute. In order to reduce the cost of this computation, the first step of the barrier algorithm is to compute a fill-reducing reordering of the rows and columns of this matrix. This step can be quite expensive, but the cost is recouped in the lower cost of the subsequent barrier iterations.

Once this fill-reducing reordering has been computed, the Gurobi Optimizer outputs information related to the barrier factor matrix:

```
Barrier statistics:
 AA' NZ     : 3.657e+04
 Factor NZ  : 8.450e+05 (roughly 12 MBytes of memory)
 Factor Ops : 3.944e+08 (less than 1 second per iteration)
 Threads    : 8
```

The First line shows the number of off-diagonal entries in the lower triangle of AA^T. A scaled version of this matrix is factored in each iteration of the barrier algorithm, so the structure of the Cholesky factor depends on the structure of AA^T.

The next two lines indicate the number of non-zero values in the factor matrix, and the number of floating-point operations required to factor it. Note that the log also provides an estimate of how much memory will be needed by the barrier algorithm, and how long each barrier iteration will require: These are rough estimates that are meant to provide a general sense of how difficult the model will be to solve. If you want to obtain an estimate of overall solution time, note that most models achieve convergence in roughly 50 iterations, but there are many exceptions. Crossover runtime is typically comparable to the cost of a few barrier iterations, but this time can vary considerably, depending on the model characteristics.

The final line shows the number of threads that will be used to perform the barrier iterations.

You may sometimes see two other lines in this section:

```
 Dense cols : 3
 Free vars  : 20
```

The first indicates how many columns from the constraint matrix were treated as dense. The second indicates how many variables in the model are free. Dense columns and free variables can sometimes lead to numerical difficulties in the barrier solver, so it is useful to know when they are present.

Progress Section

The third section of the Gurobi barrier output provides information on the progress of the barrier method:

```
                 Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   1.47340463e+12 -1.05838204e+09  1.49e+04 2.46e+02  1.94e+09     0s
   1   6.13234163e+11 -3.97417254e+10  5.97e+03 5.98e+06  8.82e+08     0s
   2   2.89634303e+11 -9.20268188e+10  2.54e+03 2.24e+06  3.81e+08     0s
   3   6.57753936e+10 -9.40746258e+10  2.39e+02 2.87e+05  5.17e+07     0s
   4   2.44710457e+10 -2.59852944e+10  3.16e+01 3.01e+04  9.00e+06     0s
   5   6.74069830e+09 -1.78169224e+10  4.01e+00 2.01e+04  3.17e+06     0s
   6   1.93163205e+09 -3.10778084e+09  2.46e-01 3.13e+03  5.62e+05     0s
   7   6.54973737e+08 -6.89946649e+08  4.40e-02 5.35e+02  1.47e+05     0s
   8   2.44764500e+08 -3.47987016e+08  1.47e-02 4.02e+02  6.46e+04     0s
   9   1.35906001e+08 -1.41063037e+08  7.16e-03 1.66e+02  3.01e+04     0s
  10   9.29132721e+07 -6.69973369e+07  4.58e-03 7.60e+01  1.73e+04     0s
```

The seven columns in each output row show the number of barrier iterations performed to that point, the primal and dual objective values for the current barrier iterate, the magnitude of the

primal and dual infeasibilities for the current iterate (computed as the infinity-norms of the primal and dual residual vectors, respectively), the magnitude of the complementarity violation of the current primal and dual iterates (the dot product of the primal solution and the dual reduced cost vector), and the amount of time expended to that point (measured using wall clock time). When the primal infeasibility, dual infeasibility, and complementarity satisfy barrier convergence tolerances (controlled using the BarConvTol parameter), the solution is declared optimal and optimization is complete.

Unlike the simplex and MIP optimizers, the barrier optimizer produces a log line for each iterate, independent of the value of the DisplayInterval parameter.

You may sometimes see a star after the iteration count in the barrier progress log:

```
 15   2.42800468e+03  8.54543324e+04  1.68e+02 1.02e-09  8.30e+04     0s
 16   4.05292006e+03  4.65997441e+04  1.82e+02 2.50e-01  4.25e+04     0s
 17*  4.88742259e+08  4.30781025e+10  5.17e+00 1.31e-01  2.52e-02     0s
 18*  1.21709951e+06  3.39471138e+13  8.55e-06 3.14e-06  3.14e-05     0s
 19* -1.38021972e+06  3.31580578e+16  3.42e-08 8.20e-09  3.22e-08     0s
 20*  1.25182178e+06  3.31575855e+19  6.54e-12 7.34e-09  3.22e-11     0s
```

This indicates that the model may be primal or dual infeasible. Note that these intermediate indications of infeasibility won't necessarily turn into an infeasibility proof, so the star may disappear in later iterations.

Crossover Section

The fourth section of the barrier log provides information on the crossover step. This section is only present when crossover is selected (as controlled through the Crossover parameter. Crossover converts the interior point solution produced by the barrier algorithm to a basic solution.

The first stage in crossover is to *push* variables to bounds in order to obtain a valid basic solution. By default, this is done for dual variables first, then for primal variables. Progress of this phase is tracked with this portion of the crossover log...

```
Crossover log...

    1610 DPushes remaining with DInf 0.0000000e+00                 1s
       0 DPushes remaining with DInf 0.0000000e+00                 1s

     144 PPushes remaining with PInf 5.7124800e-03                 1s
       0 PPushes remaining with PInf 5.7124800e-03                 1s

  Push phase complete: Pinf 5.7124800e-03, Dinf 8.1488315e-07      1s
```

Each line indicates how many push steps remain, the amount of infeasibility in the current solution, and the elapsed barrier time.

Upon completion of the push phase, crossover has a basic solution that isn't necessarily optimal. The resulting basis is passed to simplex, and simplex completes the optimization...

```
Iteration    Objective       Primal Inf.    Dual Inf.      Time
    1700    1.1266352e+07   5.712480e-03   0.000000e+00      1s
    1868    1.1266393e+07   0.000000e+00   0.000000e+00      1s
```

The five columns in each output row of the simplex log show the number of simplex iterations performed to that point in the crossover algorithm (including the push steps), the objective value for the current basis, the magnitude of the primal infeasibility for the current basis (computed as the sum of the absolute values of all constraint and bound violations), the magnitude of the dual infeasibility (computed as the sum of the absolute values of all dual constraint violations), and the amount of time expended by the crossover algorithm to that point (measured using wall clock time). When the primal and dual infeasibilities both reach zero, the basis is optimal and optimization is complete.

Summary Section

The final section of the barrier log provides summary information. It provides a summary of the work that the barrier algorithm performed, including the iteration count and the runtime, and it provides information on outcome of the optimization. The summary for a model that is solved to optimality would look like this:

```
Solved in 1868 iterations and 1.05 seconds
Optimal objective  1.126639304e+07
```

Other termination states produce different summaries. For example, a user interrupt would produce a summary that looks like:

```
Stopped in 7482 iterations and 3.41 seconds
Solve interrupted
```

Hitting a time limit would produce a summary that looks like:

```
Stopped in 9221 iterations and 5.00 seconds
Time limit exceeded
```

19.4 Sifting Logging

Sifting will sometimes be used within the dual simplex method, either as a result of an automatic choice by the Gurobi Optimizer or because the user selected it through the Sifting parameter. The sifting log consists of three sections: the presolve section, the sifting progress section, and the summary section. The first and last are identical to those for simplex, so we'll only discuss the middle section here.

Sifting Progress Section

As we mentioned, output for sifting and dual simplex are indistinguishable until the progress section begins. For sifting, the progress section begins with a clear indication that sifting has been selected:

```
Starting sifting (using dual simplex for sub-problems)...
```

The sifting algorithm performs a number of major iterations, where each iteration solves a smaller LP sub-problem. It uses the result to update the current primal and dual solution. The sifting log prints one line per major iteration, with information on the current primal and dual objective values:

```
 Iter     Pivots    Primal Obj      Dual Obj        Time
    0          0     infinity      2.0000000e+01     11s
    1       4662   1.5220652e+03   2.7034420e+02     12s
    2       8917   1.3127217e+03   4.6530259e+02     13s
    3      16601   1.1651147e+03   6.4767742e+02     17s
    4      30060   1.0881514e+03   7.8842688e+02     29s
    5      45169   1.0618879e+03   8.8656855e+02     46s
    6      59566   1.0549766e+03   9.5404159e+02     64s
    7      73614   1.0540577e+03   1.0172213e+03     82s
```

The first column in the log gives the major iteration number. The second shows the total number of simplex iterations performed in solving the sifting sub-problems. The third and fourth columns show the primal and dual objective values for the current solution. The final column shows elapsed runtime.

The completion of sifting is indicated with the following message:

```
Sifting complete
```

The basis computed by sifting is then handed back to dual simplex, and the log from that point forward comes from the dual simplex algorithm.

19.5 MIP Logging

The MIP log can be divided into three sections: the presolve section, the progress section, and the summary section.

Presolve Section

As with the simplex and barrier logs, the first section of the MIP log is the presolve section. Here is presolve output for MIPLIB model `mas76`:

```
Presolve removed 0 rows and 3 columns
Presolve time: 0.01s
Presolved: 12 rows, 148 columns, 1615 nonzeros
Variable types: 1 continuous, 147 integer (145 binary)
```

In this example, presolve was able to remove 3 columns. The last two lines show the size of the model that is passed to the branch-and-cut algorithm and the types of remaining variables.

Progress Section

The next section in the MIP log tracks the progress of the branch-and-cut search. The search involves a number of different steps, so this section typically contains a lot of detailed information. The first thing to observe in the log for example `mas76` is this line:

```
Found heuristic solution: objective 157344.61033
```

It indicates that the Gurobi heuristics found an integer-feasible solution before the root relaxation was solved.

The next thing you will see in the log is the root relaxation solution display. For a model where the root solves quickly, this display contains a single line:

```
Root relaxation: objective 3.889390e+04, 50 iterations, 0.00 seconds
```

For models where the root relaxation takes more time (MIPLIB model `dano3mip`, for example), the Gurobi solver will automatically include a detailed simplex log for the relaxation itself:

```
Root simplex log...

Iteration    Objective       Primal Inf.    Dual Inf.      Time
   15338    5.7472018e+02   6.953458e+04   0.000000e+00      5s
   19787    5.7623162e+02   0.000000e+00   0.000000e+00      7s

Root relaxation: objective 5.762316e+02, 19787 iterations, 6.18 seconds
```

To be more precise, this more detailed log is triggered whenever the time to solve the root relaxation exceeds the DisplayInterval parameter value (5 seconds by default).

The next section provides progress information on the branch-and-cut tree search:

```
    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 38893.9036    0   11 157344.610 38893.9036  75.3%     -    0s
H    0     0                      80297.610430 38893.9036  51.6%     -    0s
H    0     0                      60361.518931 38893.9036  35.6%     -    0s
H    0     0                      41203.601476 38893.9036  5.61%     -    0s
     0     0 38923.3264    0   12 41203.6015 38923.3264  5.53%     -    0s
     0     0 38923.3264    0   12 41203.6015 38923.3264  5.53%     -    0s
H    0     0                      40697.054142 38923.3264  4.36%     -    0s
     0     0 38923.3264    0   13 40697.0541 38923.3264  4.36%     -    0s
H    0     0                      40005.054142 38923.3264  2.70%     -    0s
     0     0 38939.3131    0   15 40005.0541 38939.3131  2.66%     -    0s
     0     0 38964.7042    0   13 40005.0541 38964.7042  2.60%     -    0s
     0     0 39004.6387    0   15 40005.0541 39004.6387  2.50%     -    0s
     0     0 39008.7922    0   15 40005.0541 39008.7922  2.49%     -    0s
     0     0 39008.9356    0   12 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   14 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   15 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   15 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   16 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   17 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   17 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   19 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   18 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   18 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   19 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   19 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   18 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   18 40005.0541 39008.9356  2.49%     -    0s
```

```
     0     0 39008.9356    0   18 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   18 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   18 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   19 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   21 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   21 40005.0541 39008.9356  2.49%     -    0s
     0     0 39008.9356    0   19 40005.0541 39008.9356  2.49%     -    0s
     0     2 39008.9356    0   19 40005.0541 39008.9356  2.49%     -    0s
```

This display is somewhat dense with information, but each column is hopefully fairly easy to understand. The `Nodes` section (the first two columns) provides general quantitative information on the progress of the search. The first column shows the number of branch-and-cut nodes that have been explored to that point, while the second shows the number of leaf nodes in the search tree that remain unexplored. At times, there will be an `H` or `*` character at the beginning of the output line. These indicate that a new feasible solution has been found, either by a MIP heuristic (`H`) or by branching (`*`).

The `Current Node` section provides information on the specific node that was explored at that point in the branch-and-cut tree. It shows the objective of the associated relaxation, the depth of that node in the branch-and-cut tree, and the number of integer variables that have non-integral values in the associated relaxation.

The `Objective Bounds` section provides information on the best known objective value for a feasible solution (i.e., the objective value of the current incumbent), and the current objective bound provided by leaf nodes of the search tree. The optimal objective value is always between these two values. The third column in this section (`Gap`) shows the relative gap between the two objective bounds. When this gap is smaller than the MIPGap parameter, optimization terminates.

The `Work` section of the log provides information on how much work has been performed to that point. The first column shows the average number of simplex iterations performed per node in the branch-and-cut tree. The final column shows the elapsed time since the solve began.

By default, the Gurobi MIP solver prints a log line every 5 seconds (although the interval can sometimes be longer for models with particularly time-consuming nodes). The interval between log lines can be adjusted with the DisplayInterval parameter (see the Parameter section of this document for more information).

Note that the explored node count often stays at 0 for an extended period. This means that the Gurobi MIP solver is processing the root node. The Gurobi solver can often expend a significant amount of effort on the root node, generating cutting planes and trying various heuristics in order to reduce the size of the subsequent branch-and-cut tree.

Summary Section

The third section in the log provides summary information once the MIP solver has finished:

```
Cutting planes:
  Gomory: 1
  MIR: 17

Explored 313128 nodes (1741251 simplex iterations) in 4.80 seconds
Thread count was 8 (of 8 available processors)
```

```
Solution count 7: 40005.1 40697.1 41203.6 ... 157345

Optimal solution found (tolerance 1.00e-04)
Best objective 4.000505414200e+04, best bound 4.000505414200e+04, gap 0.0000%
```

The first part of the summary reports the cutting planes that are in the LP relaxation at the end of the MIP search. In this example, there are 18 cuts: 1 of type Gomory, and 17 of the type MIR (Mixed Integer Rounding). Note that more cuts and cuts of other types may have been separated and used by Gurobi but if they are not in the last LP formulation they are not reported here. The different types of cuts are listed in the MIP Cuts section of the MIP Cuts section of the parameter manual.

The solver required just under 5 seconds to solve the model to optimality, and it used 8 threads to do so (the thread count can be limited with the Threads parameter).

The solution count provides the number of solutions found during the optimization process and stored in the Solution Pool (7 in this case), as well as the objective values of these solutions. The maximum number of solutions stored in the pool is the value of the PoolSolutions parameter.

The gap between the best feasible solution objective and the best bound is 0.0%, which produces an `Optimal` termination status, since the achieved gap is smaller than the default MIPGap parameter value.

19.6 Solution Pool and Multi-Scenario Logging

Populating a solution pool or solving for multiple scenarios involves looking for more than one solution, which leads to different logging output. In particular, logging for these methods comes in two phases. In the first, the log shows progress towards finding one provably optimal solution (for multi-scenario optimization, this is the best solution over all scenarios). The log for this first phase is identical to the standard MIP log. It shows progress in the lower and upper bounds, and the phase terminates when these are sufficiently close to each other.

The second phase starts once attention has shifted towards finding solutions beyond that one optimal solution. You will see a message indicating that a new phase has begun. When populating a solution pool, you will see:

```
Optimal solution found at node 7407 - now completing solution pool...
```

When solving for multiple scenarios, you will see:

```
Optimal solution found at node 15203 - now completing multiple scenarios...
```

You will also see an additional header, which is slightly different from the standard MIP header. For a solution pool:

```
    Nodes    |    Current Node    |      Pool Obj. Bounds      |     Work
             |                    |   Worst                    |
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time
```

For multiple scenarios:

```
    Nodes    |    Current Node    |    Scenario Obj. Bounds    |     Work
             |                    |   Worst                    |
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time
```

The most important difference versus the standard header is in the `Incumbent` column. In the standard MIP log, this column shows the objective value for the *best* solution found so far. For a solution pool or multiple scenarios, this column shows the objective value for the *worst* solution. This of course isn't the worst solution ever found. Rather, it shows the objective value for the worst solution among all the solutions that the MIP solver has been asked to find. For example, if you have set the PoolSolutions parameter to 100 to ask for the 100 best solutions, this column will show the objective value for the 100th best solution found so far. If you are solving a multi-scenario model, this column shows the worst solution found over all scenarios. As the search progresses, the value in this column will improve monotonically as the MIP solver replaces this worst solution with better solutions.

One other important difference in this second phase log is in the meaning of the `BestBd` column. In the standard MIP log, this column gives a bound on the best possible objective value for any solution. In this log, this column shows the best possible objective value for any solution that has not yet been found. To give an example, if a minimization model has a unique optimal solution at objective 100, the second phase will begin once the lower bound reaches 100, and the `BestBd` column will show a value larger than 100 once the solver has determined that only one solution exists at objective 100.

The `BestBd` and `Incumbent` columns allow you to track progress towards completion of the solution pool or multi-scenario solve. Specifically, once the best bound for any solution that has not yet been found reaches the objective value for the worst solution, we know that we can't improve that solution and we can stop.

19.7 Multi-Objective Logging

The contents of the log for a multi-objective solve will depend on the approach you use to solve the model. As noted in the section on multi-objective models, you have two options. In a blended approach, where the objectives are combined into a single objective, the log will be the same as what you'd see for a single-objective model. When using a hierarchical approach, a series of models is solved, one for each objective priority level. If your model is a pure hierarchical multi-objective problem with three objectives, the optimization process log will start with

```
------------------------------------------------------------------------
Multi-objectives: starting optimization with 3 objectives ...
------------------------------------------------------------------------
```

If your model is a mixed hierarchical-blended multi-objective problem with five objectives but only three priorities, the optimization log will start with

```
------------------------------------------------------------------------
Multi-objectives: starting optimization with 5 objectives (3 combined) ...
------------------------------------------------------------------------
```

You will also see a log for each solve, introduced by a small header

```
------------------------------------------------------------------------
Multi-objectives: optimize objective 1 Name ...
------------------------------------------------------------------------
```

Where `Name` will be the name of the objective function being optimize, or `(weighted)` if the objective function is the result of blending more than one objective function.

The logs for the individual solves will again be the same as what you'd see for a single-objective model.

19.8 Distributed MIP Logging

Logging for distributed MIP is very similar to the standard MIP logging. The main difference is in the progress section. The header for the standard MIP logging looks like this:

```
    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time
```

In contrast, the distributed MIP header has a different label for the second-to-last field:

```
    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | ParUtil Time
```

Instead of showing iterations per node, this field in the distributed log shows parallel utilization. Specifically, it shows the fraction of the preceding time period (the time since the previous progress log line) that the workers spent actively processing MIP nodes.

Here is an example of a distributed MIP progress log:

```
    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | ParUtil Time

H    0                            157344.61033          -      -          0s
H    0                            40707.729144          -      -          0s
H    0                            28468.534497          -      -          0s
H    0                            18150.083886          -      -          0s
H    0                            14372.871258          -      -          0s
H    0                            13725.475382          -      -          0s
     0     0 10543.7611    0   19 13725.4754 10543.7611  23.2%   99%    0s
*  266                            12988.468031 10543.7611  18.8%          0s
H 1503                            12464.099984 10630.6187  14.7%          0s
* 2350                            12367.608657 10632.7061  14.0%          1s
* 3360                            12234.641804 10641.4586  13.0%          1s
H 3870                            11801.185729 10641.4586  9.83%          1s

Ramp-up phase complete - continuing with instance 2 (best bd 10661)

 16928  2731 10660.9626    0   12 11801.1857 10660.9635  9.66%   99%    2s
 135654 57117 11226.5449   19   12 11801.1857 11042.3036  6.43%   98%    5s
 388736 135228 11693.0268   23   12 11801.1857 11182.6300  5.24%   96%   10s
```

```
 705289 196412     cutoff           11801.1857 11248.8963  4.68%   98%   15s
 1065224 232839 11604.6587   28   10 11801.1857 11330.2111  3.99%   98%   20s
 1412054 238202 11453.2202   31   12 11801.1857 11389.7119  3.49%   99%   25s
 1782362 209060     cutoff           11801.1857 11437.2670  3.08%   97%   30s
 2097018 158137 11773.6235   20   11 11801.1857 11476.1690  2.75%   92%   35s
 2468495 11516     cutoff           11801.1857 11699.9393  0.86%   78%   40s
 2481830     0     cutoff           11801.1857 11801.1857  0.00%   54%   40s
```

One thing you may find in the progress section is that node counts may not increase monotonically. Distributed MIP tries to create a single, unified view of node numbers, but with multiple machines processing nodes independently, possibly at different rates, some inconsistencies are inevitable.

Another difference is the line that indicates that the distributed ramp-up phase is complete. At this point, the distributed strategy transitions from a concurrent approach to a distributed approach. The log line indicates which worker was the *winner* in the concurrent approach. Distributed MIP continues by dividing the partially explored MIP search tree from this worker among all of the workers.

Another difference in the distributed log is in the summary section. The distributed MIP log includes a breakdown of how runtime was spent:

```
Runtime breakdown:
Runtime breakdown:
  Active:  37.85s (93%)
  Sync:     2.43s (6%)
  Comm:     0.34s (1%)
```

This is an aggregated view of the utilization data that is displayed in the progress log lines. In this example, the workers spent 93% of runtime actively working on MIP nodes, 6% waiting to synchronize with other workers, and 1% communicating data between machines.

19.9 IIS Logging

Recall that an Irreducible Infeasible Subsystem (IIS) is a subset of the constraints and variable bounds in your infeasible model with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

The process of identifying an IIS is one of addition and subtraction. The algorithm grows a set of constraints and bounds that are known to be part of an IIS, and also a set that are known not to be. You can track the progress of this process in the IIS log.

```
Computing Irreducible Inconsistent Subsystem (IIS)...

           Constraints          |            Bounds           |  Runtime
      Min       Max     Guess   |   Min       Max     Guess   |
-------------------------------------------------------------------------
```

```
      0     17509        -        0        4        -         0s
      0     12996        -        0        4        -         5s
      0     10398       20        0        0        -        10s
      0      9749       20        0        0        -        15s
      1      9584       20        0        0        -        20s
      4      9576       20        0        0        -        25s
      6      9406       20        0        0        -        30s
      8      8432       40        0        0        -        35s
     11      8422       40        0        0        -        40s
     15      8413       60        0        0        -        45s
     18      8241       60        0        0        -        50s
     21      7908       50        0        0        -        55s
     24      7735       50        0        0        -        60s
...
```

The first two columns show the minimum and maximum sizes of the set of constraints in the IIS. The fourth and fifth column give the same information for the set of variable bounds. The final column shows elapsed runtime.

The third and sixth columns in the IIS log provide guesses at the final size of the IIS. Computing an IIS can be quite time-consuming, and it is often useful to have a sense of how large the result will be. These guesses can be quite accurate in some cases, but unfortunately there are some models that can fool our heuristic. You should treat these as very rough estimates.

When the process completes, the algorithm outputs the size of the IIS (the number of constraints and bounds in the irreducible subsystem):

```
IIS computed: 102 constraints, 0 bounds
IIS runtime: 129.91 seconds
```

Note in this case that early guesses were in the neighborhood of 50 constraints, but the IIS contained 102.

If you terminate the process early, you will get a non-minimal infeasible subsystem instead:

```
Non-minimal IIS computed: 3179 constraints, 0 bounds
IIS runtime: 120.29 seconds
```

Gurobi Command-Line Tool

The Gurobi command-line tool allows you to perform simple commands without the overhead or complexity of an interactive interface. While the most basic usage of the command-line tool is quite straightforward, the tool has a number of uses that are perhaps less obvious. This section talks about its full capabilities.

To use this tool, you'll need to type commands into a command-line interface. Linux and Mac users can use a *Terminal* window. Windows users will need to open a *Command Prompt* (also known as a *Console* window or a `cmd` window). To launch one, hold down the *Start* and *R* keys simultaneously, and then type `cmd` into the *Run* box that appears.

The command to solve a model using the command-line tool is:

```
gurobi_cl [parameter=value]* modelfile
```

The Gurobi log file is printed to the screen as the model solves, and the command terminates when the solve is complete. Parameters are chosen from among the Gurobi parameters. The final argument is the name of a file that contains an optimization model, stored in MPS or LP format. You can learn more about using the command-line tool to solve models in this section.

The command-line tool can also be used to replay recordings of API calls. The command for this usage is:

```
gurobi_cl recordingfile
```

A recording file is a binary file generated by Gurobi with a `.grbr` extension. You can learn more about using the command-line tool to replay recordings in this section.

The command-line tool can also be used to check on the status of a Gurobi token server. The command is:

```
gurobi_cl --tokens
```

This command will show you whether the token server is currently serving tokens, and which users and machines are currently using tokens.

You can also type:

```
gurobi_cl --help
```

to get help on the use of the tool, or:

```
gurobi_cl --version
```

to get version information, or:

```
gurobi_cl --license
```

to get the location of the current Gurobi license file.

20.1 Solving a Model

The command-line tool provides an easy way to solve a model stored in a file. The model can be stored in several different formats, including MPS, REW, LP, and RLP, and the file can optionally be compressed using `gzip`, `bzip2`, or `7z`. See the File Format discussion for more information on accepted formats.

The most basic command-line command is the following:

```
gurobi_cl model.mps
```

This will read the model from the indicated file, optimize it, and display the Gurobi log file as the solve proceeds.

You can optionally include an arbitrary number of `parameter=value` commands before the name of the file. For example:

```
gurobi_cl Method=2 TimeLimit=100 model.mps
```

The full set of Gurobi parameters is described in the Parameter section.

Gurobi Compute Server users can add the `--server=` switch to specify a server. For example, the command:

```
gurobi_cl --server=server1 Method=2 TimeLimit=100 model.mps
```

would solve the model stored in file `model.mps` on machine `server1`, assuming it is running Gurobi Compute Server. If the Compute Server has an access password, use the `--password=` switch to specify it.

Gurobi Instant Cloud users can add the `--accessid=`, `--secretkey=`, and `--pool=` switches to run a model on a cloud instance. For example, the command:

```
gurobi_cl --accessid=0f5e0ace-f929-a919-82d5-02272b3b0e19 \
          --secretkey=8EDZOIf7T9avp0ZHef9Tsw --pool=mypool model.mps
```

would solve the model stored in file `model.mps` on cloud pool `mypool` using the provided access ID and secret key. If the pool isn't currently active, it will launch it first.

Writing Result Files

While it is often useful to simply solve a model and display the log, it is also common to want to review the resulting solution. You can use the ResultFile parameter to write the solution to a file:

```
gurobi_cl ResultFile=model.sol model.mps
```

The file name suffix determines the type of file written. Useful file formats for solution information are `.sol` (for solution vectors) and `.bas` (for simplex basis information). Again, you should consult the section on File Formats for a list of the supported formats

If you have an infeasible model, you may want to examine a corresponding Irreducible Inconsistent Subsystem (IIS) to identify the cause of the infeasibility. You can ask the command-line tool to write a `.ilp` format file. It will attempt to solve the model, and if the model is found to be infeasible, it will automatically compute an IIS and write it to the requested file name. An IIS is a subset of the constraints and variable bounds with the following properties:

- It is still infeasible, and
- If a single constraint or bound is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the smallest one; there may exist others with fewer constraints or bounds.

IIS results are returned in a number of attributes: IISConstr, IISLB, IISUB, IISSOS, IISQConstr, and IISGenConstr. Each indicates whether the corresponding model element is a member of the computed IIS.

Note that for models with general function constraints, piecewise-linear approximation of the constraints may cause unreliable IIS results.

The IIS log provides information about the progress of the algorithm, including a guess at the eventual IIS size.

If an IIS computation is interrupted before completion, Gurobi will return the smallest infeasible subsystem found to that point.

The IISConstrForce, IISLBForce, IISUBForce, IISSOSForce, IISQConstrForce, and IISGenConstrForce attributes allow you mark model elements to either include or exclude from the computed IIS. Setting the attribute to 1 forces the corresponding element into the IIS, setting it to 0 forces it out of the IIS, and setting it to -1 allows the algorithm to decide.

To give an example of when these attributes might be useful, consider the case where an initial model is known to be feasible, but it becomes infeasible after adding constraints or tightening bounds. If you are only interested in knowing which of the changes caused the infeasibility, you can force the unmodified bounds and constraints into the IIS. That allows the IIS algorithm to focus exclusively on the new constraints, which will often be substantially faster.

Note that setting any of the `Force` attributes to 0 may make the resulting subsystem feasible, which would then make it impossible to construct an IIS. Trying anyway will result in a `GRB_ERROR_IIS_NOT_INFEASIBLE` error. Similarly, setting this attribute to 1 may result in an IIS that is not irreducible. More precisely, the system would only be irreducible with respect to the model elements that have force values of -1 or 0.

Another use of `ResultFile` is to translate between file formats. For example, if you want to translate a model from MPS format to LP format, you could issue the following command:

```
gurobi_cl TimeLimit=0 ResultFile=model.lp model.mps
```

Gurobi can write compressed files directly, so this command would also work (assuming that `7zip` is installed on your machine):

```
gurobi_cl TimeLimit=0 ResultFile=model.lp.7z model.mps
```

The `ResultFile` parameter works differently from other parameters in the command-line interface. While a parameter normally takes a single value, you can actually specify multiple result files. For example, the following command:

```
gurobi_cl ResultFile=model.sol ResultFile=model.bas model.mps
```

will write two files.

Reading Input Files

You can use the InputFile parameter to read input files during the optimization. The most common input formats are `.bas` (a simplex basis), `.mst` (a MIP start), `.sol` (also a MIP start), `.hnt` (MIP hints), and `.ord` (a MIP priority order). For example, the following command:

```
gurobi_cl InputFile=model.bas model.mps
```

would start the optimization of the continuous model stored in file `model.mps` using the basis provided in file `model.bas`.

Reading input files is equivalent to setting the values of Gurobi attributes. A `.bas` file populates the VBasis and CBasis attributes, while a `.ord` file populates the BranchPriority attribute. A `.mst` or `.sol` file populates the Start attribute. A `.hnt` file populates the VarHintVal and VarHintPri attributes.

Again, you should consult the File Formats section for more information on supported file formats

20.2 Replaying Recording Files

If you've generated a recording of the Gurobi API calls made by your program, you may use the command-line tool to replay this recording.

Recordings are stored in files with `.grbr` extensions. To replay a recording from a file named `recording000.grbr` issue the following command:

```
gurobi_cl recording000.grbr
```

You should adjust the file name to match the recording you wish to replay.

You will know you have succeeded in replaying a recording, if you see lines similar to the following at the beginning of the command-line tool's output:

```
Replay* Replay of file 'recording000.grbr'
Replay* Recording captured Tue Sep 13 19:28:48 2023
Replay* Recording captured with Gurobi version 11.0.0 (linux64)
```

For information about recording API calls and replaying them, see the Recording API Calls chapter.

While the default goal of the Gurobi Optimizer is to find one proven optimal solution to your model, with a possible side-effect of finding other solutions along the way, the solver provides a number of parameters that allow you to change this behavior.

21.1 Finding Multiple Solutions

By default, the Gurobi MIP solver will try to find one proven optimal solution to your model. It will typically find multiple sub-optimal solutions along the way, which can be retrieved later (using the SolutionNumber parameter, and the Xn and PoolObjVal attributes). However, these solutions aren't produced in a systematic way. The set of solutions that are found depends on the exact path the solver takes through the MIP search. You could solve a MIP model once, obtaining a set of interesting sub-optimal solutions, and then solve the same problem again with different parameter settings, and find only one optimal solution or a different set of sub-optimal solutions.

If you'd like more control over how solutions are found and retained, the Gurobi Optimizer has a number of parameters available for this. The first and simplest one is PoolSolutions, which controls the size of the solution pool. Changing this parameter won't affect the number of solutions that are found - it simply determines how many of those are retained.

You can use the PoolSearchMode parameter to control the approach used to find multiple solutions. In its default setting (0), the MIP search simply aims to find one optimal solution. Setting the parameter to 1 causes the MIP search to expend additional effort to find more solutions, but in a non-systematic way, and with no guarantee on the quality of these solutions: you will get more solutions, but not necessarily the best solutions. Setting the parameter to 2 causes the MIP to do a systematic search for the `n` best solutions. For both non-default settings, the PoolSolutions parameter sets the target for the number of solutions to find.

If you are only interested in solutions that are within a certain gap of the best solution found, you can set the PoolGap parameter. Solutions that are not within the specified gap are discarded.

Obtaining an `OPTIMAL` optimization return status when using `PoolSearchMode=2` indicates that the MIP solver succeeded in finding the desired number of best solutions, or it proved that the model doesn't have that many distinct feasible solutions. If the solver terminated early (e.g., due to a time limit), you can use the PoolObjBound attribute to evaluate the quality of the solutions that were found. You can find more details on how to do this later in this section.

There are a few subtleties associated with finding multiple solutions that you should be aware of. For example, the notion of finding the n best solutions can be a bit ambiguous when you have a non-zero optimality tolerance. Also, it isn't obvious whether two solutions should be considered different when the model has continuous variables. We'll discuss these issues later in this section.

21.2 Retrieving Solutions

After optimization has completed, you can retrieve solutions from the solution pool using a few parameters and attributes. The SolCount attribute indicates how many solutions were retained by the MIP solver. The best solution can always be obtained through the X attribute. Sub-optimal solutions can be obtained by first setting the SolutionNumber parameter and then querying the Xn attribute to obtain the solution or the PoolObjVal attribute to obtain the objective value for the corresponding solution.

Solutions in the solution pool are ordered from best to worst. For example, to retrieve the worst solution kept by the MIP solver, you'd first query `SolCount` to determine how many solutions are available, then set the `SolutionNumber` parameter to `SolCount-1`, then query the `Xn` attribute.

The PoolObjBound attribute gives a bound on the objective value of undiscovered solutions (i.e., of solutions that aren't already in the solution pool). Further tree exploration won't find solutions having a better objective value than `PoolObjBound`.

The difference between this attribute and ObjBound is that the latter gives a bound on the objective for any solution, while `PoolObjBound` only refers to undiscovered solutions (which are not in the solution pool). `PoolObjBound` is always at least as tight as `ObjBound` and it is often strictly tighter than `ObjBound` if `PoolSearchMode=2`. If `PoolSearchMode=0` or `PoolSearchMode=1`, then `PoolObjBound` and `ObjBound` will always take the same value.

If the solver terminated early (e.g. due to reaching the time limit), you can still use the attribute `PoolObjBound` to get a count of how many of the n best solutions you found: any solutions having objective values that are at least as good as `PoolObjBound` are among the n best. This is illustrated in the examples on the next section.

21.3 Examples

Parameter usage

Let's continue with a few examples on how the parameters related to solution pools would be used. Imagine that you are solving a MIP model with an optimal (minimization) objective of 100. Further imagine that, using default settings, the MIP solver finds four solutions to this model with objectives 100, 110, 120, and 130.

If you set the PoolSolutions parameter to 3 and solve the model again, the MIP solver would return with 3 solutions in the solution pool (i.e., the SolCount attribute would have value 3, and the objective values would be 100, 110 and 120). If you instead set the PoolGap parameter to value `0.2`, the MIP solver would discard any solutions whose objective value is worse than 120 (which would also leave 3 solutions in the solution pool).

If you set the PoolSearchMode parameter to 1 and the PoolSolutions parameter to 10, the MIP solver would continue running after having found an optimal solution trying to find and store 10 solutions, but with no guarantee on the quality of the additional solutions.

If you set the PoolSearchMode parameter to 2 and the PoolSolutions parameter to 10, the MIP solver would attempt to find the 10 best solutions to the model. An `OPTIMAL` return status would indicate that either (i) it found the 10 best solutions, or (ii) it found all feasible solutions to the model, and there were fewer than 10. If you also set the PoolGap parameter to a value of 0.1, the MIP solver would try to find 10 solutions with objective no worse than 110. While this may appear equivalent to asking for 10 solutions and simply ignoring those with objective worse than 110, the

solve will typically complete significantly faster with this parameter set, since the solver does not have to expend effort looking for solutions beyond the requested gap.

Interpreting attribute information

Let's try to better understand the attributes related to solution pools. Consider again a minimization problem where the parameter settings `PoolSearchMode=2` and `PoolSolutions=10` have been used.

First, imagine that the solver terminated with an `OPTIMAL` return status. We look at several possible hypothetical values of some attributes:

- *Case 1*: `ObjVal=100`, `ObjBound=100`, `PoolObjBound=500`, and the objective value of the 10th solution in the pool is 500.

  The first solution in the pool is optimal (because `ObjVal` and `ObjBound` are equal), and the solver was able to find 10 solutions of value at most 500. Because `PoolObjBound=500`, we know that all solutions that the solver did not find have an objective value of at least 500. Since the last solution in the pool has value 500, the 10 solutions in the pool are definitely the 10 best solutions.

- *Case 2*: `ObjVal=100`, `ObjBound=100`, `PoolObjBound=+infinity`, `SolCount=7`, and the values of the 7th and 8th solutions in the pool are 350 and `+infinity`, respectively.

  The solver has found 7 solutions and has proven that no other feasible solution for the model exists. The best of these 7 solutions has objective 100, the worst of them has objective 350.

Now, imagine that the solver terminated early due to a time limit (return status `TIME_LIMIT`). Again, we look at several possible hypothetical values of some attributes:

- *Case 3:* `ObjVal=100`, `ObjBound=50`, `PoolObjBound=50`

  Since `ObjBound` $<$ `ObjVal`, the solver did not prove optimality of the incumbent solution (the first solution in the pool). There can be better solutions than the incumbent.

- *Case 4:* `ObjVal=100`, `ObjBound=100`, `PoolObjBound=100`, and the objective value of the 10th solution in the pool is 500.

  The first solution in the pool is optimal (because `ObjVal` and `ObjBound` are equal), and the solver was able to find 10 solutions with objective value at most 500. Because `PoolObjBound=100`, we know that all solutions that the solver did not find have an objective value of at least 100. Since the value of the last solution in the pool is 500, it could be the case that there exist 10 or more solutions with objective smaller than 500 (but greater than or equal to 100). In particular, solutions that are currently in the pool and have objective value greater than 100 might not be among the 10 best solutions.

- *Case 5:* `ObjVal=100`, `ObjBound=100`, `PoolObjBound=200`, and the objective values of the 4th, 5th and 10th solutions in the pool are 180, 220 and 500, respectively.

  The first solution in the pool is optimal (because `ObjVal` and `ObjBound` are equal), and the solver was able to find 10 solutions of value at most 500. Because `PoolObjBound=200`, we know that all solutions that the solver did not find have an objective value of at least 200. This means that the first 4 solutions in the pool (up to the solution with value 180) are

definitely the best four solutions that exist. The 5th solution with value 220 (and subsequent solutions in the pool) may be inferior to other undiscovered solutions.

21.4 Subtleties and Limitations

There are a few subtleties associated with finding multiple solutions that we'll cover now.

Continuous Variables

One subtlety arises when considering multiple solutions for models with continuous variables. Specifically, you may have two solutions that take identical values on the integer variables but where some continuous variables differ. By choosing different points on the line between these two solutions, you actually have an infinite number of choices for feasible solutions to the problem. This might be an issue, because the solution pool could be filled with solutions that only differ in the values of continuous variables but are otherwise equivalent, providing little interesting information. To avoid this issue, we define two solutions as being equivalent if

- in the presolved space, the integer variables and continuous variables participating in SOS and bilinear constraints assume the same values, or
- in the original model space, all variables assume the same values.

A solution will be discarded if it is equivalent to another solution that is already in the pool.

Optimality Gap

The interplay between the optimality gap (MIPGap or MIPGapAbs) and multiple solutions can be a bit subtle. When using the default `PoolSearchMode`, a non-zero optimality gap indicates that you are willing to allow the MIP solver to declare a solution optimal, even though the model may have other, better solutions. The claim the solver makes upon termination is that no other solution would improve the incumbent objective by more than the optimality gap. Terminating at this point is ultimately a pragmatic choice - we'd probably rather have the true best solution, but the cost of reducing the optimality gap to zero can often be prohibitive.

This pragmatic choice can produce a bit of confusion when finding multiple optimal solutions. Specifically, if you ask for the n best solutions, the optimality gap plays a similar role as it does in the default case, but the implications may be a bit harder to understand. Specifically, a non-zero optimality gap means that you are willing to allow the solver to declare that it has found the n best solutions, even though there may be solutions that are better than those that were returned. The claim in this case is that any solution not among the reported n best would improve on the objective for the worst among the n best by less than the optimality gap.

If you want to avoid this source of potential confusion, you should set the optimality gap to 0 when using `PoolSearchMode=2`.

Logging

The log for a MIP solve with `PoolSearchMode` set to a non-default value is different from the standard MIP log. You should consult the section Solution Pool and Multi-Scenario Logging for details.

Distributed MIP

One limitation that we should point out related to multiple solutions is that the distributed MIP solver has not been extended to support non-default `PoolSearchMode` settings. Distributed MIP will typically produce many more feasible solutions than non-distributed MIP, but there's no way to ask it to find the n best solutions.

Multiple Objectives

While typical optimization models have a single objective function, real-world optimization problems often have multiple, competing objectives. For example, in a production planning model, you may want to both maximize profits and minimize late orders, or in a workforce scheduling application, you may want to minimize the number of shifts that are short-staffed while also respecting worker's shift preferences.

The main challenge you face when working with multiple, competing objectives is deciding how to manage the trade-offs between them. Gurobi provides tools that simplify the task: Gurobi allows you to *blend* multiple objectives, to treat them *hierarchically*, or to combine the two approaches. In a blended approach, you optimize a weighted combination of the individual objectives. In a hierarchical or lexicographic approach, you set a priority for each objective, and optimize in priority order. When optimizing for one objective, you only consider solutions that would not degrade the objective values of higher-priority objectives. Gurobi allows you to enter and manage your objectives, to provide weights for a blended approach, and to set priorities for a hierarchical approach.

22.1 Specifying Multiple Objectives

Let us first discuss the interface for managing multiple objectives. An empty model starts with one objective function, which is initially just 0.0. We'll refer to this as the *primary* objective. You can modify the primary objective in two ways: you can set the Obj attribute, or you can use the `setObjective` method from your language API (e.g., Model.setObjective in Python). In contrast to models with a single objective, where the primary objective can be linear, quadratic, or piecewise-linear, all objectives must be linear for multi-objective models. In general, attributes and methods that aren't specific to multi-objective optimization will work with the primary objective function.

Every objective in a multi-objective model has the following settable attributes: ObjNCon with default value 0, ObjNPriority with default value 0, ObjNWeight with default value 1, ObjNRelTol with default value 0, ObjNAbsTol with default value 0, and ObjNName.

To provide additional objectives, use the `setObjectiveN` method from your language API (e.g. Model.setObjectiveN in Python). Objectives are numbered `0` through `NumObj-1`. The order of the objectives is arbitrary, but you must provide a unique index for each one (specified using the `index` argument to `setObjectiveN`). You can query the number of objectives in your model using the NumObj attribute. As noted above, all objectives, including the primary one, must be linear.

You can query and modify information about multiple objectives using the ObjNumber parameter, in conjunction with several model and variable attributes. For example, to retrieve the coefficient for variable `x` in objective 2, you'd set the `ObjNumber` parameter to 2, then query the ObjN attribute for `x`. Similarly, querying the ObjNName attribute after setting `ObjNumber` to 3 would give the name of objective 3.

We should note that there is one important exception to our statement above that the order of

objectives is arbitrary: objective 0 is treated as the primary objective. One consequence is that the original objective automatically becomes objective 0 when you add a second objective. Another is that querying the `ObjN` attribute is equivalent to querying the Obj attribute when `ObjNumber` is 0.

Note that a model has a single objective sense (controlled by the ModelSense attribute). This means that you can't maximize the first objective and minimize the second. However, you can achieve the same result with a simple trick. Each objective has a weight, and these weights are allowed to be negative. Minimizing an objective function is equivalent to maximizing the negation of that function.

You can change the number of objectives in your model as many times as you like (by modifying the NumObj attribute). When you increase the objective count, the newly added objectives and their associated attributes are set to 0. When you decrease the count, objectives beyond the new count are discarded. If you set the number of objectives to zero, the model becomes a pure feasibility problem.

We have extended the LP and MPS file formats, so writing a model with multiple objectives to a file will capture those objectives. Similarly, if you read a model file that contains multiple objectives, then NumObj and ObjN will capture the objectives stored in the file. See the file format section for details.

22.2 Working With Multiple Objectives

Of course, specifying a set of objectives is only the first step in solving a multi-objective optimization problem. The next step is to indicate how the objectives should be combined. As noted earlier, we support two approaches: blended and hierarchical.

Blended Objectives

A blending approach creates a single objective by taking a linear combination of your objectives. You provide a weight for each objective as an argument to `setObjectiveN`. Alternatively, you can use the ObjNWeight attribute, together with ObjNumber. The default weight for an objective is 1.0.

To give an example, if your model has two objectives, $1 + x + 2y$ and $y + 2z$, and if you give weights of -1 and 2 to them, respectively, then Gurobi would solve your model with a blended objective of $-1 \cdot (1 + x + 2y) + 2 \cdot (y + 2z) = -1 - x + 4z$.

You should avoid weights that are very large or very small. A very large weight (i.e., larger than 10^6) may lead to very large objective coefficients, which can cause numerical difficulties. A very small weight (i.e., smaller than $1e - 6$) may cause the contribution from that objective to the overall blended objective to be smaller than tolerances, which may lead to that objective being effectively ignored.

Hierarchical Objectives

A hierarchical or lexicographic approach assigns a priority to each objective, and optimizes for the objectives in decreasing priority order. During each of these optimization passes, it finds the best solution for the current objective, but only from among those that would not degrade the solution quality for higher-priority objectives. You provide the priority for each objective as an argument to `setObjectiveN`. Alternatively, you can use the ObjNPriority attribute. Priorities are integral, not continuous. Larger values indicate higher priorities. The default priority for an objective is 0.

To give an example, if your model has two objectives, with priorities 10 and 5, and objective weights 1.0 and -1.0. Assuming the optimal solution for the first objective has value 100, then the solver will find the solution that optimizes −1 times the second objective from among all solutions with objective 100 for the first objective.

Allowing Multiple-Objective Degradation

By default, our hierarchical approach won't allow later objectives to degrade earlier objectives, subject to the user-given ending gap conditions for the optimization problem. More precisely, the base value used to define what solutions are acceptable for lower priorities objectives – for a minimization problem – is computed as:

$$\text{base\_value} = \max\{bestsol, bestbound + |bestsol| * rgap, bestbound + agap\},$$

where `bestsol` is the value of the best incumbent solution, `bestbound` is the value of the best proven lower bound for the problem, `rgap` is the relative MIP gap, and `agap` is the absolute MIP gap, and the set of feasible solutions for the next objective will consider solutions whose objective value is at most that value.

This behavior can be relaxed for MIPs through a pair of tolerances: a relative and an absolute tolerance. These are provided as arguments to `setObjectiveN`, or they can be set using attributes ObjNRelTol and ObjNAbsTol. By setting one of these for a particular objective, you can indicate that later objectives are allowed to degrade this objective by the specified relative or absolute amount, respectively. In our earlier example, if the optimal value for the first objective is 100, and if we set `ObjNAbsTol` for this objective to 20, then the second optimization pass would find the best solution for the second objective from among all solutions with objective 120 or better for the first objective. Note that if you modify both tolerances, later optimizations would use the looser of the two values (i.e., the one that allows the larger degradation).

Objective degradations are handled differently for multi-objective LP models. For LP models, solution quality for higher-priority objectives is maintained by fixing some variables to their values in previous optimal solutions. These fixings are decided using variable reduced costs. The value of the `ObjNAbsTol` parameter indicates the amount by which a fixed variable's reduced cost is allowed to violate dual feasibility, whereas the `ObjNRelTol` parameter is simply ignored. If you want the MIP behavior, where the degradation is controlled more directly, you can add a dummy binary variable to the model, thus transforming it into a MIP. Solving the resulting multi-objective MIP will be much more time consuming than solving the original multi-objective LP.

Combining Blended and Hierarchical Objectives

Every objective in a multi-objective model has both a weight and a priority, which allows you to seamlessly combine blended and hierarchical approaches. To understand how this works, we should first provide more detail on how hierarchical objectives are handled.

When you specify a different priority for each of n objectives, the solver performs n separate optimization passes. In each pass, in decreasing priority order, it optimizes for the current objective multiplied by its ObjNWeight attribute, while imposing constraints that ensure that the quality of higher-priority objectives isn't degraded by more than the specified tolerances.

If you give the same priority to multiple objectives, then they will be handled in the same optimization pass, resulting in fewer than n total passes for n objectives. More precisely, one optimization pass is performed per distinct priority value, in order of decreasing priority, and all

objectives with the same priority are blended together, using the weights for those objectives. This gives you quite a bit of flexibility when combining the blended and hierarchical approaches.

One subtle point when blending multiple objectives within a single level in a hierarchical approach relates to the handling of degradations from lower-priority levels. The objective degradation allowed after a blended optimization pass is the maximum absolute and relative degradation allowed by each of the participating objectives. For example, if we have three objectives with `ObjNPriority` equal to $\{2, 2, 1\}$, and `ObjNRelTol` equal to $\{0.10, 0.05, 0.00\}$ and `ObjNAbsTol` equal to $\{0, 1, 2\}$, and if the best solution for the first priority objective is 10, then the allowed degradation for the first priority objective in subsequent passes is $\max\{10 \cdot 0.10, 10 \cdot 0.05, 0, 1\} = 1$.

Querying multi-objective results

Once you have found one or more solutions to your multi-objective model, you can query the achieved objective value for each objective on each solution. Specifically, if you set the ObjNumber parameter to choose an objective, and the SolutionNumber parameter to choose a solution, then the ObjNVal attribute will give the value of the chosen objective on the chosen solution. This is illustrated in the following Python example:

```
# Read and solve a model with multiple objectives
m = read('input.mps')
m.optimize()

# get the set of variables
x = m.getVars()

# Ensure status is optimal
assert m.Status == GRB.Status.OPTIMAL

# Query number of multiple objectives, and number of solutions
nSolutions  = m.SolCount
nObjectives = m.NumObj
print('Problem has', nObjectives, 'objectives')
print('Gurobi found', nSolutions, 'solutions')

# For each solution, print value of first three variables, and
# value for each objective function
solutions = []
for s in range(nSolutions):
  # Set which solution we will query from now on
  m.params.SolutionNumber = s

  # Print objective value of this solution in each objective
  print('Solution', s, ':', end='')
  for o in range(nObjectives):
    # Set which objective we will query
    m.params.ObjNumber = o
    # Query the o-th objective value
    print(' ',m.ObjNVal, end='')

  # print first three variables in the solution
  n = min(len(x),3)
  for j in range(n):
    print(x[j].VarName, x[j].Xn, end='')
  print('')
```

```
# query the full vector of the o-th solution
solutions.append(m.getAttr('Xn',x))
```

22.3 Additional Details

Multi-Objective Environments

As we progress from higher-priority objectives to lower-priority objectives in a hierarchical multi-objective model, we won't necessarily solve each pass to exact optimality. By default, termination criteria (e.g. TimeLimit, SolutionLimit, etc.) are controlled by the parameters defined in the model environment. However, we provide a feature called *multi-objective environments* that allows you to create a Gurobi environment for each optimization pass and set parameters on those environments. Those settings will only affect the corresponding pass of the multi-objective optimization. Thus, for example, if the TimeLimit parameter for the model is 100, but you use a multi-objective environment to set the parameter to 10 for a particular optimization pass, then the multi-objective optimization will spend at most 10 seconds on that particular pass (and at most 100 seconds in total).

To create a multi-objective environment for a particular optimization pass, use the `getMultiobjEnv` method from your language API (e.g. Model.getMultiobjEnv in Python). The `index` argument gives the index of the optimization pass that you want to control.

Please note that optimization passes and objectives are not quite synonymous, due the possibility of blending objectives at the same priority level. Multi-objective environment 0 is always tied to the highest priority (possibly blended) objective, while multi-objective environment 1 is always tied to the second highest priority objective (if any). For details on how multiple objectives with the same priority are treated, please refer to the hierarchical objectives section.

Once you create multi-objective environments, they will be used for every subsequent multi-objective optimization on that model. Use the `discardMultiobjEnvs` method from your language API (e.g. Model.discardMultiobjEnvs in Python) to revert back to default multi-objective optimization behavior.

Please note that parameter values are copied from the model when the multi-objective environment is created. Therefore, changes to parameter values on the model have no effect on multi-objective environments that have already been created. This is a frequent source of confusion.

Other Details

We haven't attempted to generalize the notions of dual solutions or simplex bases for continuous multi-objective models, so you can't query attributes such as Pi, RC, VBasis, or CBasis for multi-objective solutions. Because of this, we've concluded that the most consistent result to return for attribute IsMIP is 1. Note, however, that several MIP-specific attributes such as ObjBound, ObjBoundC and MIPGap don't make sense for multi-objective models and are also not available.

Gurobi will only solve multi-objective models with strictly linear objectives. If the primary objective is quadratic or piecewise linear, the solve call will return an error.

When solving a continuous multi-objective model using a hierarchical approach, you have a choice of which optimization algorithm to use for the different passes (primal simplex, dual simplex, or barrier). The first pass will always use the algorithm specified in the Method parameter. The

algorithm for subsequent passes is controlled by the MultiObjMethod parameter. This parameter has no effect for multi-objective MIP models. Note you can get finer-grained control over the algorithm choice using our multi-objective environment feature, by setting the Method parameter for individual objectives.

For the hierarchical approach, Gurobi will perform a conservative presolve step at the beginning of the multi-objective optimization, and a more aggressive presolve step at the beginning of each pass (assuming presolve hasn't been turned off). You can optionally perform a more aggressive presolve step at the beginning of the multi-objective optimization by setting parameter MultiObjPre to value 2. This can help performance, but it makes a few simplifying assumptions that could lead to small degradations in the values achieved for lower-priority objectives.

The log file when using a hierarchical approach will show optimization progress for each pass in the process. You'll see log lines that look like this:

```
Multi-objectives: optimize objective 1 (Obj1Name) ...
...
Multi-objectives: optimize objective 2 (weighted) ...
...
```

For further details, please see section Multi-Objective Logging.

Callbacks are available for multi-objective optimization, but they are a bit more involved than those for single-objective optimization. When you are solving for a specific objective (either in one phase of a hierarchical optimization or when solving a blended objective), you will receive callbacks from the algorithm that solves that model: MIP callbacks if the model is a MIP, and simplex or barrier callbacks if the model is continuous. For a hierarchical objective, you will also get a `MULTIOBJ` callback at the end of each phase that allows you to query the current solution, the number of solutions found, and the number of objectives that have been solved for at that point. Refer to the Callback discussion for further details.

Multiple Scenarios

When solving an optimization model, it is often useful to understand the sensitivity of the computed solution to changes in the inputs. How would profits be affected if the price of a particular raw material increased significantly? Would I still be able to satisfy customer orders if one of my machines broke down? The most general form of this problem would fall into the domain of stochastic or robust optimization, but those fields bring significant complexity with them. The Gurobi Optimizer includes scenario analysis features that have a much more modest goal: to allow the user to specify a set of scenarios, and to compute optimal solutions for all of these scenarios as quickly as possible. These solutions often provide significant insight into how the solution would change as inputs vary.

23.1 Definition of a Multi-Scenario Model

Before diving into the details of working with multiple scenarios, we first need to explain exactly what we mean by the term. Let us start by claiming that it only makes sense to consider a set of models as being different scenarios for the same underlying model if they have a lot in common. They should definitely share the same set of variables. They should also have similar sets of constraints and similar objectives. In our approach, scenarios are described as a set of changes from a single `base` model. More specifically, scenarios can only modify model features that are present in the base model. We should add that other modifications, including addition and deletion of variables or constraints, can be achieved through the clever use of various tricks. For now, though, it is best to think of scenarios as being small variations on the same base.

What variations do we allow from this base model? Scenarios can differ in the following attributes:

- Linear objective function coefficients.
- Variable lower and upper bounds.
- Constraint right-hand side values.

A single scenario can have multiple changes from the base, so for example you could change an objective coefficient, two variable bounds, and a right-hand side value in the same scenario.

After you have defined a set of scenarios (the specific mechanics for doing so will be described shortly), the next step is to find solutions for all of the scenarios. A single call to the standard Gurobi `optimize` method is all that is needed. This will of course be much more expensive than finding an optimal solution for a single model, but our goal is for it to be faster and more convenient than formulating and solving separate models for each scenario.

23.2 Specifying Multiple Scenarios

Your first step in building a multi-scenario model is to modify the NumScenarios attribute to indicate how many scenarios you would like to consider. Once you have changed this attribute, you

can describe the different scenarios by changing various scenario-related attributes (listed below). When you later call `optimize` on a multi-scenario model (a model where `NumScenarios` is greater than 0), the solver will try to find optimal solutions for all specified scenarios. Note that it will *not* try to find a solution for the base model.

Variations in the different scenarios are expressed through a set of four attributes:

- ScenNObj
- ScenNLB
- ScenNUB
- ScenNRHS

The first three are variable attributes, and the last is a linear constraint attribute. You can give each scenario a name through the ScenNName attribute (a model attribute).

You use the ScenarioNumber parameter to modify scenario attributes for a specific scenario. Scenarios are numbered `0` through `NumScenarios-1`. To give an example, to create a model where binary variable x is fixed to `0` and `1` in two scenarios, you would:

- Set the `NumScenarios` attribute to 2, to indicate that your model has two scenarios.
- Set the `ScenarioNumber` parameter to 0, to indicate that you would first like to modify scenario attributes for scenario 0.
- Set the `ScenNUB` attribute for variable x to 0 (to fix the binary variable to zero in this scenario).
- Set the `ScenarioNumber` parameter to 1, to move on to scenario 1.
- Set the `ScenNLB` attribute for variable x to 1 (to fix the binary variable to one in this scenario).

You query scenario attributes in a similar manner: set the `ScenarioNumber` parameter to choose the scenario you would like to query, and then use the appropriate attribute query routine to obtain the desired attribute values (consult our Attribute Examples for examples).

Note that unmodified scenario attributes take a special value of `GRB.UNDEFINED`. If you modified a scenario attribute and would like to revert that modification, you can set the attribute back to `GRB.UNDEFINED`.

You can change the number of scenarios in your model as many times as you like (by modifying the NumScenarios attribute). When you increase the count, new empty scenarios are created (an empty scenario is a scenario with no changes from the base model). When you decrease the count, existing scenarios are discarded. When you set the count to zero, the model is no longer treated as a multi-scenario model.

We have extended the LP and MPS file formats, so writing a model with multiple scenarios to a file will capture those scenarios. Similarly, if you read a model file that contains multiple scenarios, then NumScenarios and the various scenario attributes will capture the scenarios stored in the file. See the file format section for details.

23.3 Logging

When solving a multi-scenario model, logging is somewhat different from standard MIP logging. You should consult this section for details.

23.4 Retrieving Solutions for Multiple Scenarios

Your first step in retrieving the solutions computed by an `optimize` call on a multi-scenario model is to query the Status attribute. A status of `OPTIMAL` indicates that optimal solutions were found (subject to tolerances) for all scenarios that have feasible solutions, and that the other scenarios were determined to be infeasible. If all scenarios were found to be infeasible, the status will be `INFEASIBLE`. If any scenario is unbounded, the status will be `UNBOUNDED`. An early termination status code (e.g., `TIME_LIMIT`) indicates that the outcomes may vary across the different scenarios, and you will have to look at individual scenarios for more information.

Results for individual scenarios can be found in three attributes:

- ScenNObjVal: The objective value for the solution for scenario number n.
- ScenNObjBound: The best known bound on the optimal objective value for scenario number n.
- ScenNX: The solution vector for scenario number n.

The `ScenNObjVal` and `ScenNObjBound` attributes are model attributes, while `ScenNX` is a variable attribute. Again, use the ScenarioNumber parameter to select the scenario you'd like to query.

If your `optimize` call terminated early, you should use the ScenNObjBound attribute to interpret the results. This attribute provides a bound on the optimal objective value for the selected scenario (much like ObjBound provides a bound for a single model). For example, if `ScenNObjVal` is 100 for a scenario and `ScenNObjBound` is 90 (assuming minimization), then you have a solution with a 10% optimality gap for that scenario.

You can also query the ObjVal and ObjBound attributes for a multi-scenario model. The former gives the best objective value for any solution found in any scenario. The latter provides a lower bound on the objective value for any solution that was not found.

Note that `ScenNObjVal` and `ScenNObjBound` are computed using the objective function for the corresponding scenario, which you may have changed from the base model.

If you query the `ScenNX` attribute and no feasible solution has been found for that scenario, you will get a `DATA_NOT_AVAILABLE` error.

23.5 Tips and Tricks

Through clever use of the features provided in the multi-scenario interface, it is actually possible to do a lot more than it may first appear.

Adding or Deleting Variables or Constraints

The multi-scenario interface provides no way to add or remove variables or constraints in a scenario, but the same effect can be achieved by changing variable bounds and constraint right-hand side values. To remove a variable in a scenario, simply change its lower and upper bounds to zero. To add a variable, set its bounds to zero in the base model and change them to their true values in the scenario. To remove a less-than constraint, change the right-hand-value in the scenario to `GRB.INFINITY`. To add one, set its right-hand side to `GRB.INFINITY` in the base model and change it to its true value in the scenario.

Changing the sense of a constraint can also be done using similar tricks. For example, you can transform an equality constraint in the base model into an inequality in a scenario by splitting the

equality into a pair of inequalities. The right-hand side values for both inequalities in the base model would be equal to the true value in the equality. The right-hand side value on one of the two inequalities can then be relaxed to `GRB.INFINITY` in the scenario.

You can also change the type of a variable. For example, to transform an integer variable in the base model into a continuous variable in a scenario, you can add both variables in the base model, along with a split equality constraint that sets them equal to each other. That equality constraint could then be relaxed in the scenario (using the techniques just described).

This isn't meant to be an exhaustive list of all of the ways that you can use supported multi-scenario features to achieve seemingly unsupported outcomes. The set of building blocks that we have provided can be assembled in a variety of different ways.

If all scenarios in your multi-scenario model are infeasible, your `optimize` call will produce an `INFEASIBLE` (or `INF_OR_UNBD`) status code. While you can't compute an IIS on a multi-scenario model, you can extract individual scenarios as Gurobi model objects using the `singleScenarioModel` method (see below) and then compute an IIS on each scenario individually.

Solving The Base Model

As noted earlier, an `optimize` call on a multi-scenario model will not solve the base model. If you'd like to solve that model too, include an empty scenario among your scenarios.

Extracting One Scenario

If you'd like to extract one scenario from a multi-scenario model, you can use the `singleScenarioModel` method (in C, C++, Java, .NET, and Python).

Performance Considerations

While it may appear to be important to minimize the number of scenarios in your model, note that some scenarios are trivial to solve and thus have no impact on overall solution cost. The main thing to keep in mind is that, if (1) the solution for one scenario is feasible for another scenario, and (2) the bounds and right-hand side values for the first scenario are never tighter than the bounds for the other scenario, then the optimal solution for the first scenario is also optimal for the other scenario. This means that some scenarios won't increase solution cost significantly.

23.6 Limitations and Additional Considerations

We should note a few additional considerations for solving multi-scenario models.

Nearly any model can serve as the base model in a multi-scenario model. The base model can be continuous or discrete, and it can contain quadratic constraints, general constraints, SOS constraints, semi-continuous variables, etc. One important exception is multiple objectives; a multi-scenario model can only have a single objective.

The IsMIP attribute will always be 1 for a multi-scenario model, even if the model is otherwise continuous.

Multi-scenario models can have lazy constraints, specified either through a user callback or through the Lazy attribute. Those that are provided through a callback must be valid across all scenarios. Those that are specified through the `Lazy` attribute can have different right-hand side values in different scenarios.

You can also add your own cutting planes to multi-scenario models through a user callback. Cuts must be valid across all scenarios.

Batch Optimization

Batch optimization is a feature available with the Gurobi Cluster Manager. It allows a client program to build an optimization model, submit it as a batch request to a Compute Server cluster (through a Cluster Manager), and later check on the status of the request and retrieve the solution. Once a batch is submitted to the Cluster Manager, it is identified through a unique BatchID. The client program (or any other program) can use this ID to query the BatchStatus of the batch (submitted, completed, etc.). Once the batch has completed and a solution is available, the client can retrieve that solution as a JSON string.

This section explains the steps required to perform the various tasks listed above. We'll use the `batchmode.py` Python example, which is included with the distribution, to illustrate these steps.

24.1 Setting Up a Batch Environment

Recall that the first step in building an optimization model is to create a Gurobi environment. An environment provides a number of configuration options; among them is an option to indicate where the model should be solved. You can solve a model locally, on a Compute Server, or using a Gurobi Instant Cloud server. If you have a Cluster Manager installed, you also have the option of using batch optimization.

To use batch optimization, you should configure your environment as if you will be using a Compute Server through a Cluster Manager. You'll need to set the CSManager parameter to point to your Cluster Manager, and provide a valid UserName and ServerPassword. The difference is that you will also need to set the CSBatchMode parameter to 1. This will cause the client to build the model locally, and only submit it to the server once a call to the `optimizeBatch` method is made. This is in contrast to a standard Compute Server job, where the connection to the server is established immediately and the model is actually built on the server.

The following shows how you might set up your environment for batch optimization (in Python):

```
env = gp.Env(empty=True)
env.setParam("LogFile", "batchmode.log")
env.setParam("CSManager", "http://localhost:61080")
env.setParam("UserName", "gurobi")
env.setParam("ServerPassword", "pass")
env.setParam("CSBatchMode", 1)
```

Note that you can also use CSAPIAccessID and CSAPISecret (instead of UserName and ServerPassword) to connect to a Cluster Manager.

24.2 Tagging Variables or Constraints

Batch optimization separates the process of building a model from the process of retrieving and acting on its solution. For example, you can build your model on one machine, submit a batch

request, and then use the resulting BatchID to retrieve the solution from a completely different machine.

Of course, disconnecting a model from its solution introduces a mapping problem: the process that retrieves the solution needs to know how to map the elements of the solution back to the corresponding elements of the model. This is done through *tags*. When a model is built, the user associates unique strings with the variables and constraints of interest in the model. Solution values are then associated with these strings. If the user doesn't provide a tag for a model element, no solution information is stored or returned for that element.

You can tag variables (using the VTag attribute), linear constraints (using the CTag attribute), and quadratic constraints (using the QCTag attribute). We should point out that solutions to mixed-integer models don't contain any constraint information, so constraint tags have no effect for such models.

For details on the information that is available in the solution in different situations, please refer to the JSON solution format section.

Here's a simple example that tags the first 10 variables in a model:

```
# Define tags for some variables in order to access their values later
for count, v in enumerate(model.getVars()):
    v.VTag = f"Variable{count}"
    if count >= 10:
        break
```

24.3 Submitting a Batch Optimization Request

Once you have built your model and tagged the elements of interest, you are ready to submit your batch request. This is done by invoking the `optimizeBatch` method (e.g., optimizeBatch in Python). This method returns a BatchID string which is used for later queries. Here's a simple example:

```
# Submit batch request
batchID = model.optimizeBatch()
```

24.4 Interacting with Batch Requests

You can use a BatchID string to ask the Cluster Manager for more information about the corresponding batch. Specifically, you can query the BatchStatus for that batch, and if the batch is complete you can retrieve the computed solution as a JSON string.

Your first step in using a `BatchID` to gather more information is to create a Gurobi environment that enables you to connect to your Cluster Manager. This is done in this line of our Python example:

```
with setupbatchenv().start() as env, gp.Batch(batchID, env) as batch:
```

The `setupbatchenv` method creates an environment with the CSManager, UserName, and ServerPassword parameters set to appropriate values.

With this environment and our `BatchID`, we can now create a Batch object (by calling the `Batch` constructor in the above code segment) that holds information about the batch.

That Batch object can be used to query the BatchStatus:

```
# Setup and start environment, create local Batch handle object
with setupbatchenv().start() as env, gp.Batch(batchID, env) as batch:
    starttime = time.time()
    while batch.BatchStatus == GRB.BATCH_SUBMITTED:
        # Abort this batch if it is taking too long
        curtime = time.time()
        if curtime - starttime > maxwaittime:
            batch.abort()
            break

        # Wait for two seconds
        time.sleep(2)

        # Update the resident attribute cache of the Batch object with the
        # latest values from the cluster manager.
        batch.update()

        # If the batch failed, we retry it
        if batch.BatchStatus == GRB.BATCH_FAILED:
            batch.retry()
```

It can also be used to perform various operations on the batch, including aborting or retrying the batch.

Once a batch has been completed, you can query the solution and all related attributes for tagged elements in the model by retrieving the associated JSON solution string (or by saving it into a file):

```
print("JSON solution:")
# Get JSON solution as string, create dict from it
sol = json.loads(batch.getJSONSolution())
```

By default, the Cluster Manager will keep the solution for the model and other information for a while (the exact retention policy is set by the Cluster Manager). You can ask the Cluster Manager to discard information for a batch by explicitly calling the `discard` method:

```
# Remove batch request from manager
batch.discard()
```

No further queries on that batch are possible after this has been done.

24.5 Interpreting the JSON Solution

Once you have retrieved a JSON solution string, you can use a JSON parser to retrieve solution information for individual variables and constraints. This parser isn't included in the Gurobi library. Rather, programming languages have libraries for doing this. The appropriate package in Python is (not surprisingly) called `json`. The following provides a simple example of how this library can be used to parse a JSON solution string and extract a few pieces of solution information:

```
# Get JSON solution as string, create dict from it
sol = json.loads(batch.getJSONSolution())

# Pretty printing the general solution information
print(json.dumps(sol["SolutionInfo"], indent=4))
```

24.6 A Complete Example

```
#!/usr/bin/env python3.11

# Copyright 2023, Gurobi Optimization, LLC

# This example reads a MIP model from a file, solves it in batch mode,
# and prints the JSON solution string.
#
# You will need a Compute Server license for this example to work.

import sys
import time
import json
import gurobipy as gp
from gurobipy import GRB


# Set up the environment for batch mode optimization.
#
# The function creates an empty environment, sets all necessary parameters,
# and returns the ready-to-be-started Env object to caller.   It is the
# caller's responsibility to dispose of this environment when it's no
# longer needed.
def setupbatchenv():
    env = gp.Env(empty=True)
    env.setParam("LogFile", "batchmode.log")
    env.setParam("CSManager", "http://localhost:61080")
    env.setParam("UserName", "gurobi")
    env.setParam("ServerPassword", "pass")
    env.setParam("CSBatchMode", 1)

    # No network communication happened up to this point.  This will happen
    # once the caller invokes the start() method of the returned Env object.

    return env


# Print batch job error information, if any
def printbatcherrorinfo(batch):
    if batch is None or batch.BatchErrorCode == 0:
        return

    print(
        f"Batch ID {batch.BatchID}: Error code {batch.BatchErrorCode} ({batch.BatchErrorMessage}
    )


# Create a batch request for given problem file
def newbatchrequest(filename):
    # Start environment, create Model object from file
    #
    # By using the context handlers for env and model, it is ensured that
    # model.dispose() and env.dispose() are called automatically
    with setupbatchenv().start() as env, gp.read(filename, env=env) as model:
```

```
        # Set some parameters
        model.Params.MIPGap = 0.01
        model.Params.JSONSolDetail = 1

        # Define tags for some variables in order to access their values later
        for count, v in enumerate(model.getVars()):
            v.VTag = f"Variable{count}"
            if count >= 10:
                break

        # Submit batch request
        batchID = model.optimizeBatch()

    return batchID


# Wait for the final status of the batch.
# Initially the status of a batch is "submitted"; the status will change
# once the batch has been processed (by a compute server).
def waitforfinalstatus(batchID):
    # Wait no longer than one hour
    maxwaittime = 3600

    # Setup and start environment, create local Batch handle object
    with setupbatchenv().start() as env, gp.Batch(batchID, env) as batch:
        starttime = time.time()
        while batch.BatchStatus == GRB.BATCH_SUBMITTED:
            # Abort this batch if it is taking too long
            curtime = time.time()
            if curtime - starttime > maxwaittime:
                batch.abort()
                break

            # Wait for two seconds
            time.sleep(2)

            # Update the resident attribute cache of the Batch object with the
            # latest values from the cluster manager.
            batch.update()

            # If the batch failed, we retry it
            if batch.BatchStatus == GRB.BATCH_FAILED:
                batch.retry()

        # Print information about error status of the job that processed the batch
        printbatcherrorinfo(batch)


def printfinalreport(batchID):
    # Setup and start environment, create local Batch handle object
    with setupbatchenv().start() as env, gp.Batch(batchID, env) as batch:
        if batch.BatchStatus == GRB.BATCH_CREATED:
            print("Batch status is 'CREATED'")
        elif batch.BatchStatus == GRB.BATCH_SUBMITTED:
            print("Batch is 'SUBMITTED")
        elif batch.BatchStatus == GRB.BATCH_ABORTED:
```

```
            print("Batch is 'ABORTED'")
        elif batch.BatchStatus == GRB.BATCH_FAILED:
            print("Batch is 'FAILED'")
        elif batch.BatchStatus == GRB.BATCH_COMPLETED:
            print("Batch is 'COMPLETED'")
            print("JSON solution:")
            # Get JSON solution as string, create dict from it
            sol = json.loads(batch.getJSONSolution())

            # Pretty printing the general solution information
            print(json.dumps(sol["SolutionInfo"], indent=4))

            # Write the full JSON solution string to a file
            batch.writeJSONSolution("batch-sol.json.gz")
        else:
            # Should not happen
            print("Batch has unknown BatchStatus")

        printbatcherrorinfo(batch)


# Instruct the cluster manager to discard all data relating to this BatchID
def batchdiscard(batchID):
    # Setup and start environment, create local Batch handle object
    with setupbatchenv().start() as env, gp.Batch(batchID, env) as batch:
        # Remove batch request from manager
        batch.discard()


# Solve a given model using batch optimization
if __name__ == "__main__":
    # Ensure we have an input file
    if len(sys.argv) < 2:
        print(f"Usage: {sys.argv[0]} filename")
        sys.exit(0)

    # Submit new batch request
    batchID = newbatchrequest(sys.argv[1])

    # Wait for final status
    waitforfinalstatus(batchID)

    # Report final status info
    printfinalreport(batchID)

    # Remove batch request from manager
    batchdiscard(batchID)

    print("Batch optimization OK")
```

24.7 Limitations

It is currently not possible to optimize models with Multiple Objectives via the Batch Optimization feature. The Concurrent Optimizer and the Parameter Tuning Tool are also not available via the Batch Optimization feature.

Recording API Calls

The Gurobi Optimizer provides the option to record the set of Gurobi commands issued by your program and store them to a file. The commands can be played back later using the Gurobi Command-Line Tool. If you replay the commands on a machine with the same specs (operating system, core count, and instruction set) as the machine where you created the recording, your Gurobi calls will take the exact same computational paths that they took when you ran your original program.

Recording can be useful in a number of situations.

- If you want to understand how much time is being spent in Gurobi routines, the replay will show you the total time spent in Gurobi API routines, and the total time spent in Gurobi algorithms.
- If you want to check for leaks of Gurobi data, the replay will show you how many Gurobi models and environments were never freed by your program.
- If you run into a question or an issue and you would like to get help from Gurobi, your recording will allow Gurobi technical support to reproduce the exact results that you are seeing without requiring you to send your entire application.

Recording is useful for testing across deployment scenarios. In particular, you can do the following:

- A recording made on a Compute Server can be replayed on a Compute Server or locally. By default, the recording will use `localhost:61000`; this can be overridden by setting the `GRB_COMPUTESERVER` environment variable. If `GRB_COMPUTESERVER` is set to an empty string, the replay will occur locally.
- A Cluster Manager recording can be replayed via a Cluster Manager, on a Compute Server, or locally. `GRB_COMPUTESERVER` has priority over `GRB_CSMANAGER`. If `GRB_CSMANAGER` is set to `""`, the replay will run locally or on the specified Compute Server.
- A Gurobi Instant Cloud recording can be replayed on the cloud, on a Compute Server, or locally. Setting `GRB_CLOUDACCESSID` and `GRB_CLOUDSECRETKEY` will run the replay on the cloud. `GRB_COMPUTESERVER` has priority over `GRB_CLOUDACCESSID` (and `GRB_CLOUDSECRETKEY`). If `GRB_CLOUDACCESSID` is set to `""`, the replay will run locally or on the specified Compute Server.
- A token server recording can be replayed. The recording tries to use a token server at `localhost` (with the default port 41954). This can be overridden by setting the `GRB_TOKENSERVER` environment variable. If `GRB_TOKENSERVER` is set to an empty string, the replay will be done using a local license.

- A recording that includes the `WorkerPool` parameter (used in the distributed MIP and distributed concurrent algorithms) can only be done by setting the `GRB_WORKERPOOL` environment variable.

For recordings of optimization with Compute Server or Instant Cloud, the recording will query the number of processors and cores from the remote worker and store this information in the recording file. When replaying a recording file, Gurobi uses these stored values.

25.1 Recording

To enable recording, you simply need to set the Record parameter to 1 as soon as you create your Gurobi environment. The easiest way to do this is with a `gurobi.env` file. This file should contain the following line:

```
Record 1
```

If you put this file in the same directory as your application, Gurobi will pick up the setting when your application makes its first Gurobi call. You can also set this parameter through the standard parameter modification routines in your program.

Once this parameter is set, you should see the following in your log:

```
*** Start recording in file recording000.grbr
```

If your application creates more than one Gurobi environment, you may see more than one of these messages. Each will write to a different file:

```
*** Start recording in file recording001.grbr
```

As your program runs, Gurobi will write the commands and data that are passed into Gurobi routines to these files. Recording continues until you free your Gurobi environment (or until your program ends). When you free the environment, if Gurobi logging is enabled you will see the following message:

```
*** Recording complete - close file recording000.grbr
```

At this point, you have a recording file that is ready for later replay.

25.2 Replay

To replay a Gurobi recording, you issue the following command:

```
> gurobi_cl recording000.grbr
```

You should adjust the file name to match the file you wish to replay. If your program generated multiple recording files, you will need to replay each one separately.

When the replay starts, the first output you will see will look like this:

```
Replay* Replay of file 'recording000.grbr'
Replay* Recording captured Tue Sep 13 19:28:48 2023
Replay* Recording captured with Gurobi version 11.0.0 (linux64)
```

After this output, the replay will start executing the commands issued by your program...

```
*Replay* Load new Gurobi environment
*Replay* Create new Gurobi model (0 rows, 0 cols)
*Replay* Update Gurobi model
*Replay* Change objective sense to -1
*Replay* Add 3 new variables
```

This continues until the recording file ends. At that point, the replay will print out a final runtime accounting...

```
*Replay* Replay complete

*Replay* Gurobi API routine runtime: 0.05s
*Replay* Gurobi solve routine runtime: 2.31s
```

If your program leaked any Gurobi models or environments, you may also see that in the output:

```
*Replay* Models leaked: 2
*Replay* Environments leaked: 1
```

25.3 Limitations

Recording works with most programs that call Gurobi. There are a few Gurobi features that aren't supported, though:

- Recording won't capture calls to the Gurobi tuning tool.
- Recording won't capture data passed into control callbacks. In other words, you can't record a program that adds lazy constraints, user cuts, or solutions through callbacks.
- Recording is not yet compatible with Batch Mode.

Concurrent optimization is a simple approach for exploiting multiple processors. It starts multiple, independent solves on a model, using different strategies for each. Optimization terminates when the first one completes. By pursuing multiple different strategies simultaneously, the concurrent optimizer can often obtain a solution faster than it would if it had to choose a single strategy.

Concurrent optimization is our default choice for solving LP models, and a user-selectable option for solving MIP models. The concurrent optimizer can be controlled in a few different ways. These will be discussed in this section. To avoid confusion when reporting results from multiple simultaneous solves, we've chosen to produce simplified logs and callbacks when performing concurrent optimization. These will also be discussed in this section.

Controlling Concurrent Optimization

If you wish to use the concurrent optimizer to solve your model, the steps you need to take depend on the model type. As mentioned earlier, the concurrent optimizer is the default choice for LP models. This choice is controlled by the Method parameter. For MIP models, you can select the concurrent optimizer by modifying the ConcurrentMIP parameter.

For LP models, the strategies used by the independent solvers can be controlled with the ConcurrentMethod parameter. While we reserve the right to change our choices in the future, with the default setting of ConcurrentMethod=-1 we currently devote the first concurrent thread to dual simplex, the second through fourth to a single parallel barrier solve, and the fifth to primal simplex. Additional threads are devoted to the parallel barrier solve. Thus, for example, a concurrent LP solve using four threads would devote one thread to dual simplex and three to parallel barrier.

For MIP, we divide available threads evenly among the independent solves, and we choose different values for the MIPFocus and Seed parameters for each.

If you want more control over concurrent optimization (e.g., to choose the exact strategies used for each independent solve), you can do so by creating two or more *concurrent environments*. These can be created via API routines (in C, C++, Java, .NET, or Python), or they can be created from `.prm` files using the ConcurrentSettings parameter if you are using our command-line interface. Once these have been created, subsequent optimization calls will start one independent solve for each concurrent environment you created. To control the strategies used for each solve, you simply set the parameters in each environment to the values you would like them to take in the corresponding solve. For example, if you create two concurrent environments and set the `MIPFocus` parameter to 1 in the first and 2 in the second, subsequent MIP optimize calls will perform two solves in parallel, one with `MIPFocus=1` and the other with `MIPFocus=2`.

Please note that parameter values are copied from the model when the concurrent environment is created. Therefore, changes to parameter values on the model have no effect on concurrent environments that have already been created. This is a frequent source of confusion.

Logging

Your first indication that the concurrent optimizer is being used is output in the Gurobi log that looks like this...

```
Concurrent LP optimizer: dual simplex and barrier
Showing barrier log only...
```

...or like this...

```
Concurrent MIP optimizer: 2 concurrent instances (2 threads per instance)
```

These log lines indicate how many independent solves will be launched. For the LP case, the lines also indicate which methods will be used for each.

Since it would be quite confusing to see results from multiple solves interleaved in a single log, we've chosen to use a simplified log format for concurrent optimization. For concurrent LP, we only present the log for a single solve. For concurrent MIP, the log is similar to our standard MIP log, except that it only provides periodic summary information (see the MIP logging section if you are unfamiliar with our standard MIP log). Each concurrent MIP log line shows the objective for the best feasible solution found by any of the independent solves to that point, the best objective bound proved by any of the independent solves, and the relative gap between these two values:

```
    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

    0     0      -      -      -   24.00000   13.00000  45.8%          0s
    0     0      -      -      -   16.50000   13.21154  19.9%          0s
    0     0      -      -      -   16.50000   13.25000  19.7%          0s
    0     0      -      -      -   16.50000   13.37500  18.9%          0s
    0     0      -      -      -   16.50000   13.37500  18.9%          0s
    0     0      -      -      -   16.50000   13.37500  18.9%          0s
    0     6      -      -      -   15.50000   13.37500  13.7%          0s
  310   149      -      -      -   15.00000   13.66923  8.87%          0s
 3873  1634      -      -      -   15.00000   14.00000  6.67%          5s
 9652  4298      -      -      -   15.00000   14.12500  5.83%         10s
16535  6991      -      -      -   15.00000   14.18056  5.46%         15s
23610  9427      -      -      -   15.00000   14.22333  5.18%         20s
...
```

We also include node counts from one of the independent solves, as well as elapsed times, to give some indication of forward progress.

Determinism

Concurrent optimization essentially sets up a race between multiple threads to solve your model, with the winning thread returning the solution that it found. In cases where multiple threads solve the model in roughly the same amount of time, small variations in runtime from one run to the next could mean that the winning thread is not the same each time. If your model has multiple optimal solutions (which is quite common in LP and MIP), then it is possible that running a concurrent solver multiple times on the same model could produce different optimal solutions. This is known as non-deterministic behavior.

By default, the Gurobi concurrent solvers all produce non-deterministic behavior. You can obtain deterministic behavior for the concurrent LP solver by setting the Method parameter to value 4. This setting typically increases runtimes slightly, but if your application is dependent on

deterministic behavior, deterministic concurrent LP is often your best option. There is no similar setting for the concurrent MIP solver.

Callbacks

Rather than providing callbacks from multiple independent solves simultaneously, we've again chosen to simplify behavior for the concurrent optimizer. In particular, we only supply callbacks from a single solve. A few consequences of this choice:

- Information retrieved by your callback (solutions, objective bounds, etc.) will come from a single model.
- User cutting planes are only applied to a single model.
- You aren't allowed to use lazy constraints with concurrent MIP, since they would only be applied to one model.

Callbacks are further restricted for multi-objective problems. While you would normally get both multi-objective-specific callbacks and callbacks from the algorithm that solves the model associated with each phase of the computation, you only get multi-objective-specific callbacks (and only from a single solve) for concurrent multi-objective solves.

Parameter Tuning Tool

The Gurobi Optimizer provides a wide variety of parameters that allow you to control the operation of the optimization engines. The level of control varies from extremely coarse-grained (e.g., the Method parameter, which allows you to choose the algorithm used to solve continuous models) to very fine-grained (e.g., the MarkowitzTol parameter, which allows you to adjust the tolerances used during simplex basis factorization). While these parameters provide a tremendous amount of user control, the immense space of possible options can present a significant challenge when you are searching for parameter settings that improve performance on a particular model. The purpose of the Gurobi tuning tool is to automate this search.

The Gurobi tuning tool performs multiple solves on your model, choosing different parameter settings for each solve, in a search for settings that improve runtime. The longer you let it run, the more likely it is to find a significant improvement. If you are using a Gurobi Compute Server, you can harness the power of multiple machines to perform distributed parallel tuning in order to speed up the search for effective parameter settings.

The tuning tool can be invoked through two different interfaces. You can either use the `grbtune` command-line tool, or you can invoke it from one of our programming language APIs. Both approaches share the same underlying tuning algorithm. The command-line tool offers more tuning features. For example, it allows you to provide a list of models to tune, or specify a list of base settings to try (TuneBaseSettings).

A number of tuning-related parameters allow you to control the operation of the tuning tool. The most important is probably TuneTimeLimit, which controls the amount of time spent searching for an improving parameter set. Other parameters include TuneTrials (which attempts to limit the impact of randomness on the result), TuneCriterion (which specifies the tuning criterion), TuneResults (which controls the number of results that are returned), and TuneOutput (which controls the amount of output produced by the tool).

Before we discuss the actual operation of the tuning tool, let us first provide a few caveats about the results. While parameter settings can have a big performance effect for many models, they aren't going to solve every performance issue. One reason is simply that there are many models for which even the best possible choice of parameter settings won't produce an acceptable result. Some models are simply too large and/or difficult to solve, while others may have numerical issues that can't be fixed with parameter changes.

Another limitation of automated tuning is that performance on a model can experience significant variations due to random effects (particularly for MIP models). This is the nature of search. The Gurobi algorithms often have to choose from among multiple, equally appealing alternatives. Seemingly innocuous changes to the model (such as changing the order of the constraint or variables), or subtle changes to the algorithm (such as modifying the random number seed) can lead to different choices. Often times, breaking a single tie in a different way can lead to an entirely different search. We've seen cases where subtle changes in the search produce 100X performance swings. While the tuning tool tries to limit the impact of these effects, the final result will typically still be heavily influenced by such issues.

The bottom line is that automated performance tuning is meant to give suggestions for parameters that could produce consistent, reliable improvements on your models. It is not meant to be a replacement for efficient modeling or careful performance testing.

27.1 Command-Line Tuning

The `grbtune` command-line tool provides a very simple way to invoke parameter tuning on a model (or a set of models). You specify a list of `parameter=value` arguments first, followed by the name of the file containing the model to be tuned. For example, you can issue the following command (in a Windows command window, or in a Linux/Mac terminal window)...

```
> grbtune TuneTimeLimit=10000 c:\gurobi1100\win64\examples\data\misc07
```

(substituting the appropriate path to a model, stored in an MPS or LP file). The tool will try to find parameter settings that reduce the runtime on the specified model. When the tuning run completes, it writes a set of `.prm` files in the current working directory that capture the best parameter settings that it found. It also writes the Gurobi log files for these runs (in a set of `.log` files).

You can also invoke the tuning tool through our programming language APIs. That will be discussed shortly.

If you specify multiple model files at the end of the command line, the tuning tool will try to find settings that minimize the total runtime for the listed models.

Running the Tuning Tool

The first thing the tuning tool does is to perform a baseline run. The parameters for this run are determined by your choice of initial parameter values. If you set a parameter, it will take the chosen value throughout tuning. Thus, for example, if you set the Method parameter to 2, then the baseline run and all subsequent tuning runs will include this setting. In the example above, you'd do this by issuing the command:

```
> grbtune Method=2 TuneTimeLimit=100 misc07
```

For a MIP model, you will note that the tuning tool actually performs several baseline runs, and captures the mean runtime over all of these trials. In fact, the tool will perform multiple runs for each parameter set considered. This is done to limit the impact of random effects on the results, as discussed earlier. Use the TuneTrials parameter to adjust the number of trials performed.

Once the baseline run is complete, the time for that run becomes the time to beat. The tool then starts its search for improved parameter settings. Under the default value of the TuneOutput parameter, the tool prints output for each parameter set that it tries...

```
Testing candidate parameter set 33...

        Method 2 (fixed)
        BranchDir -1
        Heuristics 0.001
        VarBranch 1
        Presolve 2
```

```
Solving with random seed #1 ... runtime 0.50s
Solving with random seed #2 ... runtime 0.60s+

Progress so far:
  baseline: mean runtime 0.76s
  best:     mean runtime 0.46s
Total elapsed tuning time 49s (51s remaining)
```

This output indicates that the tool has tried 33 parameter sets so far. For the 33rd set, it changed the value of the BranchDir parameter, the Heuristics parameter, the VarBranch parameter and the Presolve parameter (the Method parameter was changed in our initial parameter settings, so this change will appear in every parameter set that the tool tries and is marked to be fixed). The first trial solved the model in 0.50 seconds, while the second hit a time limit that was set by the tuning tool (as indicated by the + after the runtime output). If any trial hits a time limit, the corresponding parameter set is considered to be worse than any set that didn't hit a time limit. The output also shows that the best parameter set found so far gives a runtime of 0.46s. Finally, it shows elapsed and remaining runtime.

Tuning normally proceeds until the elapsed time exceeds the tuning time limit. However, hitting CTRL-C will also stop the tool.

When the tuning tool finishes, it prints a summary...

```
Tested 83 parameter sets in 98.87s

Baseline parameter set: mean runtime 0.76s

        Method 2 (fixed)

 # Name              0        1        2      Avg  Std Dev
 0 MISC07        0.85s    0.76s    0.67s    0.76s     0.07

Improved parameter set 1 (mean runtime 0.40s):

        Method 2 (fixed)
        DegenMoves 1
        Heuristics 0
        VarBranch 1
        CutPasses 5

 # Name              0        1        2      Avg  Std Dev
 0 MISC07        0.38s    0.41s    0.42s    0.40s     0.01

Improved parameter set 2 (mean runtime 0.44s):

        Method 2 (fixed)
        Heuristics 0
```

```
        VarBranch 1
        CutPasses 5

 # Name              0        1        2      Avg  Std Dev
 0 MISC07        0.42s    0.44s    0.45s    0.44s     0.01

Improved parameter set 3 (mean runtime 0.49s):

        Method 2 (fixed)
        Heuristics 0
        VarBranch 1

 # Name              0        1        2      Avg  Std Dev
 0 MISC07        0.49s    0.50s    0.49s    0.49s     0.01

Improved parameter set 4 (mean runtime 0.67s):

        Method 2 (fixed)
        VarBranch 1

 # Name              0        1        2      Avg  Std Dev
 0 MISC07        0.74s    0.58s    0.70s    0.67s     0.07

Wrote parameter files: tune1.prm through tune4.prm
Wrote log files: tune1.log through tune4.log
```

The summary shows the number of parameter sets it tried, and provides details on a few of the best parameter sets it found. It also shows the names of the `.prm` and `.log` files it writes. You can change the names of these files using the ResultFile parameter. If you set `ResultFile=model.prm`, for example, the tool would write `model1.prm` through `model4.prm` and `model1.log` through `model4.log`. For each displayed parameter set, we print the parameters used and a small summary table showing results for each model and each trial, together with the average runtime and the standard deviation.

The number of sets that are retained by the tuning tool is controlled by the TuneResults parameter. The default behavior is to keep the sets that achieve the best trade-off between runtime and the number of changed parameters. In other words, we report the set that achieves the best result when changing one parameter, when changing two parameters, etc. We actually report a Pareto frontier, so for example we won't report a result for three parameter changes if it is worse than the result for two parameter changes.

Other Tuning Parameters

So far, we've only talked about using the tuning tool to minimize the time to find an optimal solution. For MIP models, you can also minimize the optimality gap after a specified time limit. You don't have to take any special action to do this; you just set a time limit. Whenever a baseline run hits this limit, the tuning tool will automatically try to minimize the MIP gap. To give an example, the command...

```
> grbtune TimeLimit=100 glass4
```

...will look for a parameter set that minimizes the optimality gap achieved after 100s of runtime on model `glass4`. If the tool happens to find a parameter set that solves the model within the time limit, it will then try to find settings that minimize mean runtime.

For models that don't solve to optimality in the specified time limit, you can gain more control over the criterion used to choose a *winning* parameter set with the TuneCriterion parameter. This parameter allows you to tell the tuning tool to search for parameter settings that produce the best incumbent solution or the best lower bound, rather than always minimizing the MIP gap,

You can modify the TuneOutput parameter to produce more or less output. The default value is 2. A setting of 0 produces no output; a setting of 1 only produces output when an improvement is found; a setting of 3 produces a complete Gurobi log for each run performed.

If you would like to use a MIP start with your tuning run, you can include the name of the start file immediately after the model name in the argument list. For example:

```
> grbtune misc07.mps misc07.mst
```

You can also use MIP starts when tuning over multiple models; any model that is immediately followed by a start file in the argument list will use the corresponding start. For example:

```
> grbtune misc07.mps misc07.mst p0033.mps p0548.mps p0548.mst
```

27.2 Tuning API

The tuning tool can be invoked from our C, C++, Java, .NET, and Python interfaces. The tool behaves slightly differently when invoked from these interfaces. Rather than writing the results to a set of files, upon completion the tool populates a TuneResultCount attribute, which gives a count of the number of improving parameter sets that were found and retained. The user program can then query the value of this attribute, and then use the `GetTuneResult` method to copy any of these parameter sets into a model (using C, C++, Java, .NET, or Python). Once loaded into the model, the parameter set can be used to perform a subsequent optimization, or the list of changed parameters can be written to a `.prm` file using the appropriate `Write` routine (from C, C++, Java, .NET, or Python).

Gurobi Instant Cloud

Gurobi Instant Cloud allows you to start and stop Gurobi Compute Servers on the cloud. You can start multiple machines without the need for your own hardware or local Gurobi licenses. Computations are seamlessly offloaded to these servers. Depending on your cloud license type, these machines provide the full set of Compute Server features, including queuing, load balancing, and distributed parallel computing.

Overview

When using the Instant Cloud, there are always three systems involved: your client machine, the Instant Cloud Manager, and a cloud Compute Server.

The program that requests a Gurobi Instant Cloud machine and submits optimization models to this server runs on your client machine. Note, however, that this program does not actually need to be aware that it will be using Gurobi Instant Cloud. You have a few options for configuring the client to use the Instant Cloud. The simplest and most seamless is to set up a cloud license file. The alternative is to use a programming language API, which gives your program additional control over how it uses the cloud. Details on launching cloud machines from your client program follow shortly.

The Instant Cloud Manager manages the configuration and launching of cloud machines. Your client program will send credential information to this website, along with a request to launch an Instant Cloud machine. The specific action taken in response to this request depend on configuration information that you manage through the website. For each license, you set up things like the number of servers to launch, the types and geographic regions of these machines, the maximum number of simultaneous jobs to run on each server, etc.

Once the Instant Cloud Manager launches the requested Compute Servers, it passes information about these servers back to your client program. The client program then directly interacts with the servers, sending the Gurobi model, launching a solve on the model, requesting solution information, etc. As with any Gurobi Compute Server, this process is entirely transparent to the client program.

Now that we've given a high-level description of the overall process, we need to cover a few important details.

28.1 Client Setup

As noted previously, a client program that wishes to launch a Gurobi Instant Cloud machine must pass credential information to the Instant Cloud Manager. Every Instant Cloud license has such credentials associated with it. This information is captured in a pair strings, an *access ID* and a *secret key*. These strings can be retrieved from your account on the Instant Cloud Manager. Note that you shouldn't share these credentials with others, since anyone who knows these two strings can launch Instant Cloud machines in your account.

Once you have the credentials associated with your license, there are two ways to configure your client program to use them. The simplest is to use a cloud license file. This is just like any other

Gurobi license file, except that its fields are specific to the cloud. A cloud license file will contain two lines with credential information:

```
CLOUDACCESSID=312e9gef-e0bc-4114-b6fb-26ed7klaeff9
CLOUDKEY=ae32L0H321dgaL
```

It may also contain an optional third line:

```
CLOUDPOOL=pool1
```

We'll discuss cloud pools a bit later. You can download a *gurobi.lic* file containing this information from the Instant Cloud website, or you can create one yourself in a text editor. If you follow the standard process for setting up a Gurobi license file (refer to the Getting Started Knowledge Base article for details), then Gurobi will automatically use the Instant Cloud rather than running locally.

The other option for passing credential information to the Instant Cloud Manager is programmatically via parameters. Use the configuration parameters to set access ID and secret key.

28.2 Instant Cloud Setup

As noted previously, cloud configuration is done via the Instant Cloud Manager. The client program requests that a cloud machine be launched, but the Instant Cloud Manager determines exactly how to respond to that request.

One essential concept when configuring your Instant Cloud license is the notion of a *cloud pool*. Pools allow you to create multiple configurations within a single cloud license. For example, you may set up one pool for jobs in the US and another for jobs in Europe, or one for short-running jobs and another for long-running jobs, or one for single-machine jobs and another for distributed parallel jobs. For each of the available cloud configuration options (which will be discussed below), you can select different values for different pools. Every license always has a default pool, which comes preconfigured with what we consider to be reasonable default values. Thus, you always have the option of ignoring cloud pools and simply using the default pool if you don't need multiple configurations.

The main things that a user may want to configure on the Instant Cloud website are the idle shutdown time, the number of machines to launch, the number of distributed workers to launch, the machine region, and the machine type. These can take different values in different pools.

The idle shutdown time is a vital concept in the Instant Cloud. When a client program requests a cloud server, it takes some time (typically less than 2 minutes) to launch that server. Rather than forcing client programs to incur this delay each time they run, the Gurobi Instant Cloud leaves a server running until is has been idle for the specified idle shutdown time. In this way, later client programs may find a cloud server already available. You can set this to a small value if you want your server to shut down immediately after your job finishes, or to a very large value if you want your server to always be available.

Another configuration option is the number of machines associated with the pool. Gurobi Compute Server automatically handles queuing and load balancing between servers, so launching multiple machines allows you to distribute the work of many simultaneous client programs among them. A pool can also be configured to launch any number of distributed workers, if you want to use distributed computing.

Cloud machines can be launched in multiple geographic regions, including the US, Europe, Asia, and South America. You should visit the website to see the full list. We offer several options

for machine type, although we've chosen what we believe is the best general-purpose machine for running Gurobi as the default, so you are unlikely to want to change this setting.

Guidelines for Numerical Issues

Numerical instability is a generic label often applied to situations where solving an optimization model produces results that are erratic, inconsistent, or unexpected, or when the underlying algorithms exhibit poor performance or are unable to converge. There are many potential causes of this behavior; however, most can be grouped into four categories:

- Rounding coefficients while building the model.
- Limitations of floating-point arithmetic.
- Unrealistic expectations about achievable precision.
- Ill conditioning, or geometry-induced issues.

This section explains these issues and how they affect both performance and solution quality. We also provide some general rules and some advanced techniques to help avoid them. Although we will treat each of these four sources separately, it is important to remember that their effects often feed off of each other. We also provide tips on how to diagnose numerical instability in your models.

Finally, we discuss the Gurobi parameters that can be modified to improve solution accuracy. We should stress now, however, that the best way to improve numerical behavior and performance is to reformulate your model. Parameters can help to manage the effects of numerical issues, but there are limits to what they can do, and they typically come with a substantial performance cost.

29.1 Avoid rounding of input

A common source of numerical issues is numerical rounding in the numbers that are used to represent constraint matrix coefficients. To illustrate the issue, consider the following example:

$$\begin{aligned} x - 6y &= 1 \\ 0.333x - 2y &= .333 \end{aligned}$$

It may be tempting to say that the two equations are equivalent, but adding both to a model will lead to an incorrect result. This is an important point for our users: Gurobi will always trust the input numbers that they provide, and will never change them unless the change can be shown to not affect the solution.

So, with this in mind, during presolve Gurobi can use the second constraint to determine:

$$y := 0.1665x - 0.1665$$

When substituted into the first constraint, this yields

$$\begin{aligned} x - 6 \cdot (0.1665x - 0.1665) &= 1 \\ \Leftrightarrow 0.001x &= 0.001 \end{aligned}$$

and thus $x = 1,\ y = 0$ as the only solution.

If user had provided these two equations instead:

$$\begin{aligned} x - 6y &= 1 \\ 0.3333333333333333x - 2y &= 0.3333333333333333 \end{aligned}$$

this would give:

$$y := 0.1666666666666667x - 0.1666666666666667$$

which yields:

$$\begin{aligned} x - 6 \cdot (0.1666666666666667x - 0.1666666666666667) &= 1 \\ \Leftrightarrow 2 \cdot 10^{-16}x + 1 + 2 \cdot 10^{-16} &\approx 1 \end{aligned}$$

Even with a very small threshold for treating a coefficient as zero, the result here is that the first constraint is truly redundant. Any solution with $x = 6y + 1$ would be accepted as feasible.

The main point is that constraints that are exactly parallel, or linearly dependent (within double-precision floating-point and small tolerances) are harmless, but constraints that are almost parallel to each other produce tiny coefficients in the linear system solves and in preprocessing, which can wreak havoc on the solution process. In the next section, we expand on the limits *double-precision floating-point* numbers, and in particular why $1 \approx 1 + 2 \cdot 10^{-16}$.

29.2 Real numbers are not real

To say that real numbers aren't real is not just a play on words, but a computational reality. Let's do a simple experiment: try the following in your favorite number-crunching tool. In Excel:

```
=IF(1+1E-016 = 1,1,0)
```

will print 1. In Python:

```
>>> 1 == 1+1e-16
True
```

In C, the code

```
#include<stdio.h>
int main(void)
{
  if (1+1e-16 == 1) printf("True\n");
  else              printf("False\n");
  return 0;
}
```

will print `True`. In R:

```
> 1 == 1+1e-16
[1] TRUE
```

Note that this behavior is not restricted to *small* numbers; it also happens with larger numbers. For example:

```
>>> 1+1e16 == 1e16
True
```

This shows that the *precision* of the result depends on the relative scale of the involved numbers.

Although this behavior is typical, there are some exceptions. One is the GNU-bc command line tool:

```
> bc
1.0 == 1.0+10^(-16)
1
scale=20
1.0 == 1.0+10^(-16)
0
1.0 == 1.0+10^(-21)
1
```

When we set the `scale` parameter to 20, the code is able to recognize that the numbers are different. This just shifts the bar, though; `bc` still fails to recognize the difference between the last two numbers. Another library that allows for extended, or even *unlimited* (up to memory) precision is the GNU Multiple Precision Arithmetic Library, but its details are beyond the scope of this document.

The reason for these *failures* is that computers must store numbers as a sequence of bits, and most common implementations adhere to the IEEE 754 standard. In particular, IEEE-754 sets the standard for *double-precision* format. This standard is so pervasive that almost all computers have specialized hardware to improve performance for operations on numbers represented as such. One consequence is that mathematical operations on alternative extended number representations tend to be significantly slower than operations on numbers represented following the IEEE 754 standard. Degradation of 10X or even 100X are common.

Due to the performance obtained from hardware support for double-precision arithmetic, Gurobi relies on this standard (as does most software). However, this speed comes at a cost: computed results often differ from what mathematics may dictate. For example, the associative property (`a + (b + c) = (a + b) + c`) is a fundamental property of arithmetic, but double-precision arithmetic gives (in Python):

```
>>> (1+1e-16)+1e-16 == 1 + (1e-16 + 1e-16)
False
```

Furthermore, many common numbers (e.g. 0.1) cannot be represented exactly.

Consequently, simple questions like whether two numbers are equal, or whether a number is equal zero, or whether a number is integral, can be quite complicated when using floating-point arithmetic.

29.3 Tolerances and user-scaling

Gurobi will solve the model as defined by the user. However, when evaluating a candidate solution for feasibility, in order to account for possible round-off errors in the floating-point evaluations, we must allow for some *tolerances*.

To be more precise, satisfying *Optimality Conditions* requires us to test at least the following three criteria:

`IntFeasTol`: Integrality of solutions, i.e., whether a integer variable x takes an integer value or not.
More precisely, x will be considered *integral* if `abs(x - floor(x + 0.5))` $\leq$ `IntFeasTol`.

`FeasibilityTol`: Feasibility of primal constraints, i.e., whether $a \cdot x \leq b$ holds for the *primal* solution. More precisely, $a \cdot x \leq b$ will be considered to hold if `(a * x) - b` $\leq$ `FeasibilityTol`.

`OptimalityTol`: Feasibility of dual constraints, i.e., whether $a \cdot y \leq c$ holds for the *dual* solution. More precisely, $a \cdot y \leq c$ will be considered to hold if `(a * y) - c` $\leq$ `OptimalityTol`.

Note that these tolerances are **absolute**; they do not depend on the scale of the quantities involved in the computation. This means that when formulating a problem, these tolerances should be taken into account, specially to select the units in which variables and constraints will be expressed.

It is very important to note that the usage of these *tolerances* implicitly defines a *gray zone* in the search space in which solutions that are very slightly infeasible can still be accepted as feasible. However, the solver will not explicitly search for such solutions.

For this reason, it is actually possible (although highly unlikely for well-posed problems) for a model to be reported as being both *feasible* and *infeasible* (in the sense stated above). This can occur if the model is infeasible in exact arithmetic, but there exists a solution that is feasible within the solver tolerances. For instance, consider:

$$\begin{array}{lll} \min & 0 & \\ s.t. & x \leq & 0 \\ & x \geq & 10^{-10} \end{array}$$

Models at the edge of infeasibility

- As we saw in the introduction, seemingly contradictory results regarding the feasibility or infeasibility of a model can legitimately occur for models that are at the boundary between feasibility and infeasibility.
- A more complicated example is $x + 10^8 y = -1, x,\ y >= 0$. It has two bases, one where x is basic and one where y is basic. If x is basic, we get $x = -1$, which is clearly infeasible. However, if y is basic we get $y = -10^8$, which is feasible within tolerance. Different algorithms could lead to either of such bases and thus come to apparently contradictory feasibility results.
- Presolve reductions can also play a role. A presolve reduction, e.g. fixing a variable to a bound, implicitly forces a tolerance of 0 for that variable. When solving the reduced model, the optimizer therefore no longer has the option to "spread" a slight infeasibility of the model over these variables and produce a solution that is feasible within tolerances. This leads to seemingly contradictory feasibility results when solving the model with presolve enabled or disabled.
- What can be done to diagnose such cases:
  - First step is to tighten the `FeasibilityTol` to 10^{-9} and try again. In many cases this will lead to a consistent declaration of infeasibility of the model at hand, which tells you that the model is on this boundary of infeasibility.
  - Use `feasRelax` to solve your model (again with a tight `FeasibilityTol`. This boundary case is identified by a non-zero relaxation value.
  - Compute the IIS (again with a tight `FeasibilityTol`) to analyze the infeasibility.
- Another source of seemlingly contradictory results is due to numerical issues of the model and will be discussed in the following subsections.

Gurobi tolerances and the limitations of double-precision arithmetic

The default values for these primal and dual feasibility tolerances are 10^{-6}, and the default for the integrality tolerance is 10^{-5}. If you choose the range for your inequalities and variables correctly, you can typically ignore tolerance issues entirely.

To give an example, if your constraint right-hand side is on the order of 10^3, then relative numeric errors from computations involving the constraint (if any) are likely to be less than 10^{-9}, i.e., less than one in a billion. This is usually far more accurate than the accuracy of input data, or even of what can be measured in practice.

However, if you define a variable $x \in [-10^{-6}, 10^{-6}]$, then relative numeric error may be as big as 50% of the variable range.

If, on the other hand, you have a variable $x \in [-10^{10}, 10^{10}]$, and you are using default primal feasibility tolerances; then what you are really asking is for the relative numeric error (if any) to be less than 10^{-16}. However, this is beyond the limits of comparison for double-precision numbers. This implies that you are not allowing any round-off error at all when testing feasible solutions for this particular variable. And although this might sound as a good idea, in fact, it is really bad, as any round-off computation may result in your truly optimal solution being rejected as infeasible.

Why scaling and geometry is relevant

This section provides a simple example of how scaling problems can slow down problem solving and, in extreme cases, result in unexpected answers. Consider the problem:

$$(P)\, \max\{cx : Ax = b, l \leq x \leq u\}$$

and let D be a diagonal matrix where $D_{ii} > 0$, $\forall i$. In theory, solving (P) should be equivalent to solving the related problem (P_D):

$$(P_D)\, \max\{cDx' : ADx' = b, D^{-1}l \leq x' \leq D^{-1}u\}$$

However, in practice, the two models behave very differently. To demonstrate this, we use a simple script `rescale.py` that randomly rescales the columns of the model. Let's consider the impact of rescaling on the problem `pilotnov.mps.bz2`. Solving the original problem gives the following output:

```
Using license file /opt/gurobi950/gurobi.lic
Read MPS format model from file pilotnov.mps.bz2
Reading time = 0.01 seconds
PILOTNOV: 975 rows, 2172 columns, 13057 nonzeros
Gurobi Optimizer version 9.5.0 build v9.5.0rc0 (linux64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Model fingerprint: 0xe6e7cf84
Coefficient statistics:
  Matrix range     [1e-06, 1e+07]
  Objective range  [2e-03, 1e+00]
  Bounds range     [5e-06, 8e+04]
  RHS range        [1e-05, 4e+04]
Warning: Model contains large matrix coefficient range
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 254 rows and 513 columns
```

```
Presolve time: 0.01s
Presolved: 721 rows, 1659 columns, 11455 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -3.2008682e+05   1.274630e+05   0.000000e+00      0s
Extra simplex iterations after uncrush: 1
    1085   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 1085 iterations and 0.05 seconds
Optimal objective -4.497276188e+03
Kappa: 1.841766e+07
```

Note the log message regarding the matrix coefficient range in the log (which in this case shows a range of [1e-06, 1e+07]).

If we run `rescale.py -f pilotnov.mps.bz2 -s 1e3` (randomly rescaling columns up or down by as much as 10^3), we obtain:

```
Using license file /opt/gurobi950/gurobi.lic
Read MPS format model from file pilotnov.mps.bz2
Reading time = 0.01 seconds
PILOTNOV: 975 rows, 2172 columns, 13057 nonzeros
Gurobi Optimizer version 9.5.0 build v9.5.0rc0 (linux64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Model fingerprint: 0x79ed9354
Coefficient statistics:
  Matrix range     [6e-09, 1e+10]
  Objective range  [5e-06, 9e+02]
  Bounds range     [5e-09, 5e+07]
  RHS range        [1e-05, 4e+04]
Warning: Model contains large matrix coefficient range
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 100 rows and 255 columns
Presolve time: 0.00s
Presolved: 875 rows, 1917 columns, 11899 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -6.2390277e+32   7.095340e+31   6.239028e+02      0s
Extra simplex iterations after uncrush: 2
    1141   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 1141 iterations and 0.05 seconds
Optimal objective -4.497276188e+03
Kappa: 1.166265e+18
```

This time, the optimization process takes more iterations, and also, we get an extra warning:

```
Extra 2 simplex iterations after uncrush,
```

This indicates that extra simplex iterations were performed on the unpresolved model. Also, note the very large value for `Kappa`; its meaning will be discussed in this section.

If we run `rescale.py -f pilotnov.mps.bz2 -s 1e6`, we obtain:

```
Using license file /opt/gurobi950/gurobi.lic
Read MPS format model from file pilotnov.mps.bz2
Reading time = 0.01 seconds
PILOTNOV: 975 rows, 2172 columns, 13057 nonzeros
Gurobi Optimizer version 9.5.0 build v9.5.0rc0 (linux64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Model fingerprint: 0xf2c9bd15
Coefficient statistics:
  Matrix range     [9e-12, 1e+13]
  Objective range  [2e-09, 1e+06]
  Bounds range     [6e-12, 4e+10]
  RHS range        [1e-05, 4e+04]
Warning: Model contains large matrix coefficient range
Warning: Model contains large bounds
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 101 rows and 255 columns
Presolve time: 0.00s
Presolved: 874 rows, 1917 columns, 11897 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -6.8659608e+34   7.041252e+31   6.865961e+04      0s
Extra simplex iterations after uncrush: 47
    1537   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 1537 iterations and 0.07 seconds
Optimal objective -4.497276188e+03
Kappa: 1.851538e+25
```

Now we get a much larger number of extra simplex iterations, and more troublingly, we get a warning about the quality of the resulting solution:

```
   Warning:  unscaled primal violation = 5.65274e-05 and residual = 5.65274e-05,
```

This message indicates that the solver had trouble finding a solution that satisfies the default tolerances.

Finally, if we run `rescale.py -f pilotnov.mps.bz2 -s 1e8`, we obtain:

```
Using license file /opt/gurobi950/gurobi.lic
Read MPS format model from file pilotnov.mps.bz2
Reading time = 0.01 seconds
PILOTNOV: 975 rows, 2172 columns, 13057 nonzeros
Gurobi Optimizer version 9.5.0 build v9.5.0rc0 (linux64)
Thread count: 6 physical cores, 12 logical processors, using up to 12 threads
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Model fingerprint: 0xfea2a63e
Coefficient statistics:
  Matrix range     [3e-13, 1e+15]
  Objective range  [6e-11, 2e+08]
  Bounds range     [6e-14, 5e+12]
  RHS range        [1e-05, 4e+04]
Warning: Model contains large matrix coefficient range
```

```
Warning: Model contains large bounds
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 103 rows and 258 columns
Presolve time: 0.00s
Presolved: 872 rows, 1914 columns, 11795 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -9.3067584e+36   7.106821e+31   9.306758e+06      0s

Solved in 1469 iterations and 0.06 seconds
Infeasible model
```

In this case, the optimization run terminates almost instantly, but with the unexpected `Infeasible` result.

As you can see, as we performed larger and larger rescalings, we continued to obtain the same optimal value, but there were clear signs that the solver struggled. We see warning messages, as well increasing iteration counts, runtimes, and `Kappa` values. However, once we pass a certain rescaling value, the solver is no longer able to solve the model and instead reports that it is `Infeasible`.

Note that this is not a bug in Gurobi. It has to do with changing the meaning of numbers depending on their range, the use of fixed tolerances, and in the changing geometry of the problem due to scaling. We will discuss this topic further in a later section.

Recommended ranges for variables and constraints

Keeping the lessons of the previous section in mind, we recommended that right-hand sides of inequalities representing physical quantities (even budgets) should be scaled so that they are on the order of 10^4 or less. The same applies to variable domains, as variable bounds are again linear constraints.

In the case of objective functions, we recommend that good solutions should have an optimal value that is less than 10^4, and ideally also above one (unless the objective coefficients are all zero). This is because the `OptimalityTol` is used to ensure that *reduced cost* are *close enough* to zero. If coefficients are too large, we again face difficulties in determining whether an LP solution truly satisfies the optimality conditions or not. On the other hand, if the coefficients are too small, then it may be too easy to satisfy the feasibility conditions.

The coefficients of the constraint matrix are actually more important than the right-hand side values, variable bounds, and objective coefficients mentioned here. We'll discuss those shortly.

Improving ranges for variables and constraints

There are three common ways to improve ranges for objectives, constraints and variables:

- Use problem-specific information to tighten bounds:

  Although presolve, and, in particular, bound strengthening, is quite good at deriving implied variables bounds, it may not have access to all of the information known to the modeler. Incorporating tighter bounds directly into the model can not only improve the numerical behavior, but it can also speed up the optimization process.

- Choose the right units to express your variables and constraints:

When defining your variables and constraints, it is important to choose *units* that are consistent with tolerances. To give an example, a constraint with a 10^{10} right-hand side value is not going to work well with the default 10^{-6} feasibility tolerance. By changing the units (e.g., replacing pounds with tons, or dollars with millions of dollars, or ...), it is often possible to significantly improve the numerics of the problems.

- Disaggregate multiple objectives:

  A common source for very large range of objective coefficients is the practice of modeling hierarchical objectives as an aggregation of objective functions with large multipliers. For example, if the user wants to optimize a problem P with objective function $f_1(x)$ and then, subject to $f_1(x)$ being optimal, optimize $f_2(x)$, a common trick is to use as surrogate objective $\bar{f}(x) = Mf_1(x) + f_2(x)$ where M is a large constant. When you combine a large M with a relatively tight dual feasibility tolerance, it becomes much harder for the solver to find solutions that achieve dual feasibility. We recommend that you either use as small a constant M as possible or reformulate your model using a hierarchical objective (which is made easier by our multi-objective optimization features).

These techniques are usually sufficient to eliminate the problems that arise from bad scaling.

Advanced user scaling

In the previous sections, we presented some simple strategies to limit the ranges of variable bounds, constraint right-hand sides, objective values, and constraint matrix coefficients. However, it could happen that by scaling constraints or variables, some constraint coefficients become too small. Note that Gurobi will treat any constraint coefficient with absolute value under 10^{-13} as zero. Consider the following example:

$$\begin{aligned} 10^{-7}x + 10y &\leq 10 \\ x + 10^4 z &\leq 10^3 \\ x, y, z &\geq 0, \end{aligned}$$

In this example, the matrix coefficients range in $[10^{-7}, 10^4]$. If we multiply all x coefficients by 10^5, and divide all coefficients in the second constraint by 10^3, we obtain:

$$\begin{aligned} 10^{-2}x' + 10y &\leq 10 \\ 10^2 x' + 10z &\leq 1 \\ x', y, z &\geq 0, \end{aligned}$$

where $x = 10^5 x'$. The resulting matrix coefficients have a range in $[10^{-2}, 10^2]$. Essentially the trick is to *simultaneously* scale a column and a row to achieve a smaller range in the coefficient matrix.

We recommend that you scale the matrix coefficients so that their range is contained in six orders of magnitude or less, and hopefully within $[10^{-3}, 10^6]$.

Avoid hiding large coefficients

As we said before, a typical recommendation for improving numerics is to limit the range of constraint matrix coefficients. The rationale behind this guideline is that terms to be added in a linear expression should be of comparable magnitudes so that rounding errors are minimized. For example:

$$\begin{aligned} x - 10^6 y &\geq 0 \\ y &\in [0, 10] \end{aligned}$$

is usually considered a potential source of numerical instabilities due to the wide range of the coefficients in the constraint. However, it is easy to implement a simple (but useless) alternative:

$$\begin{aligned} x - 10y_1 &\geq 0 \\ y_1 - 10y_2 &= 0 \\ y_2 - 10y_3 &= 0 \\ y_3 - 10y_4 &= 0 \\ y_4 - 10y_5 &= 0 \\ y_5 - 10y &= 0 \\ y &\in [0, 10] \end{aligned}$$

This form certainly has nicer values in the matrix. However, the solution $y = -10^{-6}$, $x = -1$ might still be considered feasible as the bounds on variables and constraints might be violated within the tolerances. A better alternative is to reformulate

$$\begin{aligned} x - 10^6 y &\geq 0 \\ y &\in [0, 10] \end{aligned}$$

as

$$\begin{aligned} x - 10^3 y' &\geq 0 \\ y' &\in [0, 10^4] \end{aligned}$$

where $10^{-3} y' = y$. In this setting, the most negative values for x which might be considered feasible would be -10^{-3}, and for the original y variable it would be -10^{-9}, which is a clear improvement over the original situation.

Dealing with big-M constraints

Big-M constraints are a regular source of instability for optimization problems. They are so named because they typically involve a large coefficient M that is chosen to be larger than any reasonable value that a continuous variable or expression may take. Here's a simple example:

$$\begin{aligned} x &\leq 10^6 y \\ x &\geq 0 \\ y &\in \{0, 1\}, \end{aligned}$$

Big-M constraints are typically used to propagate the implications of a binary, on-off decision to a continuous variable. For example, a big-M might be used to enforce the condition that an edge can only admit flow if you pay the fixed charge associated with opening the edge, or a facility can only produce products if you build it. In our example, note that the $y = 0.0000099999$ satisfies the default integrality tolerance (`IntFeasTol`=10^{-5}), which allows x to take a value of 9.999. In other words, x can take a positive value without incurring an expensive fixed charge on y, which subverts the intent of only allowing a non-zero value for x when the binary variable y has the value of 1. You can reduce the effect of this behavior by adjusting the `IntFeasTol` parameter, but you can't avoid it entirely.

However, if the modeler has additional information that the x variable will never be larger than 10^3, then you could reformulate the earlier constraint as:

$$\begin{aligned} x &\leq 10^3 y \\ x &\geq 0 \\ y &\in \{0,1\} \end{aligned}$$

And now, $y = 0.0000099999$ would only allow for $x \leq 0.01$.

For cases when it is not possible to either rescale variable x or tighten its bounds, an SOS constraints or an indicator constraint (of the form $y = 0 \Rightarrow x = 0$) may produce more accurate solutions, but often at the expense of additional processing time.

29.4 Does my model have numerical issues?

You can follow any of the following suggestions to help determine whether a model is experiencing numerical issues:

1. Using the Gurobi Interactive shell, run some simple Python code to read the model that the replay produces, and print the summary statistics:

   ```
   m = read('gurobi.rew')
   m.printStats()
   ```

   The output will look like:

   ```
   Statistics for model (null) :
     Linear constraint matrix    : 25050 Constrs, 15820 Vars, 94874 NZs
     Variable types              : 14836 Continuous, 984 Integer
     Matrix coefficient range    : [ 0.00099, 6e+06 ]
     Objective coefficient range : [ 0.2, 65 ]
     Variable bound range        : [ 1, 5e+07 ]
     RHS coefficient range       : [ 1, 5e+07 ]
   ```

   The range of numerical coefficients is one indication of potential numerical issues. As a very rough guideline, the ratio of the largest to the smallest coefficient should be less than 10^9; smaller is better.

   In this example, the matrix range is

   $$6 \cdot 10^6 / 0.00099 = 6.0606 \cdot 10^9.$$

2. If possible, re-solve the model using the same parameters and review the logs. With the Python shell, use code like the following:

   ```
   m.read('gurobi.prm')
   m.optimize()
   ```

   Here are some examples of warning messages that suggest numerical issues:

   ```
   Warning: Model contains large matrix coefficient range
            Consider reformulating model or setting NumericFocus parameter
            to avoid numerical issues.
   Warning: Markowitz tolerance tightened to 0.5
   Warning: switch to quad precision
   Numeric error
   ```

```
Numerical trouble encountered
Restart crossover...
Sub-optimal termination
Warning: ... variables dropped from basis
Warning: unscaled primal violation = ... and residual = ...
Warning: unscaled dual violation = ... and residual = ...
```

3. When the `optimize` function completes, print solution statistics. With the Python shell, use code like the following:

   ```
   m.printQuality()
   ```

   which provides a summary of solution quality:

   ```
   Solution quality statistics for model Unnamed :
     Maximum violation:
       Bound       : 2.98023224e-08 (X234)
       Constraint  : 9.30786133e-04 (C5)
       Integrality : 0.00000000e+00
   ```

   Violations that are larger than the tolerances are another indication of numerical issues. Also, for a pure LP (without integer variables), print the condition number via the following Python command:

   ```
   m.KappaExact
   ```

   The condition number measures the potential for error in linear calculations; a large condition number, such as 10^{12}, is another indication of possible numerical issues, see this section for more details.

4. If changing parameters (e.g., Method or Seed) leads to a different optimization status (e.g., `Infeasible` instead of `optimal`), or if the optimal objective values changes, this is usually a sign of numerical issues. To further assess this you can tighten tolerances (to the order of 10^{-8} or even 10^{-9}), and see if the behavior of the solver becomes consistent again. Note that tightening tolerances usually comes at the price of more computing time, and should not be considered as a solution for numerical issues.

29.5 Solver parameters to manage numerical issues

Reformulating a model may not always be possible, or it may not completely resolve numerical issues. When you must solve a model that has numerical issues, some Gurobi parameters can be helpful. We discuss these now, in descending order of relevance.

Presolve

Gurobi presolve algorithms are designed to make a model smaller and easier to solve. However, in some cases, presolve can contribute to numerical issues. The following Python code can help you determine if this is happening. First, read the model file and print summary statistics for the presolved model:

```
m = read('gurobi.rew')
p = m.presolve()
p.printStats()
```

If the numerical range looks much worse than the original model, try the parameter `Aggregate=0`:

```
m.reset()
m.Params.Aggregate = 0
p = m.presolve()
p.printStats()
```

If the resulting model is still numerically problematic, you may need to disable presolve completely using the parameter `Presolve=0`; try the steps above using

```
m.reset()
m.Params.Presolve = 0
p = m.presolve()
p.printStats()
```

If the statistics look better with `Aggregate=0` or `Presolve=0`, you should further test these parameters. For a continuous (LP) model, you can test them directly. For a MIP, you should compare the LP relaxation with and without these parameters. The following Python commands create three LP relaxations: the model without presolve, the model with presolve, and the model with `Aggregate=0`:

```
m = read('gurobi.rew')
r = m.relax()
r.write('gurobi.relax-nopre.rew')
p = m.presolve()
r = p.relax()
r.write('gurobi.relax-pre.rew')
m.reset()
m.Params.Aggregate = 0
p = m.presolve()
r = p.relax()
r.write('gurobi.relax-agg0.rew')
```

With these three files, use the techniques mentioned earlier to determine if `Presolve=0` or `Aggregate=0` improves the numerics of the LP relaxation.

Finally, if `Aggregate=0` helps numerics but makes the model too slow, try `AggFill=0` instead.

Choosing the right algorithm

Gurobi Optimizer provides two main algorithms to solve continuous models and the continuous relaxations of mixed-integer models: barrier and simplex.

The barrier algorithm is usually fastest for large, difficult models. However, it is also more numerically sensitive. And even when the barrier algorithm converges, the crossover algorithm that usually follows can stall due to numerical issues.

The simplex method is often a good alternative, since it is generally less sensitive to numerical issues. To use dual simplex or primal simplex, set the Method parameter to 1 or 0, respectively.

Note that, in many optimization applications, not all problem instances have numerical issues. Thus, choosing simplex exclusively may prevent you from taking advantage of the performance advantages of the barrier algorithm on numerically well-behaved instances. In such cases, you should use the concurrent optimizer, which uses multiple algorithms simultaneously and returns

the solution from the first one to finish. The concurrent optimizer is the default for LP models, and can be selected for MIP by setting the Method parameter to 3 or 4.

For detailed control over the concurrent optimizer, you can create concurrent environments, where you can set specific algorithmic parameters for each concurrent solve. For example, you can create one concurrent environment with `Method=0` and another with `Method=1` to use primal and dual simplex simultaneously. Finally, you can use concurrent optimization with multiple distinct computers using distributed optimization. On a single computer, the different algorithms run on multiple threads, each using different processor cores. With distributed optimization, independent computers run the separate algorithms, which can be faster since the computers do not compete for access to memory.

Making the algorithm less sensitive

When all else fails, try the following parameters to make the algorithms more robust:

`ScaleFlag, ObjScale (All models)`: It is always best to reformulate a model yourself. However, for cases when that is not possible, these two parameters provide some of the same benefits. Set `ScaleFlag=2` for aggressive scaling of the coefficient matrix. `ObjScale` rescales the objective row; a negative value will use the largest objective coefficient to choose the scaling. For example, `ObjScale=-0.5` will divide all objective coefficients by the square root of the largest objective coefficient.

`NumericFocus (All models)`: The `NumericFocus` parameter controls how the solver manages numerical issues. Settings 1-3 increasingly shift the focus towards more care in numerical computations, which can impact performance. The `NumericFocus` parameter employs a number of strategies to improve numerical behavior, including the use of quad precision and a tighter `Markowitz` tolerance. It is generally sufficient to try different values of `NumericFocus`. However, when `NumericFocus` helps numerics but makes everything much slower, you can try setting `Quad=1` and/or larger values of `MarkowitzTol` such as 0.1 or 0.5.

`NormAdjust (Simplex)`: In some cases, the solver can be more robust with different values of the simplex pricing norm. Try setting `NormAdjust` to 0, 1, 2 or 3.

`BarHomogeneous (Barrier)`: For models that are infeasible or unbounded, the default barrier algorithm may have numerical issues. Try setting `BarHomogeneous=1`.

`CrossoverBasis (Barrier)`: Setting `CrossoverBasis=1` takes more time but can be more robust when creating the initial crossover basis.

`GomoryPasses (MIP)`: In some MIP models, Gomory cuts can contribute to numerical issues. Setting `GomoryPasses=0` may help numerics, but it may make the MIP more difficult to solve.

`Cuts (MIP)`: In some MIP models, various cuts can contribute to numerical issues. Setting `Cuts=1` or `Cuts=0` may help numerics, but it may make the MIP more difficult to solve.

Tolerance values (`FeasibilityTol, OptimalityTol, IntFeasTol`) are generally not helpful for addressing numerical issues. Numerical issues are better handled through model reformulation.

29.6 Instability and the geometry of optimization problems

As we have seen, whenever we solve a problem numerically, we have to accept that the input we provide and the output we obtain may differ from the *theoretical* or *mathematical* solution to the given problem. For example, 0.1, in a computer, will be represented by a number that differs from 0.1 by about 10^{-17}. Thus, a natural thing to worry about is if these small differences may induce large differences in the computed solution.

This is the idea behind the notion of the *Condition Number* for a given problem. While it is true that for most practical optimization problems, small perturbations in the input only induce small perturbations in the final answer to the problem, there are some special situations where this is not the case. These ill behaving problems are called *Ill Conditioned* or *Numerically Unstable*.

This sections aims to show, in the context of linear optimization problems, the most common sources for this behavior, and also how to avoid the behavior altogether. We will review first the problem of solving linear systems with unique solutions, and then move into the more central issue of linear optimization problems, its geometric interpretation, and then describe some of the most common bad cases. We then provide two thought experiments with interactive material to help illustrate the concepts of this section. We conclude with some further thoughts on this topic.

Note that although the notion of the *Condition Number* has received a lot of attention from the academic community, reviewing this literature is beyond the scope of this document. If you want to start looking into this topic, a good entry point can be the Condition Number page at Wikipedia.

The case of linear systems:

Solving linear systems is a very common sub-routine in any MI(QC)P-solver, as we have to solve many linear systems during the full execution of the algorithm.

So, consider that we have a linear system $Ax = b$ with an unique solution (i.e. A is a square matrix with full rank), and you want to evaluate how the solution to the system might change if we perturb the right-hand side b. Since the system has a unique solution, we know that given b, the solution will be $A^{-1}b$, and if we perturb b with ε, the solution will be $A^{-1}(b+\varepsilon)$. A measure for the *relative* change in the solution with respect to the *relative* change in the input would be the ratio

$$\eta(b,\varepsilon) := \frac{\|A^{-1}b\|}{\|A^{-1}(b+\varepsilon)\|} / \frac{\|b\|}{\|b+\varepsilon\|}.$$

Note that the above definition is independent of the magnitudes of b and ε. From there, the *worst* possible ratio would be the result of

$$\kappa(A) := \max_{b,\varepsilon} \eta(b,\varepsilon).$$

This quantity is known as the condition number of the matrix A and is defined as

$$\kappa(A) = \|A\|\|A^{-1}\|.$$

A common interpretation of $\kappa(A) = 10^k$ is that, when solving the system $Ax = b$, you may lose up to k digits of accuracy in x from the accuracy you have in b.

The condition number for the optimal simplex basis in an LP is captured in the `KappaExact` attribute. A very large κ value *might* be an indication that the result might be unstable.

When this is indeed the case, the best advice is to scale the constraint matrix coefficients so that the resulting range of coefficients is small. This transformation will typically reduce the κ

value of the final basis; please refer to the Scaling section for a discussion on how to perform this rescaling, and also for caveats on scaling in general.

The geometry of linear optimization problems

Before showing optimization models that exhibit bad behavior, we first need to understand the *geometry* behind them. Consider a problem of the form

$$\begin{array}{ll} \max & cx \\ s.t. & Ax \leq b. \end{array}$$

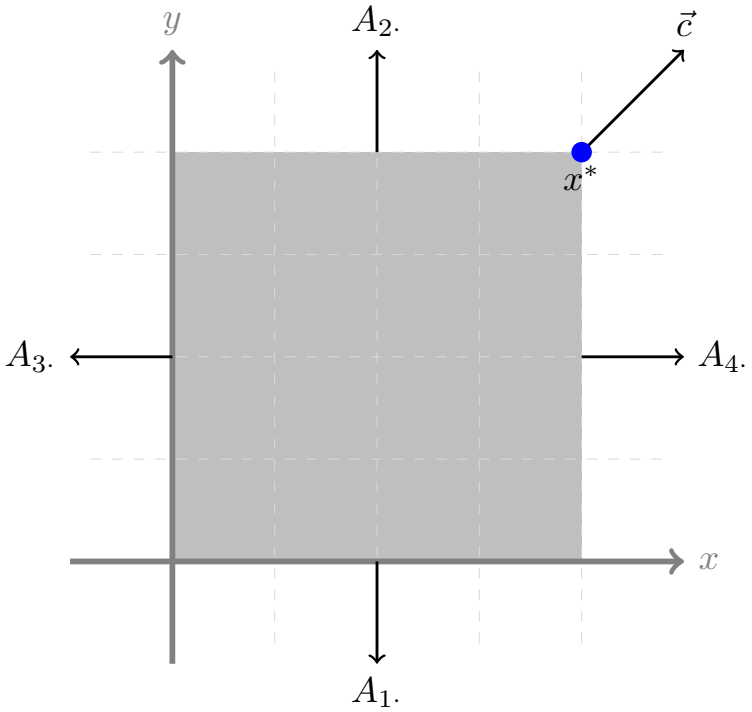
For example:

$$\begin{array}{llll} \max & x+y & \vec{c} = & (1,1) \\ s.t. & -x \leq 0 & A_{1\cdot} = & (-1,0) \\ & x \leq 1 & A_{2\cdot} = & (1,0) \\ & -y \leq 0 & A_{3\cdot} = & (0,-1) \\ & y \leq 1 & A_{4\cdot} = & (0,1). \end{array}$$

Note that if we denote $b^t := (0,1,0,1)$, then the problem can be stated as

$$\max_{x \in \mathbb{R}^2} \{\vec{c}x : Ax \leq b\}.$$

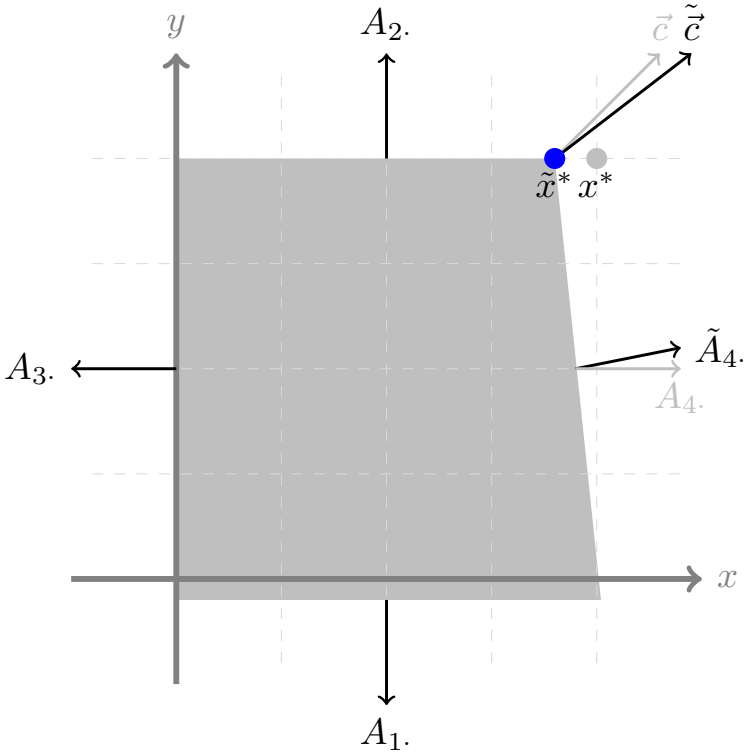
The feasible region, direction of improvement $\vec{c}$, and optimal solution x^* can be depicted as



Note that whenever we move in the direction of $\vec{c}$, the value $\vec{c}x$ increases. Furthermore, since we can not move from x^* to another feasible point with better objective value, we can conclude that x^* is indeed the optimal solution for the problem. Note that x^* is a *corner* point of the feasible

region. This is not a coincidence; you will always find an optimal solution at a corner point if the feasible region is bounded and $\vec{c}$ is not zero. If the objective is zero then all feasible solutions are optimal; we will talk more about zero objectives and their implications later.

To understand how changes in the input data affect the feasible region and the optimal solution, consider a small modification: $\tilde{b}^t = (\varepsilon, 1, 0, 1)$, $\tilde{\vec{c}} = (1+\varepsilon, 1)$, and $\tilde{A}_{4\cdot} = (\varepsilon, 1)$. Then our optimization problem would look like



Note that although we changed the right-hand side, this change had no effect in the optimal solution to the problem, but it did change the feasible region by enlarging the bottom part of the feasible area.

Changing the objective vector tilts the corresponding vector in the graphical representation. This of course also changes the optimal objective value. Perturbing a constraint tilts the graphical representation of the constraint. The change in $A_{4\cdot}$ changes the primal solution itself. The amount of *tilting* constraint undergoes depends on the relative value of the perturbation. For example, although the constraint $x \leq 1$ and the constraint $100x \leq 100$ induce the same feasible region, the perturbation $x + \varepsilon y \leq 1$ will induce more tilting that the perturbation $100x + \varepsilon y \leq 100$.

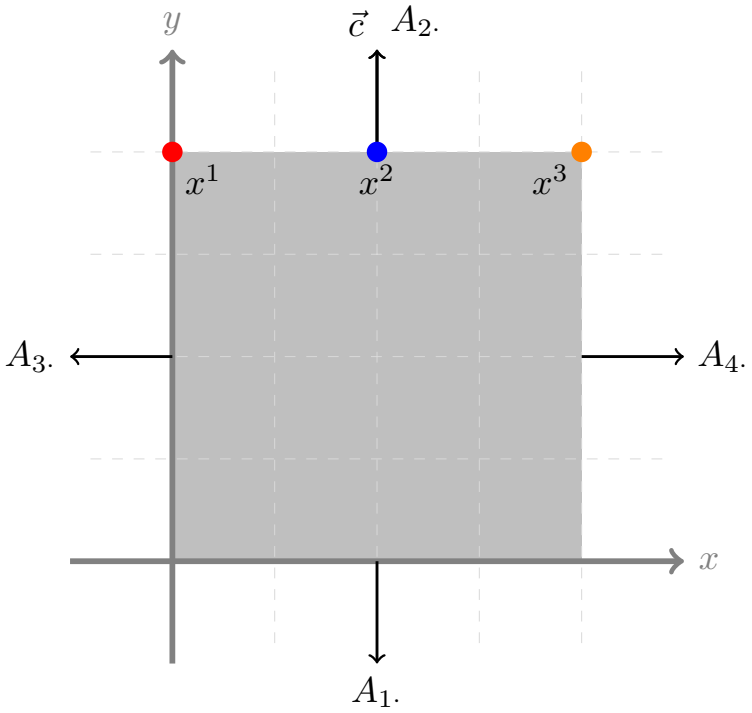
Multiple optimal solutions

A common misconception among beginners in optimization is the idea that optimization problems really have just one solution. Surprisingly, this is typically not true. For many practical problems, the objective (whether it is cost or revenue or ...) is dominated by a handful of variables, while most variables are just there to ensure that the actual *operation* of the solution is possible. Consider a staffing problem, for example, where cost is typically driven by the number of people who work on a given day, not by the specific people.

These kind of situations naturally lead to problems similar to

$$\begin{array}{lrll} \max & y & \vec{c} = & (0,1) \\ s.t. & -x \leq 0 & A_{1\cdot} = & (-1,0) \\ & x \leq 1 & A_{2\cdot} = & (1,0) \\ & -y \leq 0 & A_{3\cdot} = & (0,-1) \\ & y \leq 1 & A_{4\cdot} = & (0,1). \end{array}$$

Graphically this can be depicted as



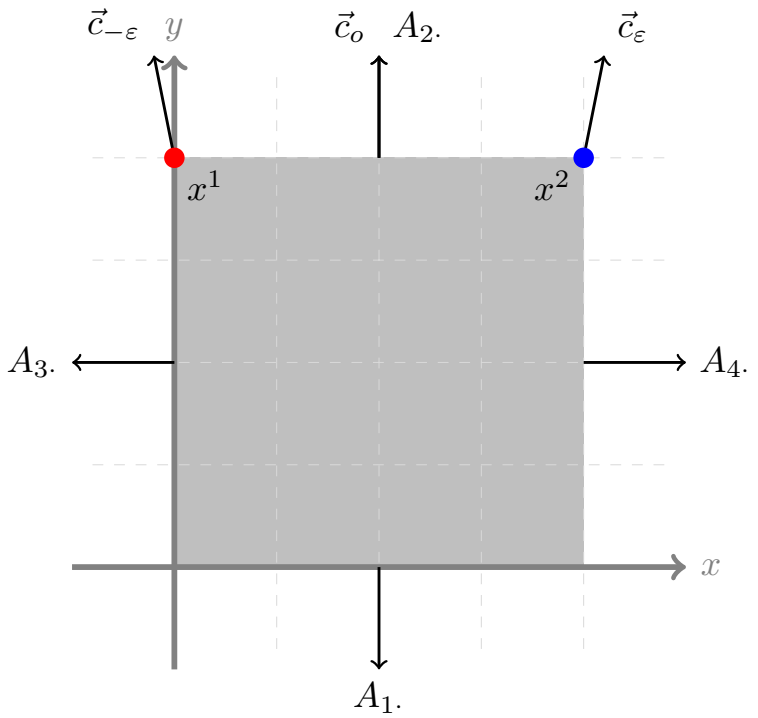
In this situation is clear that x^1, x^3, and all solutions lying on the line between these two points are optimal. The simplex algorithm will return either x^1 or x^3 (and may switch if you change parameters). The barrier algorithm (without crossover) will return x^2. These solutions are all correct; the problem as stated has no reason to prefer one over the other. If you do have a preference, you'll need to state it in your objective function.

Dealing with epsilon-optimal solutions

The previous section considered the case of multiple (true) optimal solutions. What happens when we have several ε-optimal solutions? To be more specific, consider

$$\begin{array}{lrll} \max & \varepsilon x + y & \vec{c} = & (\varepsilon,1) \\ s.t. & -x \leq 0 & A_{1\cdot} = & (-1,0) \\ & x \leq 1 & A_{2\cdot} = & (1,0) \\ & -y \leq 0 & A_{3\cdot} = & (0,-1) \\ & y \leq 1 & A_{4\cdot} = & (0,1). \end{array}$$

Graphically this can be depicted as



If ε is zero, then we are in the situation described before. Note, however, that a small perturbation of the objective vector may lead to either x^1 or x^2 being reported as optimal. And tolerances can play a big role here. If ε is negative, for example, then x^1 would be the mathematically optimal result, but due to the *optimality tolerance*, simplex might conclude that x^2 is optimal. More precisely, if ε is less than the default optimality tolerance of 10^{-6}, then simplex is free to declare either solution optimal (within tolerances).

The above statement is true whenever the *distance* between x^1 and x^2 is not too large. To see this, consider what happens when we change the right-hand side of A_4. from 1 to 10^6. Then the feasible region would then be a very long rectangular box, with vertices $(0, 0)$, $(0, 1)$, $(10^6, 1)$ and $(10^6, 0)$. Perhaps somewhat surprisingly, if ε is below the dual tolerance, simplex may consider $(10^6, 1)$ optimal, even though its objective value is $1 - 10^6\varepsilon$, which can be very relevant in terms of the final objective value.

Note that both situations share one ingredient: The objective function is (almost) parallel to one of the sides of the feasible region. In the first case, this side is relatively short, and thus jumping from x^2 to x^1 translate into a small change in objective value. In the second case, the side almost parallel to the objective function is very long, and now the jump from x^2 to x^1 can have a significant impact on the final objective function.

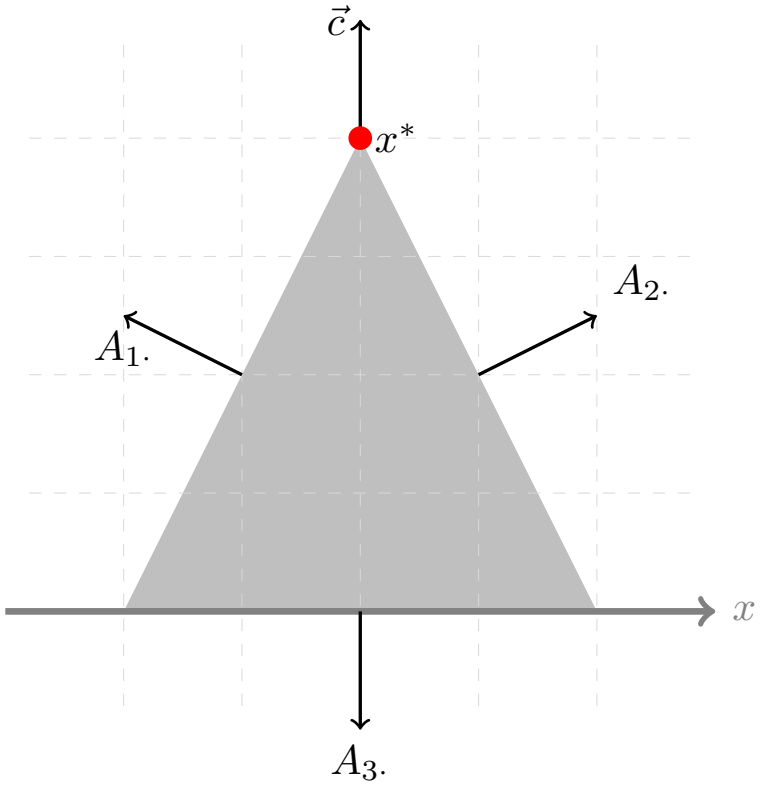
If you take out either of these two ingredients, namely the objective vector being almost parallel to a constraint, or the *edge* induced by this nearly-parallel constraint being very long, then this problem can not arise. For the reasons discussed at the beginning of this section, it is common for the objective function to be close to parallel to one or more constraints. Thus, the best way to avoid this situation is to avoid the second condition. The simplest way to do this is to ensure that the ranges for your variables are not too large. Please refer to the Scaling section for guidance on this.

Thin feasible regions

We now consider another extreme situation that can lead to unexpected results. Consider the problem defined as

$$\begin{array}{lrll} \max & y & \vec{c} = & (0,1) \\ s.t. & -x+\varepsilon y \leq 1 & A_{1\cdot} = & (-1,\varepsilon) \\ & x+\varepsilon y \leq 1 & A_{2\cdot} = & (1,\varepsilon) \\ & -y \leq 0 & A_{3\cdot} = & (0,-1) \end{array}$$

and its graphical representation



It is clear from the graphical representation that the optimal solution for the problem will be at the intersection of constraints $A_{1\cdot}$ with $A_{2\cdot}$; and if we do the algebra, we will get that $x^* = (0, \frac{1}{\varepsilon})$. Also note that as you decrease ε the feasible region stretches upwards, leaving its base unchanged. We will consider the case where ε is a very small, positive number (between 10^{-9} and 10^{-6}).

If we perturb the right-hand side vector b from $(1,1)$ to $(1+\delta,1)$, the new solution will be $\tilde{x}^* = (-\frac{\delta}{2}, \frac{2+\delta}{2\varepsilon})$. To assess the impact of this perturbation, we compute the L_1 distance between this modified solution and the previous solution, which is given by

$$\|x^* - \tilde{x}^*\|_1 = \frac{|\delta|}{2} + \frac{|\delta|}{\varepsilon}$$

This quantity can be either small or very large, depending on the relative magnitude between δ and ε. If δ is much smaller than ε, then this quantity will be small. However, if δ is larger than or even the same order of magnitude as ε, the opposite will be true. Very small perturbations in the input data can lead to big changes in the optimal solution.

A similar issue arises if we perturb $A_{1\cdot}$ to $(-1, \delta)$; the new optimal solution becomes $\tilde{x}^* = (1 - \frac{2\varepsilon}{\varepsilon+\delta}, \frac{2}{\varepsilon+\delta})$. But now, if $\delta = \varepsilon/2$, then the new solution for y will change from $\frac{1}{\varepsilon}$ to $\frac{4}{3\varepsilon}$ (a 33% relative difference). Again, small changes in the input can produce big changes in the optimal solution.

What is driving this bad behavior? The problem is that the optimal point is defined by two constraints that are nearly parallel. The smaller ε is, the closer to parallel the are. When the constraints are so close parallel, small changes in the slopes can lead to big movements in the point where they intersect. Mathematically speaking:

$$\lim_{\varepsilon \to 0^+} \|x^*\| = \infty$$

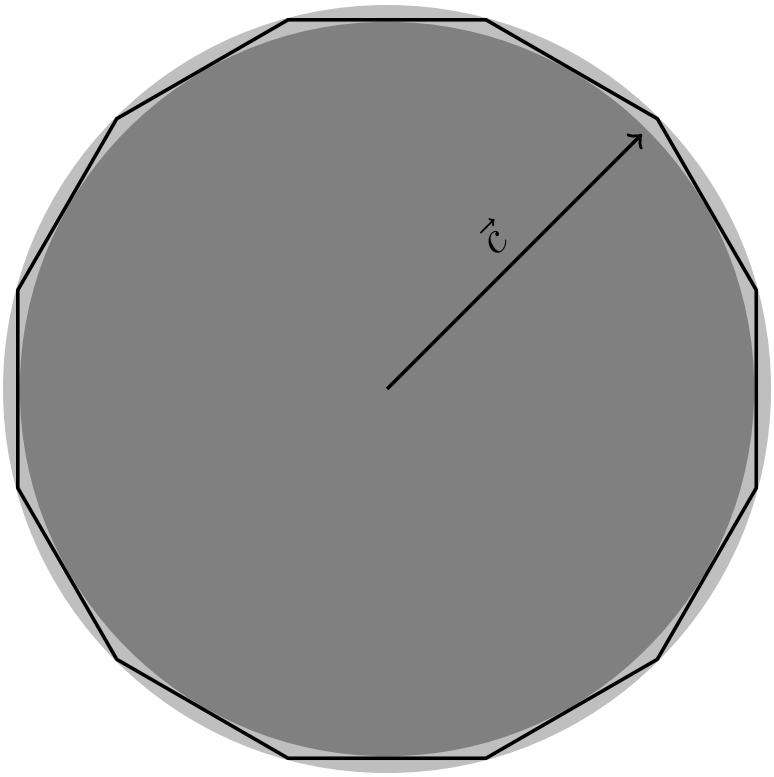
Note however that, if the original problem had an additional variable bound of the form $y \leq 10^4$, then neither of these bad behavior would have been possible. For any ε value smaller than 10^{-4}, the optimal point would be defined by the new constraint and one of the constraints $A_{2\cdot}$ or $A_{1\cdot}$, which would lead again to a well-behaved (i.e. stable) solutions. In summary, this sort of issue can only arise when either the feasible region is either unbounded or very large. See the Scaling section for further guidance on bounding the feasible region.

Optimizing over the circle:

Now we provide our first thought experiment: Consider the problem of optimizing a linear function over the feasible region defined by the constraints

$$\sin(2\pi \tfrac{i}{10^6})x + \cos(2\pi \tfrac{i}{10^6})y \leq 1, \; \forall i \in \{1, \ldots, 10^6\},$$

i.e. the feasible region is essentially a unit circle in $\mathbb{R}^2$. Note that for all objective functions, the corresponding optimal point will be defined by two linear constraints that are very close to be parallel. What will happen to the numerical solution to the problem? Can you guess? The situation is depicted in the figure below:



To perform the experiment, we execute the code circleOpt.py, where we randomly select an objective vector, find the optimal solution to the resulting optimization problem, and compute several relevant quantities:

- The worst *distance* between the reported primal solution, and the theoretical solution to the problem of actually optimizing over a perfect circle, over all previous runs.
- The worst bound violation reported by Gurobi over all previous runs.
- The worst constraint violation reported by Gurobi over all previous runs.
- The worst dual violation reported by Gurobi over all previous runs.
- The number of previous experiments.
- Accumulated number of simplex iterations.
- The κ (KappaExact attribute) value for the current optimal basis.

Sample output is shown below:

```
Added 2 Vars and 1048576 constraints in 19.19 seconds
Errors: 8.65535e-08 0 2.94137e-07 2.77556e-17 Iter 0 10 Kappa 3150.06
Errors: 4.81978e-07 0 3.22359e-07 2.77556e-17 Iter 1 21 Kappa 3009.12
Errors: 4.81978e-07 0 3.4936e-07 1.11022e-16 Iter 2 33 Kappa 2890.58
Errors: 1.53201e-06 0 9.78818e-07 1.11022e-16 Iter 6 79 Kappa 1727.89
Errors: 1.61065e-06 0 8.26005e-07 1.11022e-16 Iter 46 536 Kappa 1880.73
```

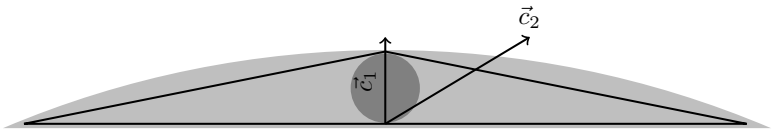
```
Errors: 1.61065e-06 0 8.84782e-07 1.11022e-16 Iter 52 602 Kappa 1817.27
Errors: 1.61065e-06 0 9.4557e-07 1.11022e-16 Iter 54 625 Kappa 1757.96
Errors: 1.69167e-06 0 9.78818e-07 1.11022e-16 Iter 64 742 Kappa 1727.89
Errors: 1.69167e-06 0 3.8268e-07 1.66533e-16 Iter 88 1022 Kappa 2761.99
Errors: 1.69167e-06 0 9.04817e-07 1.66533e-16 Iter 92 1067 Kappa 1797.06
Errors: 1.69167e-06 0 2.94137e-07 2.22045e-16 Iter 94 1089 Kappa 3150.06
Errors: 1.69167e-06 0 3.29612e-07 2.22045e-16 Iter 95 1101 Kappa 2975.84
Errors: 1.69167e-06 0 3.4936e-07 2.22045e-16 Iter 98 1137 Kappa 2890.58
Errors: 1.69167e-06 0 9.25086e-07 2.22045e-16 Iter 99 1147 Kappa 1777.3
Errors: 1.69167e-06 0 9.78818e-07 2.22045e-16 Iter 107 1237 Kappa 1727.89
Errors: 1.69167e-06 0 9.99895e-07 2.22045e-16 Iter 112 1293 Kappa 1709.61
Errors: 1.84851e-06 0 9.78818e-07 2.22045e-16 Iter 132 1523 Kappa 1727.89
Errors: 1.96603e-06 0 9.99895e-07 2.22045e-16 Iter 134 1545 Kappa 1709.61
```

Surprisingly the reported errors are rather small. Why is this? There are at least two contributing factors: the model has a bounded feasible region (in this case the range of both variables is $[-1, 1]$). In addition, the distance from one extreme point (a point at the intersection of two neighboring constraints) to its neighbor is also relatively small, so all ε-optimal solutions are close to each other.

We encourage you to play with this code, perturb some of the input data, and analyze the results. You will see the discrepancies between the theoretical and the numerical optimal solution will be comparable to the sizes of the perturbations.

Optimizing over thin regions:

Now we move to our second thought experiment: Consider a feasible region consisting of a triangle in $\mathbb{R}^2$ with a very wide base and very short height, as depicted here:



Consider the case where the ratio of the base to the height is on the order of 10^5, and that we consider a *nominal* objective function $\vec{c}_1$ as in the figure.

In theory, the optimal solution should be the apex of the triangle, but assume that we randomly perturb both the right-hand side and the objective function with terms in the order of 10^{-6}. What will happen with the numerical solution?

To perform the experiment, we execute the code `thinOpt.py`, where we perform a series of re-optimizations with different perturbations as described above. To be more precise, whenever the new computed solution is further from the mathematical solution by more than it has been in previous trials, we print:

- The new maximum distance among solutions.
- The current iteration.
- The κ (`KappaExact` attribute) value for the current optimal basis.
- The bound violation as reported by Gurobi for the current solution.

- The constraint violation as reported by Gurobi for the current solution.
- The dual violation as reported by Gurobi for the current solution.

Sample output is shown below:

```
New maxdiff 4e+16 Iter 0 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 1 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 2 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 7 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 83 Kappa 3.31072 Violations: 0 0 2.64698e-23
New maxdiff 4e+16 Iter 194 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 1073 Kappa 3.31072 Violations: 0 1.13687e-13 0
New maxdiff 4e+16 Iter 4981 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 19514 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 47117 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 429955 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 852480 Kappa 3.31072 Violations: 0 0 0
```

Results look very different from what we saw in our first test. The distance between the solution to the unperturbed model and the solution to the perturbed one is huge, even from the very first iteration. Also, the κ values are relatively small, and the reported primal, dual, and bound violations are almost zero. So, what happened? Note that when we choose $\vec{c}_1 = (0, 1)$, we are choosing an optimal point where a small tilting of the objective function may move us to another extreme point very far away, and hence the large norm. This is possible because the region is very large and, in principle, without any bounds, i.e. this is related to the case of ε-optimal solutions and very long sides.

Again, we encourage you to play with this example. For example, what would happen if the nominal objective function is $\vec{c}_2 = (1, 0)$?

Stability and convergence

The algorithms used to solve linear programming problems are all forced to make an assumption: that tiny changes to the system (e.g., making a small step in barrier) lead to small changes in the solution. If this is not true (due to ill-conditioning), then the algorithm may jump around in the solution space and have a hard time converging.

Finally, one way to improve the geometry of a problem is by suitably scaling variables and constraints as explained in the Scaling section, and working with bounded feasible sets with *reasonable* ranges for all variables.

29.7 Further reading

- *A Characterization of Stability in Linear Programming*, Stephen M. Robinson, 1977, Operations Research 25-3:435--447.
- *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754), IEEE Computer Society, 1985.
- *What every computer scientist should know about floating-point arithmetic*, David Goldberg, 1991, ACM Computing Surveys (CSUR), 23:5--48.

- *Numerical Computing with IEEE Floating Point Arithmetic*, Michael L. Overton, SIAM, 2001.
- *Practical guidelines for solving difficult linear programs*, Ed Klotz and Alexandra M. Newman, 2013, Surveys in Operations Research and Management Science, 18-1:1--17.
- *Identification, Assessment, and Correction of Ill-Conditioning and Numerical Instability in Linear and Integer Programs*, Ed Klotz, Bridging Data and Decisions, Chapter 3, 54--108.

Source code examples:

Source code for the experiment of optimizing over a circle

```
from gurobipy import *
from math import *
import random
import time
import sys

# Work on a circle defined on a million constraints
t0      = time.time()
n       = 1024 * 1024
m       = Model('Circle Optimization')
X       = m.addVars(2,lb=-2,ub=2)
Wb      = 0
Wc      = 0
Wd      = 0
maxdiff = 0
niter   = 0
margin  = 1.01

m.addConstrs(X[0]*cos((2*pi*i)/n) + X[1]*sin((2*pi*i)/n) <= 1
             for i in range(n))
print('Added 2 Vars and %d constraints in %.2f seconds' %
      (n, time.time()-t0))
m.Params.OutputFlag = 0
m.Params.Presolve   = 0

# Now select random objectives and optimize. Resulting optimal
# solution must be in the circle
for i in range(4096):
  theta=2*pi*random.random()
  a = cos(theta)
  b = sin(theta)
  m.setObjective(X[0] * a + X[1] * b)
  m.optimize()
  niter  += m.IterCount

  # See how far is the solution from the boundary of a circle of
  # radius one, if we minimize (a,b) the optimal solution should be (-a,-b)
  error  = (X[0].X+a)*(X[0].X+a) + (X[1].X+b)*(X[1].X+b)

  # Display most inacurate solution
  if (error > margin * maxdiff  or
      m.BoundVio > margin * Wb  or
```

```
      m.ConstrVio > margin * Wc or
      m.DualVio > margin * Wd     ):
    maxdiff = max(maxdiff, error)
    Wb      = max(Wb, m.BoundVio)
    Wc      = max(Wb, m.ConstrVio)
    Wd      = max(Wd, m.DualVio)
    print('Errors: %g %g %g %g Iter %d %d Kappa %g' %
          (maxdiff, Wb, Wc, Wd, i, niter, m.KappaExact))
    sys.stdout.flush()
```

Source code for the experiment on a thin feasible region

```
from gurobipy import *
import random
import sys

# Test the effect of small perturbations on the optimal solutions
# for a problem with a thin feasible region
rhs = 1e3
m   = Model('Thin line Optimization')
x   = m.addVar(obj=1)
y   = m.addVar(obj=0, lb=-GRB.INFINITY, ub=GRB.INFINITY)
c1  = m.addConstr(  1e-5 * y + 1e-0 * x <= rhs)
c2  = m.addConstr(- 1e-5 * y + 1e-0 * x <= rhs)
m.Params.OutputFlag = 0
m.Params.Presolve   = 0
m.optimize()
xval    = x.X
yval    = y.X
maxdiff = 0
for i in range(1024*1024):
  c1.Rhs = rhs + 2e-6 * random.random()
  c2.Rhs = rhs + 2e-6 * random.random()
  x.Obj  = 1   + 2e-6 * random.random()
  y.Obj  = 0   + 2e-6 * random.random()
  m.optimize()
  x2val = x.X
  y2val = y.X
  error = (xval-x2val)*(xval-x2val) + (yval-y2val)*(yval-y2val)
  if error > 1e-5 + maxdiff:
    print('New maxdiff %g Iter %d Kappa %g Violations: %g %g %g' %
          (error, i, m.KappaExact, m.BoundVio, m.ConstrVio,
           m.DualVio))
    sys.stdout.flush()
    maxdiff = error
```

Source code for the experiment with column scalings

```
import sys
import random
import argparse
import gurobipy as gp

# Use parameters for greater flexibility
parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('-f','--infile', help='Problem File',
```

```
                    default=None, required=True)
parser.add_argument('-s','--scale', help='Scaling Factor',
                    type=float, default=10000.0)
args = parser.parse_args()

# Load input problem
m = gp.read(args.infile)

# Scale domain of all columns randomly in the given domain
for var in m.getVars():
    if var.vtype == gp.GRB.CONTINUOUS:
        scale = random.uniform(args.scale/2.0, args.scale*2.0)
        flip = random.randint(0,3)
        if flip == 0:
            scale = 1.0
        elif flip == 1:
            scale = 1.0/scale
        col = m.getCol(var)
        for i in range(col.size()):
            coeff = col.getCoeff(i)
            row   = col.getConstr(i)
            m.chgCoeff(row, var, coeff*scale)
        var.obj = var.obj*scale
        if var.lb > -gp.GRB.INFINITY:
            var.lb  = var.lb/scale
        if var.ub < gp.GRB.INFINITY:
            var.ub  = var.ub/scale

# Optimize
m.optimize()
if m.Status == gp.GRB.OPTIMAL:
    print('Kappa: %e\n' % m.KappaExact)
```

File Overview – Linux platform

This section briefly describes the purposes of the more important files in the Gurobi distribution on the Linux platform. Note that the list below may not precisely agree with your installation. We've omitted a few less important files. In addition, a few file names depend on the exact version of the Gurobi Optimizer that you installed.

The following files and directories are created in your installation directory (typically `/opt/gurobi1100/linux64` for the 64-bit Linux distribution):

- EULA.pdf - Gurobi End User License Agreement - PDF format
- ReleaseNotes.html - release notes
- bin
  - python3.7 - Python shell
  - grbcluster - Compute Server command-line tool
  - grb_ts - Gurobi Token Server executable
  - grbgetkey - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe - probes system details (typically not used)
  - grbtune - parameter tuning tool
  - gurobi_cl - simple command-line binary
  - gurobi.sh - starts the Gurobi interactive shell
- docs
  - examples - Example Tour - HTML (open index.html in this directory)
  - examples.pdf - Example Tour - PDF
  - refman - Reference Manual - HTML (open index.html in this directory)
  - refman.pdf - Reference Manual - PDF
  - remoteservices - Remote Services Reference Manual - HTML (open index.html in this directory)
  - remoteservices.pdf - Remote Services Reference Manual - PDF
- examples
  - build - Makefile for C, C++, C#, Java, and Python examples
  - c - source code for C examples
  - c# - source code for C# examples
  - c++ - source code for C++ examples

  - data - data files for examples
  - java - source code for Java examples
  - matlab - source code for MATLAB examples
  - python - source code for Python examples
  - R - source code for R examples
  - vb - source code for Visual Basic examples (for Windows)
- include
  - gurobi_c.h - C include file
  - gurobi_c++.h - C++ include file
  - python3.11 - Dummy Python include files (no need to look inside this directory)
- lib
  - gurobi.jar - Java interface
  - gurobi-javadoc.jar - Javadoc documentation for our Java interface
  - gurobi110.netstandard20.dll - .NET Standard 2.0 interface
  - gurobi110.netstandard20.xml - .NET Standard 2.0 interface documentation
  - gurobi.py - Python startup file
  - libgurobi110.so - Gurobi library (symbolic link to current version used by all interfaces)
  - libgurobi.so.11.0.0 - Gurobi native library (used by all interfaces)
  - libgurobi110_light.so - light Gurobi native library (no support for Compute Server or Instant Cloud)
  - libgurobi_c++.a - C++ interface (symbolic link)
  - libgurobi_g++8.5.a - C++ interface
  - libGurobiJni110.so - Java JNI wrapper
  - python3.11 - Python 3.11 files (no need to look inside this directory)
  - rootcert.pem - libcurl distributed certificate
- matlab - Gurobi MATLAB interface
- R - R Gurobi package
- src
  - build - Makefile for Gurobi C++ interface
  - cpp - Source for Gurobi C++ interface

The following files and directories are created in your Remote Services installation (typically `/opt/gurobi_server1100/linux64` for the 64-bit Linux distribution):

- EULA.pdf - Gurobi End User License Agreement - PDF format

- ReleaseNotes.html - release notes
- bin
  - grbcluster - Compute Server command-line tool
  - grb_rs - Gurobi Remote Services executable
  - grb_rs.cnf - Default configuration file for the Remote Services
  - grb_rsm - Gurobi Cluster Manager executable
  - grb_rsm.cnf - Default configuration file for the Cluster Manager
  - grb_rsr - Gurobi Remote Services Router executable
  - grb_rsr.cnf - Default configuration file for the Remote Services Router
  - grb_ts - Gurobi Token Server executable
  - grbgetkey - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe - probes system details (typically not used)
  - grbunzip - Gurobi gzip uncompression tool
  - grbzip - Gurobi gzip compression tool
  - data
    - files - Temporary data files for job execution
    - runtimes - Worker executables for supported versions
- docs
  - remoteservices - Remote Services Reference Manual - HTML (open index.html in this directory)
  - remoteservices.pdf - Remote Services Reference Manual - PDF
- resources - Internal resource files for web UI

File Overview – macOS platform

This section briefly describes the purposes of the more important files in the Gurobi distribution on the macOS platform. Note that the list below may not precisely agree with your installation. We've omitted a few less important files. In addition, a few file names depend on the exact version of the Gurobi Optimizer that you installed.

The following files and directories are created in your installation directory (typically `/Library/gurobi1100/macos_universal2` for the Mac distribution):

- EULA.pdf - Gurobi End User License Agreement - PDF format
- ReleaseNotes.html - release notes
- bin
  - grbcluster - Compute Server command-line tool
  - grb_ts - Gurobi Token Server executable
  - grbgetkey - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe - probes system details (typically not used)
  - grbtune - parameter tuning tool
  - gurobi_cl - simple command-line binary
  - gurobi.sh - starts the Gurobi interactive shell
  - python3.11 - Python binary used for interactive shell
- docs
  - examples - Example Tour - HTML (open index.html in this directory)
  - examples.pdf - Example Tour - PDF
  - refman - Reference Manual - HTML (open index.html in this directory)
  - refman.pdf - Reference Manual - PDF
  - remoteservices - Remote Services Reference Manual - HTML (open index.html in this directory)
  - remoteservices.pdf - Remote Services Reference Manual - PDF
- examples
  - build - Makefile for C, C++, C#, Java, and Python examples
  - c - source code for C examples
  - c# - source code for C# examples
  - c++ - source code for C++ examples

  - data - data files for examples
  - java - source code for Java examples
  - matlab - source code for MATLAB examples
  - python - source code for Python examples
  - R - source code for R examples
  - vb - source code for Visual Basic examples (for Windows)
- include
  - gurobi_c.h - C include file
  - gurobi_c++.h - C++ include file
  - python3.11 - Dummy Python include files (no need to look inside this directory))
- lib
  - gurobi.jar - Java interface
  - gurobi-javadoc.jar - Javadoc documentation for our Java interface
  - gurobi110.netstandard20.dll - .NET Standard 2.0 interface
  - gurobi110.netstandard20.xml - .NET Standard 2.0 interface documentation
  - gurobi.py - Python startup file
  - libGurobiJni110.dylib - Java JNI wrapper
  - libgurobi110.dylib - Gurobi native library (used by all interfaces)
  - libgurobi110_light.dylib - light Gurobi native library (no support for Compute Server or Instant Cloud)
  - libgurobi_c++.a - C++ interface (symbolic link)
  - python3.11 - Python 3.11 interface files, also used by interactive shell (no need to look inside this directory)
  - rootcert.pem - libcurl Distributed certificate
- matlab - Gurobi MATLAB interface
- R - R Gurobi package
- src
  - build - Makefile for Gurobi C++ interface
  - cpp - Source for Gurobi C++ interface

The following files and directories are created in your Remote Services installation (typically `/Library/gurobi_server1100/macos_universal2` for the Mac distribution):

- EULA.pdf - Gurobi End User License Agreement - PDF format
- ReleaseNotes.html - release notes

- bin
  - grbcluster - Compute Server command-line tool
  - grb_rs - Gurobi Remote Services executable
  - grb_rs.cnf - Default configuration file for the Remote Services
  - grb_rsm - Gurobi Cluster Manager executable
  - grb_rsm.cnf - Default configuration file for the Cluster Manager
  - grb_rsr - Gurobi Remote Services Router executable
  - grb_rsr.cnf - Default configuration file for the Remote Services Router
  - grb_ts - Gurobi Token Server executable
  - grbgetkey - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe - probes system details (typically not used)
  - grbunzip - Gurobi gzip uncompression tool
  - grbzip - Gurobi gzip compression tool
  - data
    - files - Temporary data files for job execution
    - runtimes - Worker executables for supported versions
- docs
  - remoteservices - Remote Services Reference Manual - HTML (open index.html in this directory)
  - remoteservices.pdf - Remote Services Reference Manual - PDF
- resources - Internal resource files for web UI

File Overview – Windows platform

This section briefly describes the purposes of the more important files in the Gurobi distribution on the Windows platform. Note that the list below may not precisely agree with your installation. We've omitted a few less important files. In addition, a few file names depend on the exact version of the Gurobi Optimizer that you installed.

The following files and directories are created in your installation directory (typically `c:\gurobi1100\win64` for the 64-bit Windows distribution):

- EULA.rtf - Gurobi End User License Agreement - Microsoft Rich Text Format
- EULA.pdf - Gurobi End User License Agreement - PDF format
- ReleaseNotes.html - release notes
- bin
  - Gurobi110.NET.XML - .NET Framework interface documentation
  - Gurobi110.NET.dll - .NET Framework interface
  - GurobiJni110.dll - Java JNI wrapper
  - grbcluster.exe - Compute Server command-line tool
  - grb_ts.exe - Gurobi Token Server executable
  - grbgetkey.exe - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe.exe - probes system details (typically not used)
  - grbtune.exe - parameter tuning tool
  - gurobi.bat - starts the Gurobi interactive shell
  - gurobi110.dll - Gurobi native DLL (used by all Gurobi interfaces)
  - gurobi110_light.dll - light Gurobi native DLL (no support for Compute Server or Instant Cloud)
  - gurobi_cl.exe - simple command-line binary
  - vslauncher.exe - launcher for Visual Studio Example Solutions
  - vswhere.exe - tool to locate Visual Studio installation
- docs
  - examples - Example Tour - HTML (open index.html in this directory)
  - examples.pdf - Example Tour - PDF
  - refman - Reference Manual - HTML (open index.html in this directory)
  - refman.pdf - Reference Manual - PDF

  - remoteservices - Remote Services Reference Manual - HTML (open index.html in this directory)
  - remoteservices.pdf - Remote Services Reference Manual - PDF
- examples
  - build - Makefile for C, C++, C#, Java, and Python examples
  - c - source code for C examples
  - c# - source code for C# examples
  - c++ - source code for C++ examples
  - data - data files for examples
  - java - source code for Java examples
  - matlab - source code for MATLAB examples
  - python - source code for Python examples
  - R - source code for R examples
  - vb - source code for Visual Basic examples (for Windows)
- include
  - gurobi_c.h - C include file
  - gurobi_c++.h - C++ include file
- lib
  - gurobi.jar - Java interface
  - gurobi-javadoc.jar - Javadoc documentation for our Java interface
  - gurobi110.netstandard20.dll - .NET Standard 2.0 interface
  - gurobi110.netstandard20.xml - .NET Standard 2.0 interface
  - gurobi.py - Python startup file
  - gurobi110.lib - Gurobi library import file
  - gurobi_c++md2017.lib - C++ interface (when using -MD compiler switch with Visual Studio)
  - gurobi_c++mdd2017.lib - C++ interface (when using -MDd compiler switch with Visual Studio)
  - gurobi_c++mt2017.lib - C++ interface (when using -MT compiler switch with Visual Studio)
  - gurobi_c++mtd2017.lib - C++ interface (when using -MTd compiler switch with Visual Studio)
  - rootcert.pem - libcurl Distributed certificate
- matlab - Gurobi MATLAB interface
- R - R Gurobi package

- python311 - Python 3.11 files (no need to look inside this directory)
- src
  - build - Makefile for Gurobi C++ interface
  - cpp - Source for Gurobi C++ interface

The following files and directories are created in your Remote Services installation (typically `c:/gurobi_server1100/win64` for the 64-bit Windows distribution):

- EULA.rtf - Gurobi End User License Agreement - Microsoft Rich Text Format
- EULA.pdf - Gurobi End User License Agreement - PDF format
- ReleaseNotes.html - release notes
- bin
  - grbcluster.exe - Compute Server command-line tool
  - grb_rs.exe - Gurobi Remote Services executable
  - grb_rs.cnf - Default configuration file for the Remote Services
  - grb_rsm.exe - Gurobi Cluster Manager executable
  - grb_rsm.cnf - Default configuration file for the Cluster Manager
  - grb_rsr.exe - Gurobi Remote Services Router executable
  - grb_rsr.cnf - Default configuration file for the Remote Services Router
  - grb_ts.exe - Gurobi Token Server executable
  - grbgetkey.exe - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe.exe - probes system details (typically not used)
  - grbunzip.exe - Gurobi gzip uncompression tool
  - grbzip.exe - Gurobi gzip compression tool
  - grbprobe.exe - probes system details (typically not used)
  - data
    - files - Temporary data files for job execution
    - runtimes - Worker executables for supported versions
- docs
  - remoteservices - Remote Services Reference Manual - HTML (open index.html in this directory)
  - remoteservices.pdf - Remote Services Reference Manual - PDF
- resources - Internal resource files for web UI

Copyright Notices for 3rd Party Libraries

The Gurobi Optimizer use several open-source libraries: `libcurl`, `OpenSSL`, `zlib`, `l8w8jwt`, `mbedtls`, `chillbuff`, `checknum`, and `jsmn`. This section provides license information for these libraries.

The `libcurl` library is distributed under the following license:

```
COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1996 - 2023, Daniel Stenberg, <daniel@haxx.se>, and many
contributors, see the THANKS file.

All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose
with or without fee is hereby granted, provided that the above copyright
notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN
NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE
OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not
be used in advertising or otherwise to promote the sale, use or other dealings
in this Software without prior written authorization of the copyright holder.
```

The `OpenSSL` library is distributed under the Apache License Version 2.0:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
```

otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity
exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications,
including but not limited to software source code, documentation
source, and configuration files.

"Object" form shall mean any form resulting from mechanical
transformation or translation of a Source form, including but
not limited to compiled object code, generated documentation,
and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or
Object form, made available under the License, as indicated by a
copyright notice that is included in or attached to the work
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable

(except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work

by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
names, trademarks, service marks, or product names of the Licensor,
except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
whether in tort (including negligence), contract, or otherwise,
unless required by applicable law (such as deliberate and grossly
negligent acts) or agreed to in writing, shall any Contributor be
liable to You for damages, including any direct, indirect, special,
incidental, or consequential damages of any character arising as a
result of this License or out of the use or inability to use the
Work (including but not limited to damages for loss of goodwill,
work stoppage, computer failure or malfunction, or any and all
other commercial damages or losses), even if such Contributor
has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
the Work or Derivative Works thereof, You may choose to offer,
and charge a fee for, acceptance of support, warranty, indemnity,
or other liability obligations and/or rights consistent with this
License. However, in accepting such obligations, You may act only
on Your own behalf and on Your sole responsibility, not on behalf
of any other Contributor, and only if You agree to indemnify,
defend, and hold each Contributor harmless for any liability
incurred by, or claims asserted against, such Contributor by reason
of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

The `zlib` library is distributed under the

```
Copyright notice:

 (C) 1995-2022 Jean-loup Gailly and Mark Adler

  This software is provided 'as-is', without any express or implied
  warranty.  In no event will the authors be held liable for any damages
  arising from the use of this software.
```

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

The `l8w8jwt` library is distributed under the Apache License Version 2.0:

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

 TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

 1. Definitions.

    "License" shall mean the terms and conditions for use, reproduction,
    and distribution as defined by Sections 1 through 9 of this document.

    "Licensor" shall mean the copyright owner or entity authorized by
    the copyright owner that is granting the License.

    "Legal Entity" shall mean the union of the acting entity and all
    other entities that control, are controlled by, or are under common
    control with that entity. For the purposes of this definition,
    "control" means (i) the power, direct or indirect, to cause the
    direction or management of such entity, whether by contract or
    otherwise, or (ii) ownership of fifty percent (50%) or more of the
    outstanding shares, or (iii) beneficial ownership of such entity.

    "You" (or "Your") shall mean an individual or Legal Entity
    exercising permissions granted by this License.

    "Source" form shall mean the preferred form for making modifications,
    including but not limited to software source code, documentation
    source, and configuration files.

    "Object" form shall mean any form resulting from mechanical
    transformation or translation of a Source form, including but
    not limited to compiled object code, generated documentation,
    and conversions to other media types.

    "Work" shall mean the work of authorship, whether in Source or
    Object form, made available under the License, as indicated by a
    copyright notice that is included in or attached to the work
    (an example is provided in the Appendix below).
```

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices
stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works
that You distribute, all copyright, patent, trademark, and
attribution notices from the Source form of the Work,
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and
wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
names, trademarks, service marks, or product names of the Licensor,
except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

```
8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright 2020 Raphael Beck

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

The `mbedtls` library is distributed under the Apache License Version 2.0:

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/
```

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise

designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of,
publicly display, publicly perform, sublicense, and distribute the
Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made,
use, offer to sell, sell, import, and otherwise transfer the Work,
where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their
Contribution(s) alone or by combination of their Contribution(s)
with the Work to which such Contribution(s) was submitted. If You
institute patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Work
or a Contribution incorporated within the Work constitutes direct
or contributory patent infringement, then any patent licenses
granted to You under this License for that Work shall terminate
as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
Work or Derivative Works thereof in any medium, with or without
modifications, and in Source or Object form, provided that You
meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents

of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,

```
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

The `chillbuff` library is distributed under the Apache License Version 2.0:

```
                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
```

exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications,
including but not limited to software source code, documentation
source, and configuration files.

"Object" form shall mean any form resulting from mechanical
transformation or translation of a Source form, including but
not limited to compiled object code, generated documentation,
and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or
Object form, made available under the License, as indicated by a
copyright notice that is included in or attached to the work
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their

Contribution(s) alone or by combination of their Contribution(s)
with the Work to which such Contribution(s) was submitted. If You
institute patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Work
or a Contribution incorporated within the Work constitutes direct
or contributory patent infringement, then any patent licenses
granted to You under this License for that Work shall terminate
as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
Work or Derivative Works thereof in any medium, with or without
modifications, and in Source or Object form, provided that You
meet the following conditions:

(a) You must give any other recipients of the Work or
Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices
stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works
that You distribute, all copyright, patent, trademark, and
attribution notices from the Source form of the Work,
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and
wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed

```
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright 2019 Raphael Beck

Licensed under the Apache License, Version 2.0 (the "License");
```

```
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

The `checknum` library is distributed under the Apache License Version 2.0:

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
```

of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works
that You distribute, all copyright, patent, trademark, and
attribution notices from the Source form of the Work,
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and
wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
names, trademarks, service marks, or product names of the Licensor,
except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
whether in tort (including negligence), contract, or otherwise,
unless required by applicable law (such as deliberate and grossly

```
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright 2020 Raphael Beck

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

The `jsmn` library is distributed under the MIT License:

```
Copyright (c) 2010 Serge A. Zaitsev

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
```

```
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```