# Contents

1 Introduction

2 C API Overview

2.1 Environment Creation and Destruction
- GRBloadenv
- GRBloadclientenv
- GRBfreeenv
- GRBgetconcurrentenv
- GRBdiscardconcurrentenvs

2.2 Model Creation and Modification
- GRBloadmodel
- GRBnewmodel
- GRBcopymodel
- GRBaddconstr
- GRBaddconstrs
- GRBaddqconstr
- GRBaddqterms
- GRBaddrangeconstr
- GRBaddrangeconstrs
- GRBaddsos
- GRBaddrangeconstrs
- GRBaddvar
- GRBaddvars
- GRBchgcoeffs
- GRBdelconstrs
- GRBdelq
- GRBdelsos
- GRBdelvars
- GRBsetpwlobj
- GRBupdatemodel
- GRBfreemodel
- GRBXaddconstrs
- GRBXaddrangeconstrs
- GRBXaddsos
- GRBXchqcoeffs
- GRBXloadmodel

2.3 Model Solution
- GRBoptimize
- GRBoptimizeasync
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRBcomputeIIS</td>
<td>60</td>
</tr>
<tr>
<td>GRBfeasrelax</td>
<td>60</td>
</tr>
<tr>
<td>GRBfixedmodel</td>
<td>62</td>
</tr>
<tr>
<td>GRBresetmodel</td>
<td>62</td>
</tr>
<tr>
<td>GRBsync</td>
<td>62</td>
</tr>
<tr>
<td>2.4 Model Queries</td>
<td>64</td>
</tr>
<tr>
<td>GRBgetcoeff</td>
<td>64</td>
</tr>
<tr>
<td>GRBgetconstrbyname</td>
<td>64</td>
</tr>
<tr>
<td>GRBgetconstrs</td>
<td>65</td>
</tr>
<tr>
<td>GRBgetenv</td>
<td>65</td>
</tr>
<tr>
<td>GRBgetpwlobj</td>
<td>66</td>
</tr>
<tr>
<td>GRBgetq</td>
<td>66</td>
</tr>
<tr>
<td>GRBgetqconstr</td>
<td>67</td>
</tr>
<tr>
<td>GRBgetso</td>
<td>68</td>
</tr>
<tr>
<td>GRBgetvarbyname</td>
<td>69</td>
</tr>
<tr>
<td>GRBgetvars</td>
<td>69</td>
</tr>
<tr>
<td>GRBXgetconstrs</td>
<td>70</td>
</tr>
<tr>
<td>GRBXgetvars</td>
<td>71</td>
</tr>
<tr>
<td>2.5 Input/Output</td>
<td>73</td>
</tr>
<tr>
<td>GRBreadmodel</td>
<td>73</td>
</tr>
<tr>
<td>GRBread</td>
<td>73</td>
</tr>
<tr>
<td>GRBwrite</td>
<td>74</td>
</tr>
<tr>
<td>2.6 Attribute Management</td>
<td>75</td>
</tr>
<tr>
<td>GRBgetattrinfo</td>
<td>75</td>
</tr>
<tr>
<td>GRBgetintattr</td>
<td>75</td>
</tr>
<tr>
<td>GRBsetintattr</td>
<td>76</td>
</tr>
<tr>
<td>GRBgetintattrelement</td>
<td>76</td>
</tr>
<tr>
<td>GRBsetintattrelement</td>
<td>77</td>
</tr>
<tr>
<td>GRBgetintattrarray</td>
<td>77</td>
</tr>
<tr>
<td>GRBsetintattrarray</td>
<td>78</td>
</tr>
<tr>
<td>GRBgetintattrlist</td>
<td>78</td>
</tr>
<tr>
<td>GRBsetintattrlist</td>
<td>79</td>
</tr>
<tr>
<td>GRBgetdblattr</td>
<td>80</td>
</tr>
<tr>
<td>GRBsetdblattr</td>
<td>80</td>
</tr>
<tr>
<td>GRBgetdblattrelement</td>
<td>81</td>
</tr>
<tr>
<td>GRBsetdblattrelement</td>
<td>81</td>
</tr>
<tr>
<td>GRBgetdblattrarray</td>
<td>82</td>
</tr>
<tr>
<td>GRBsetdblattrarray</td>
<td>82</td>
</tr>
<tr>
<td>GRBgetcharattrelement</td>
<td>83</td>
</tr>
<tr>
<td>GRBsetcharattrelement</td>
<td>83</td>
</tr>
<tr>
<td>GRBgetcharattrarray</td>
<td>84</td>
</tr>
<tr>
<td>GRBsetcharattrarray</td>
<td>85</td>
</tr>
<tr>
<td>GRBgetcharattrlist</td>
<td>85</td>
</tr>
<tr>
<td>GRBsetcharattrlist</td>
<td>86</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>GRBsetcharattrlist</td>
<td>86</td>
</tr>
<tr>
<td>GRBgetstrattr</td>
<td>87</td>
</tr>
<tr>
<td>GRBsetstrattr</td>
<td>88</td>
</tr>
<tr>
<td>GRBgetstrattrelement</td>
<td>88</td>
</tr>
<tr>
<td>GRBsetstrattrelement</td>
<td>89</td>
</tr>
<tr>
<td>GRBgetstrattarray</td>
<td>89</td>
</tr>
<tr>
<td>GRBsetstrattarray</td>
<td>90</td>
</tr>
<tr>
<td>GRBgetstrattrlist</td>
<td>90</td>
</tr>
<tr>
<td>GRBsetstrattrlist</td>
<td>91</td>
</tr>
<tr>
<td>GRBtunemodel</td>
<td>93</td>
</tr>
<tr>
<td>GRBgettuneresult</td>
<td>93</td>
</tr>
<tr>
<td>GRBgetdblparam</td>
<td>94</td>
</tr>
<tr>
<td>GRBgetintparam</td>
<td>94</td>
</tr>
<tr>
<td>GRBgetstrparam</td>
<td>95</td>
</tr>
<tr>
<td>GRBsetdblparam</td>
<td>95</td>
</tr>
<tr>
<td>GRBsetintparam</td>
<td>96</td>
</tr>
<tr>
<td>GRBsetstrparam</td>
<td>96</td>
</tr>
<tr>
<td>GRBgetdblparaminfo</td>
<td>97</td>
</tr>
<tr>
<td>GRBgetintparaminfo</td>
<td>97</td>
</tr>
<tr>
<td>GRBgetstrparaminfo</td>
<td>98</td>
</tr>
<tr>
<td>GRBreadparams</td>
<td>99</td>
</tr>
<tr>
<td>GRBwriteparams</td>
<td>99</td>
</tr>
<tr>
<td>GRBmsg</td>
<td>100</td>
</tr>
<tr>
<td>GRBsetcallbackfunc</td>
<td>100</td>
</tr>
<tr>
<td>GRBgetcallbackfunc</td>
<td>101</td>
</tr>
<tr>
<td>GRBcbget</td>
<td>101</td>
</tr>
<tr>
<td>GRBversion</td>
<td>102</td>
</tr>
<tr>
<td>GRBcbcut</td>
<td>103</td>
</tr>
<tr>
<td>GRBcblazy</td>
<td>104</td>
</tr>
<tr>
<td>GRBcbsolution</td>
<td>105</td>
</tr>
<tr>
<td>GRBterminate</td>
<td>105</td>
</tr>
<tr>
<td>GRBgeterrorormsg</td>
<td>106</td>
</tr>
<tr>
<td>GRBFSolve</td>
<td>107</td>
</tr>
<tr>
<td>GRBBSSolve</td>
<td>107</td>
</tr>
<tr>
<td>GRBBinvColj</td>
<td>108</td>
</tr>
<tr>
<td>GRBBinvRowi</td>
<td>108</td>
</tr>
<tr>
<td>GRBgetBasisHead</td>
<td>109</td>
</tr>
</tbody>
</table>

2.7 Parameter Management and Tuning
GRBtunemodel                                     93
GRBgettuneresult                                 93
GRBgetdblparam                                  94
GRBgetintparam                                  94
GRBgetstrparam                                  95
GRBsetdblparam                                  95
GRBsetintparam                                  96
GRBsetstrparam                                  96
GRBgetdblparaminfo                              97
GRBgetintparaminfo                              97
GRBgetstrparaminfo                              98
GRBreadparams                                  99
GRBwriteparams                                 99

2.8 Monitoring Progress - Logging and Callbacks
GRBmsg                                         100
GRBsetcallbackfunc                              100
GRBgetcallbackfunc                              101
GRBcbget                                       101
GRBversion                                     102

2.9 Modifying Solver Behavior - Callbacks
GRBcbcut                                       103
GRBcblazy                                      104
GRBcbsolution                                 105
GRBterminate                                   105

2.10 Error Handling
GRBgeterrorormsg                               106

2.11 Advanced simplex routines
GRBFSolve                                      107
GRBBSSolve                                     107
GRBBinvColj                                    108
GRBBinvRowi                                    108
GRBgetBasisHead                                109
3 C++ API Overview

3.1 GRBEnv

- GRBEnv()
- GRBEnv::get()
- GRBEnv::getErrorMsg()
- GRBEnv::getParamInfo()
- GRBEnv::message()
- GRBEnv::readParams()
- GRBEnv::resetParams()
- GRBEnv::set()
- GRBEnv::writeParams

3.2 GRBModel

- GRBModel()
- GRBModel::addConstr()
- GRBModel::addConstrs()
- GRBModel::addQConstr()
- GRBModel::addRange()
- GRBModel::addRanges()
- GRBModel::addSOS()
- GRBModel::addVar()
- GRBModel::addVars()
- GRBModel::chgCoeff()
- GRBModel::chgCoeffs()
- GRBModel::computeIIS()
- GRBModel::discardConcurrentEnvs()
- GRBModel::feasRelax()
- GRBModel::fixedModel()
- GRBModel::get()
- GRBModel::getCoeff()
- GRBModel::getCol()
- GRBModel::getConstrByName()
- GRBModel::getConstrs()
- GRBModel::getEnv()
- GRBModel::getObjective()
- GRBModel::getPWLObj()
- GRBModel::getQConstr()
- GRBModel::getQConstrs()
- GRBModel::getRow()
- GRBModel::getSOS()
- GRBModel::getSOSs()
- GRBModel::getTuneResult()
- GRBModel::getVarByName()
- GRBModel::getVars()
- GRBModel::optimize()
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRBModel::optimizeasync()</td>
<td>142</td>
</tr>
<tr>
<td>GRBModel::presolve()</td>
<td>142</td>
</tr>
<tr>
<td>GRBModel::read()</td>
<td>142</td>
</tr>
<tr>
<td>GRBModel::remove()</td>
<td>143</td>
</tr>
<tr>
<td>GRBModel::reset()</td>
<td>143</td>
</tr>
<tr>
<td>GRBModel::setCallback()</td>
<td>144</td>
</tr>
<tr>
<td>GRBModel::set()</td>
<td>144</td>
</tr>
<tr>
<td>GRBModel::setObjective()</td>
<td>147</td>
</tr>
<tr>
<td>GRBModel::setPWLObj()</td>
<td>148</td>
</tr>
<tr>
<td>GRBModel::sync()</td>
<td>149</td>
</tr>
<tr>
<td>GRBModel::terminate()</td>
<td>149</td>
</tr>
<tr>
<td>GRBModel::tune()</td>
<td>150</td>
</tr>
<tr>
<td>GRBModel::update()</td>
<td>150</td>
</tr>
<tr>
<td>GRBModel::write()</td>
<td>150</td>
</tr>
<tr>
<td>GRBVar::get()</td>
<td>151</td>
</tr>
<tr>
<td>GRBVar::sameAs()</td>
<td>152</td>
</tr>
<tr>
<td>GRBVar::set()</td>
<td>152</td>
</tr>
<tr>
<td>GRBConstr::get()</td>
<td>153</td>
</tr>
<tr>
<td>GRBConstr::sameAs()</td>
<td>154</td>
</tr>
<tr>
<td>GRBConstr::set()</td>
<td>154</td>
</tr>
<tr>
<td>GRBQConstr::get()</td>
<td>155</td>
</tr>
<tr>
<td>GRBQConstr::set()</td>
<td>155</td>
</tr>
<tr>
<td>GRBSOS::get()</td>
<td>157</td>
</tr>
<tr>
<td>GRBSOS::set()</td>
<td>157</td>
</tr>
<tr>
<td>GRBExpr::getValue()</td>
<td>158</td>
</tr>
<tr>
<td>GRBExpr::set()</td>
<td>158</td>
</tr>
<tr>
<td>GRBLinExpr()</td>
<td>159</td>
</tr>
<tr>
<td>GRBLinExpr::addTerms()</td>
<td>160</td>
</tr>
<tr>
<td>GRBLinExpr::clear()</td>
<td>160</td>
</tr>
<tr>
<td>GRBLinExpr::getConstant()</td>
<td>160</td>
</tr>
<tr>
<td>GRBLinExpr::getCoeff()</td>
<td>160</td>
</tr>
<tr>
<td>GRBLinExpr::get()</td>
<td>161</td>
</tr>
<tr>
<td>GRBLinExpr::getVar()</td>
<td>161</td>
</tr>
<tr>
<td>GRBLinExpr::operator=</td>
<td>161</td>
</tr>
<tr>
<td>GRBLinExpr::operator+</td>
<td>161</td>
</tr>
<tr>
<td>GRBLinExpr::operator-</td>
<td>162</td>
</tr>
<tr>
<td>GRBLinExpr::operator+=</td>
<td>162</td>
</tr>
<tr>
<td>GRBLinExpr::operator-=</td>
<td>162</td>
</tr>
<tr>
<td>GRBLinExpr::operator*</td>
<td>162</td>
</tr>
<tr>
<td>GRBLinExpr::remove()</td>
<td>163</td>
</tr>
<tr>
<td>GRBLinExpr::size()</td>
<td>163</td>
</tr>
</tbody>
</table>
3.9 GRBQuadExpr

GRBQuadExpr() ................................................. 164
GRBQuadExpr::addTerm() ..................................... 164
GRBQuadExpr::addTerms() ..................................... 165
GRBQuadExpr::clear() .......................................... 166
GRBQuadExpr::getCoeff() ...................................... 166
GRBQuadExpr::getLinExpr() .................................... 166
GRBQuadExpr::getValue() ....................................... 166
GRBQuadExpr::getVar1() ........................................ 166
GRBQuadExpr::getVar2() ........................................ 166
GRBQuadExpr::operator= ...................................... 167
GRBQuadExpr::operator+ ...................................... 167
GRBQuadExpr::operator- ...................................... 167
GRBQuadExpr::operator+= .................................... 167
GRBQuadExpr::operator-= .................................... 167
GRBQuadExpr::operator*= .................................... 167
GRBQuadExpr::remove() ....................................... 168
GRBQuadExpr::size() .......................................... 168

3.10 GRBTempConstr ............................................. 170

3.11 GRBColumn .................................................... 171

GRBColumn() ..................................................... 171
GRBColumn::addTerm() ......................................... 171
GRBColumn::addTerms() ......................................... 171
GRBColumn::clear() ............................................ 171
GRBColumn::getCoeff() ........................................ 171
GRBColumn::getConstr() ....................................... 171
GRBColumn::remove() .......................................... 171
GRBColumn::size() ............................................. 171

3.12 GRBCallback .................................................. 173

GRBCallback() .................................................. 173
GRBCallback::abort() .......................................... 173
GRBCallback::addCut() ........................................ 173
GRBCallback::addLazy() ....................................... 173
GRBCallback::getDoubleInfo() ............................... 174
GRBCallback::getIntInfo() .................................... 174
GRBCallback::getNodeRel() .................................... 174
GRBCallback::getSolution() .................................. 174
GRBCallback::getStringInfo() ............................... 174
GRBCallback::setSolution() .................................. 174

3.13 GRBException ................................................ 176

GRBException() ............................................... 176
GRBException::getErrorCode() ............................ 176
GRBException::getMessage() ............................... 176

3.14 Non-Member Functions .................................... 178

operator== ....................................................... 178
3.15 Attribute Enums

- GRB_CharAttr
- GRB_DoubleAttr
- GRB_IntAttr
- GRB_StringAttr

3.16 Parameter Enums

- GRB_DoubleParam
- GRB_IntParam
- GRB_StringParam

4 Java API Overview

4.1 GRBEnv

- GRBEnv()
- GRBEnv.dispose()
- GRBEnv.get()
- GRBEnv.getErrorMsg()
- GRBEnv.getParamInfo()
- GRBEnv.message()
- GRBEnv.readParams()
- GRBEnv.release()
- GRBEnv.resetParams()
- GRBEnv.set()
- GRBEnv.writeParams()

4.2 GRBModel

- GRBModel()
- GRBModel.addConstr()
- GRBModel.addConstrs()
- GRBModel.addQConstr()
- GRBModel.addRange()
- GRBModel.addRanges()
- GRBModel.addSOS()
- GRBModel.addVar()
- GRBModel.addVars()
- GRBModel.chgCoeff()
- GRBModel.chgCoeffs()
- GRBModel.computeIIS()
- GRBModel.discardConcurrentEnvs()
- GRBModel.dispose()
- GRBModel.feasRelax()
- GRBModel.fixedModel()
4.8 GRBLinExpr

- GRBLinExpr() ......................................................... 251
- GRBLinExpr.add() .................................................. 251
- GRBLinExpr.addConstant() ....................................... 252
- GRBLinExpr.addTerm() ............................................ 252
- GRBLinExpr.addTerms() .......................................... 252
- GRBLinExpr.clear() .............................................. 253
- GRBLinExpr.getConstant() ....................................... 253
- GRBLinExpr.getCoeff() .......................................... 253
- GRBLinExpr.getValue() .......................................... 253
- GRBLinExpr.getVar() ............................................ 253
- GRBLinExpr.multAdd() ........................................... 254
- GRBLinExpr.remove() ............................................ 254
- GRBLinExpr.size() ............................................... 254

4.9 GRBQuadExpr

- GRBQuadExpr() ..................................................... 255
- GRBQuadExpr.add() ................................................ 256
- GRBQuadExpr.addConstant() ..................................... 256
- GRBQuadExpr.addTerm() ......................................... 256
- GRBQuadExpr.addTerms() ........................................ 257
- GRBQuadExpr.clear() ............................................. 258
- GRBQuadExpr.getCoeff() ......................................... 258
- GRBQuadExpr.getLinExpr() ...................................... 258
- GRBQuadExpr.getLinExpr() ...................................... 258
- GRBQuadExpr.getValue() ......................................... 259
- GRBQuadExpr.getVar1() .......................................... 259
- GRBQuadExpr.getVar2() .......................................... 259
- GRBQuadExpr.multAdd() ......................................... 259
- GRBQuadExpr.remove() .......................................... 259
- GRBQuadExpr.size() .............................................. 260

4.10 GRBColumn

- GRBColumn() ....................................................... 261
- GRBColumn.addTerm() ............................................ 261
- GRBColumn.addTerms() .......................................... 261
- GRBColumn.clear() ............................................... 262
- GRBColumn.getCoeff() ........................................... 262
- GRBColumn.getConstr() .......................................... 262
- GRBColumn.remove() ............................................. 263
- GRBColumn.size() ................................................ 263

4.11 GRBCallback

- GRBCallback() ..................................................... 264
- GRBCallback.abort() .............................................. 264
- GRBCallback.addCut() ............................................ 264
- GRBCallback.addLazy() .......................................... 265
- GRBCallback.getDoubleInfo() .................................. 266
- GRBCallback.getIntInfo() ....................................... 266
- GRBCallback.getObjVal() ....................................... 266
- GRBCallback.getIter() .......................................... 266
GRBCallback.getNodeRel() ........................................... 266
GRBCallback.getSolution() ........................................ 267
GRBCallback.getStringInfo() ...................................... 267
GRBCallback.setSolution() ......................................... 268

4.12 GRBException .................................................... 269
GRBException() .................................................... 269
GRBException.getErrorCode() ...................................... 269

4.13 GRB ............................................................ 270
  Constants .................................................................. 270
  GRB.CharAttr ....................................................... 273
  GRB.DoubleAttr .................................................... 273
  GRB.DoubleParam ................................................ 274
  GRB.IntAttr ........................................................ 274
  GRB.IntParam ....................................................... 274
  GRB.StringAttr .................................................... 274
  GRB.StringParam ................................................ 274

5 .NET API Overview .................................................. 275
5.1 GRBEnv ............................................................. 280
  GRBEnv() .................................................................. 280
  GRBEnv.Dispose() .................................................... 281
  GRBEnv.ErrorMsg ................................................... 281
  GRBEnv.Get() ........................................................ 281
  GRBEnv.GetParamInfo() .......................................... 282
  GRBEnv.Message() .................................................. 283
  GRBEnv.ReadParams() .............................................. 283
  GRBEnv.Release() ................................................... 283
  GRBEnv.ResetParams() ............................................ 284
  GRBEnv.Set() ........................................................ 284
  GRBEnv.WriteParams() ............................................ 285

5.2 GRBModel ........................................................... 286
  GRBModel() .......................................................... 286
  GRBModel.AddConstr() ........................................... 286
  GRBModel.AddConstrs() .......................................... 287
  GRBModel.AddQConstr() ......................................... 288
  GRBModel.AddRange() ............................................ 289
  GRBModel.AddRanges() .......................................... 290
  GRBModel.AddSOS() ............................................... 290
  GRBModel.AddVar() ................................................. 290
  GRBModel.AddVars() .............................................. 292
  GRBModel.ChgCoeff() ............................................. 294
  GRBModel.ChgCoeffs() ........................................... 294
  GRBModel.ComputeIIS() .......................................... 295
  GRBModel.DiscardConcurrentEnvs() ......................... 295
  GRBModel.Dispose() ............................................... 295
  GRBModel.FeasRelax() ............................................ 295
  GRBModel.WriteParams() ....................................... 295
GRBModel.FixedModel() .................................................. 297
GRBModel.Get() ........................................................... 297
GRBModel.GetCoeff() ..................................................... 308
GRBModel.GetCol() ........................................................ 309
GRBModel.GetConcurrentEnv() .......................................... 309
GRBModel.GetConstrByName() .......................................... 309
GRBModel.GetConstrs() ................................................... 309
GRBModel.GetEnv() ........................................................ 310
GRBModel.GetObjective() ................................................ 310
GRBModel.GetPWLObj() ................................................... 310
GRBModel.GetQConstr() .................................................. 310
GRBModel.GetQConstrs() ................................................ 311
GRBModel.GetRow() ....................................................... 311
GRBModel.GetSOS() ...................................................... 311
GRBModel.GetSOSs() ..................................................... 312
GRBModel.GetTuneResult() .............................................. 312
GRBModel.GetVarByName() .............................................. 312
GRBModel.GetVars() ..................................................... 312
GRBModel.Optimize() .................................................... 312
GRBModel.Presolve() ..................................................... 313
GRBModel.Read() ......................................................... 313
GRBModel.Remove() ..................................................... 313
GRBModel.Reset() ........................................................ 314
GRBModel.SetCallback() ................................................. 314
GRBModel.Set() .......................................................... 314
GRBModel.SetObjective() ................................................ 325
GRBModel.SetPWLObj() .................................................. 326
GRBModel.Terminate() ................................................... 327
GRBModel.Tune() ........................................................ 327
GRBModel.Update() ....................................................... 327
GRBModel.Write() ........................................................ 327

5.3 GRBVar ................................................................. 328
GRBVar.Get() ............................................................. 328
GRBVar.SameAs() ......................................................... 328
GRBVar.Set() ............................................................ 329

5.4 GRBConstr ............................................................. 330
GRBConstr.Get() ........................................................... 330
GRBConstr.SameAs() ...................................................... 330
GRBConstr.Set() .......................................................... 331

5.5 GRBQConstr ............................................................ 332
GRBQConstr.Get() ........................................................ 332
GRBQConstr.Set() ........................................................ 332

5.6 GRBSOS ................................................................. 334
GRBSOS.Get() ............................................................ 334

5.7 GRBExpr ................................................................. 335
<table>
<thead>
<tr>
<th>Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-</code></td>
<td>352</td>
</tr>
<tr>
<td><code>*</code></td>
<td>353</td>
</tr>
<tr>
<td><code>/</code></td>
<td>355</td>
</tr>
<tr>
<td>implicit cast</td>
<td>355</td>
</tr>
</tbody>
</table>

5.13 GRBCallback 357
- GRBCallback() 357
- GRBCallback.Abort() 357
- GRBCallback.AddCut() 357
- GRBCallback.AddLazy() 358
- GRBCallback.GetDoubleInfo() 359
- GRBCallback.GetIntInfo() 359
- GRBCallback.GetNodeRel() 359
- GRBCallback.GetSolution() 360
- GRBCallback.GetStringInfo() 360
- GRBCallback.SetSolution() 360

5.14 GRBException 362
- GRBException() 362
- GRBException.ErrorCode 362

5.15 GRB 363
- Constants 363
- GRB.CharAttr 366
- GRB.DoubleAttr 366
- GRB.DoubleParam 367
- GRB.IntAttr 367
- GRB.IntParam 367
- GRB.StringAttr 367
- GRB.StringParam 367

6 Python API Overview 368

6.1 Global Functions 372
- models() 372
- multidict() 372
- paramHelp() 372
- quicksum() 373
- read() 373
- readParams() 374
- resetParams() 374
- setParam() 374
- system() 374
- writeParams() 375

6.2 Model 376
- Model() 376
- Model.addConstr() 376
- Model.addQConstr() 377
- Model.addRange() 377
- Model.addSOS() 378
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model.addVar()</td>
<td>378</td>
</tr>
<tr>
<td>Model.chgCoeff()</td>
<td>379</td>
</tr>
<tr>
<td>Model.computeIIS()</td>
<td>379</td>
</tr>
<tr>
<td>Model.copy()</td>
<td>379</td>
</tr>
<tr>
<td>Model.discardConcurrentEnvs()</td>
<td>380</td>
</tr>
<tr>
<td>Model.feasRelaxS()</td>
<td>380</td>
</tr>
<tr>
<td>Model.feasRelax()</td>
<td>381</td>
</tr>
<tr>
<td>Model.fixed()</td>
<td>382</td>
</tr>
<tr>
<td>Model.getAttr()</td>
<td>382</td>
</tr>
<tr>
<td>Model.getCoeff()</td>
<td>383</td>
</tr>
<tr>
<td>Model.getCol()</td>
<td>383</td>
</tr>
<tr>
<td>Model.getConcurrentEnv()</td>
<td>383</td>
</tr>
<tr>
<td>Model.getConstrByName()</td>
<td>384</td>
</tr>
<tr>
<td>Model.getConstrs()</td>
<td>384</td>
</tr>
<tr>
<td>Model.getObjective()</td>
<td>385</td>
</tr>
<tr>
<td>Model.getPWLObj()</td>
<td>385</td>
</tr>
<tr>
<td>Model.getQConstrs()</td>
<td>385</td>
</tr>
<tr>
<td>Model.getParamInfo()</td>
<td>386</td>
</tr>
<tr>
<td>Model.getQCRow()</td>
<td>386</td>
</tr>
<tr>
<td>Model.getRow()</td>
<td>386</td>
</tr>
<tr>
<td>Model.getSOS()</td>
<td>387</td>
</tr>
<tr>
<td>Model.getTuneResult()</td>
<td>387</td>
</tr>
<tr>
<td>Model.getVarByName()</td>
<td>387</td>
</tr>
<tr>
<td>Model.getVars()</td>
<td>388</td>
</tr>
<tr>
<td>Model.message()</td>
<td>388</td>
</tr>
<tr>
<td>Model.optimize()</td>
<td>388</td>
</tr>
<tr>
<td>Model.presolve()</td>
<td>388</td>
</tr>
<tr>
<td>Model.printAttr()</td>
<td>389</td>
</tr>
<tr>
<td>Model.printQuality()</td>
<td>389</td>
</tr>
<tr>
<td>Model.printStats()</td>
<td>389</td>
</tr>
<tr>
<td>Model.cbCut()</td>
<td>390</td>
</tr>
<tr>
<td>Model.cbGet()</td>
<td>390</td>
</tr>
<tr>
<td>Model.cbGetNodeRel()</td>
<td>391</td>
</tr>
<tr>
<td>Model.cbGetSolution()</td>
<td>391</td>
</tr>
<tr>
<td>Model.cbLazy()</td>
<td>392</td>
</tr>
<tr>
<td>Model.cbSetSolution()</td>
<td>392</td>
</tr>
<tr>
<td>Model.read()</td>
<td>393</td>
</tr>
<tr>
<td>Model.relax()</td>
<td>393</td>
</tr>
<tr>
<td>Model.remove()</td>
<td>394</td>
</tr>
<tr>
<td>Model.reset()</td>
<td>394</td>
</tr>
<tr>
<td>Model.setAttr()</td>
<td>394</td>
</tr>
<tr>
<td>Model.setObjective()</td>
<td>394</td>
</tr>
<tr>
<td>Model.setPWLObj()</td>
<td>395</td>
</tr>
<tr>
<td>Model.setParam()</td>
<td>396</td>
</tr>
<tr>
<td>Model.terminate()</td>
<td>397</td>
</tr>
</tbody>
</table>
6.3 Var ............................................. 398
   Var.getAttr() .................................. 398
   Var.sameAs() .................................... 398
   Var.setAttr() .................................... 399
6.4 Constr ...................................... 400
   Constr.getAttr() .................................. 400
   Constr.sameAs() .................................... 400
   Constr.setAttr() .................................... 401
6.5 QConstr .................................... 402
   QConstr.getAttr() .................................. 402
   QConstr.setAttr() .................................... 402
6.6 SOS .......................................... 404
   SOS.getAttr() ...................................... 404
6.7 LinExpr ...................................... 405
   LinExpr() .......................................... 405
   LinExpr.add() ...................................... 406
   LinExpr.addConstant() ............................. 406
   LinExpr.addTerms() .................................. 406
   LinExpr.clear() ..................................... 407
   LinExpr.copy() ...................................... 407
   LinExpr.getConstant() ................................ 407
   LinExpr.getCoeff() ................................... 407
   LinExpr.getValue() ................................... 408
   LinExpr.getVar() ..................................... 408
   LinExpr.remove() ..................................... 408
   LinExpr.size() ....................................... 408
   LinExpr.__eq__() ..................................... 409
   LinExpr.__le__() ..................................... 409
   LinExpr.__ge__() ..................................... 409
6.8 QuadExpr .................................... 410
   QuadExpr() .......................................... 410
   QuadExpr.add() ...................................... 411
   QuadExpr.addConstant() ............................. 411
   QuadExpr.addTerms() .................................. 411
   QuadExpr.clear() ..................................... 412
   QuadExpr.copy() ...................................... 412
   QuadExpr.getCoeff() ................................... 412
   QuadExpr.getLinExpr() ............................... 412
   QuadExpr.getValue() ................................... 413
   QuadExpr.getVar1() ................................... 413
   QuadExpr.getVar2() ................................... 413
   QuadExpr.remove() ..................................... 413

QuadExpr.size() ................................................................. 414
QuadExpr.__eq__() .............................................................. 414
QuadExpr.__le__() ............................................................... 414
QuadExpr.__ge__() ............................................................... 414

6.9 TempConstr ................................................................. 415

6.10 Column ..................................................................... 416
Column() ........................................................................ 416
Column.addTerms() ............................................................ 416
Column.clear() ................................................................. 416
Column.copy() ................................................................. 417
Column.getCoeff() ............................................................ 417
Column.getConstr() .......................................................... 417
Column.remove() .............................................................. 417
Column.size() ................................................................. 418

6.11 Callbacks ................................................................. 419

6.12 GurobiError ............................................................... 420

6.13 Env ................................................................. 421
Env() ........................................................................ 421
Env.resetParams() ............................................................ 421
Env.setParam() ................................................................. 422
Env.writeParams() ........................................................... 422

6.14 GRB ................................................................. 423
Constants .................................................................... 423
GRB.attr ................................................................... 425
GRB.param ................................................................. 425

6.15 tuplelist ................................................................. 426
tuplelist() .................................................................... 426
tuplelist.select() ............................................................ 426
tuplelist.clean() ............................................................ 427

7 MATLAB API Overview ............................................ 428
7.1 Solving models with the Gurobi MATLAB interface ............................... 429
gurobi() .................................................................. 429

7.2 Reading and writing models with the Gurobi MATLAB interface .......... 434
gurobi_read() ................................................................. 434
gurobi_write() ............................................................... 434

7.3 Setting up the Gurobi MATLAB interface ......................................... 435

8 R API Overview .......................................................... 436

9 Attributes ................................................................. 441
9.1 Model Attributes .......................................................... 447
NumConstrs ................................................................ 448
NumVars .................................................................. 448
NumSOS ................................................................ 448
NumQConstrs ............................................................ 448

17
<table>
<thead>
<tr>
<th>Variable Attributes</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumNZs</td>
<td>448</td>
</tr>
<tr>
<td>NumQNZs</td>
<td>448</td>
</tr>
<tr>
<td>NumQCNZs</td>
<td>449</td>
</tr>
<tr>
<td>NumIntVars</td>
<td>449</td>
</tr>
<tr>
<td>NumBinVars</td>
<td>449</td>
</tr>
<tr>
<td>NumPWLObjVars</td>
<td>449</td>
</tr>
<tr>
<td>ModelName</td>
<td>449</td>
</tr>
<tr>
<td>ModelSense</td>
<td>449</td>
</tr>
<tr>
<td>ObjCon</td>
<td>450</td>
</tr>
<tr>
<td>ObjVal</td>
<td>450</td>
</tr>
<tr>
<td>ObjBound</td>
<td>450</td>
</tr>
<tr>
<td>ObjBoundC</td>
<td>450</td>
</tr>
<tr>
<td>MIPGap</td>
<td>450</td>
</tr>
<tr>
<td>Runtime</td>
<td>451</td>
</tr>
<tr>
<td>Status</td>
<td>451</td>
</tr>
<tr>
<td>SolCount</td>
<td>451</td>
</tr>
<tr>
<td>IterCount</td>
<td>451</td>
</tr>
<tr>
<td>BarIterCount</td>
<td>451</td>
</tr>
<tr>
<td>NodeCount</td>
<td>451</td>
</tr>
<tr>
<td>IsMIP</td>
<td>452</td>
</tr>
<tr>
<td>IsQP</td>
<td>452</td>
</tr>
<tr>
<td>IsQCP</td>
<td>452</td>
</tr>
<tr>
<td>IISMinimal</td>
<td>452</td>
</tr>
<tr>
<td>MaxCoeff</td>
<td>452</td>
</tr>
<tr>
<td>MinCoeff</td>
<td>452</td>
</tr>
<tr>
<td>MaxBound</td>
<td>453</td>
</tr>
<tr>
<td>MinBound</td>
<td>453</td>
</tr>
<tr>
<td>MaxObjCoeff</td>
<td>453</td>
</tr>
<tr>
<td>MinObjCoeff</td>
<td>453</td>
</tr>
<tr>
<td>MaxRHS</td>
<td>453</td>
</tr>
<tr>
<td>MinRHS</td>
<td>453</td>
</tr>
<tr>
<td>Kappa</td>
<td>453</td>
</tr>
<tr>
<td>KappaExact</td>
<td>454</td>
</tr>
<tr>
<td>FarkasProof</td>
<td>454</td>
</tr>
<tr>
<td>TuneResultCount</td>
<td>454</td>
</tr>
</tbody>
</table>

9.2 Variable Attributes

<table>
<thead>
<tr>
<th>Variable Attributes</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>454</td>
</tr>
<tr>
<td>UB</td>
<td>455</td>
</tr>
<tr>
<td>Obj</td>
<td>455</td>
</tr>
<tr>
<td>VType</td>
<td>455</td>
</tr>
<tr>
<td>VarName</td>
<td>455</td>
</tr>
<tr>
<td>X</td>
<td>455</td>
</tr>
<tr>
<td>Xn</td>
<td>456</td>
</tr>
<tr>
<td>RC</td>
<td>456</td>
</tr>
<tr>
<td>Start</td>
<td>456</td>
</tr>
</tbody>
</table>
9.3 Linear Constraint Attributes

9.4 SOS Attributes

9.5 Quadratic Constraint Attributes

9.6 Quality Attributes
9.7 Attribute Examples

C Attribute Examples
C++Attribute Examples
C# Attribute Examples
Java Attribute Examples
Python Attribute Examples
Visual Basic Attribute Examples

10 Parameters

10.1 Parameter Guidelines
Continuous Models
MIP Models

10.2 Parameter Descriptions
AggFill
Aggregate
BarConvTol
BarCorrectors
BarHomogeneous
BarOrder
BarQCPConvTol ........................................ 487
BarIterLimit ........................................... 487
BranchDir .............................................. 487
CliqueCuts ............................................. 488
ConcurrentJobs ......................................... 488
ConcurrentMIP ........................................... 489
ConcurrentSettings ..................................... 489
CoverCuts .............................................. 490
Crossover .............................................. 490
CrossoverBasis ......................................... 491
Cutoff .................................................. 491
CutAggPasses ........................................... 491
CutPasses .............................................. 491
Cuts ..................................................... 492
Disconnected .......................................... 492
DisplayInterval ....................................... 492
DistributedMIPJobs ..................................... 493
DualReductions ........................................ 493
FeasibilityTol .......................................... 493
FeasRelaxBigM ......................................... 494
FlowCoverCuts .......................................... 494
FlowPathCuts .......................................... 494
GomoryPasses .......................................... 494
GUBCoverCuts .......................................... 495
Heuristics ............................................. 495
IISMethod .............................................. 495
ImpliedCuts ............................................ 496
ImproveStartGap ....................................... 496
ImproveStartNodes .................................... 496
ImproveStartTime ..................................... 497
InfUnbdInfo ........................................... 497
InputFile .............................................. 497
IntFeasTol ............................................. 498
IterationLimit ......................................... 498
LazyConstraints ....................................... 498
LogFile .................................................. 498
LogToConsole .......................................... 499
MarkowitzTol ......................................... 499
Method .................................................. 499
MinRelNodes ........................................... 500
MIPFocus .............................................. 500
MIPGap .................................................. 501
MIPGapAbs ............................................. 501
MIPSepCuts ............................................. 501
MIQCPMethod .......................................... 501
VarBranch .................................................. 515
WorkerPassword ........................................ 516
WorkerPool ............................................. 516
ZeroHalfCuts ........................................... 516
ZeroObjNodes ......................................... 517

10.3 Parameter Examples .................................. 517
C Parameter Examples .................................. 517
C++ Parameter Examples ............................... 518
C# Parameter Examples ................................. 518
Java Parameter Examples ............................... 519
MATLAB Parameter Examples ......................... 519
Python Parameter Examples ............................ 519
R Parameter Examples .................................. 520
Visual Basic Parameter Examples ..................... 520

11 Optimization Status Codes .......................... 521

12 Callback Codes ...................................... 523

13 Error Codes .......................................... 526

14 Model File Formats .................................. 528
14.1 MPS format ........................................... 528
14.2 REW format ......................................... 532
14.3 LP format ............................................ 532
14.4 RLP format ........................................... 536
14.5 ILP format ............................................ 536
14.6 MST format ........................................... 536
14.7 ORD format .......................................... 537
14.8 BAS format ........................................... 537
14.9 SOL format ........................................... 538
14.10 PRM format ......................................... 538

15 Logging ................................................ 539
15.1 Simplex Logging ..................................... 539
15.2 Barrier Logging ..................................... 540
15.3 Sifting Logging ...................................... 543
15.4 MIP Logging ......................................... 544
15.5 Distributed MIP Logging ............................ 546

16 Gurobi Command-Line Tool ......................... 549
16.1 Solving a Model ..................................... 549
16.2 Gurobi Remote Services and Compute Server Administration .............................. 551

17 Concurrent Optimizer .................................. 553

23
## 18 Parameter Tuning Tool

18.1 Command-Line Tuning ................................................................. 556
18.2 Tuning API .............................................................................. 558

## 19 Gurobi Remote Services

19.1 Setting Up and Administering Gurobi Remote Services ....................... 559
Gurobi Remote Services Parameters .................................................. 559
Firewalls ............................................................................... 560
Administrative Commands ............................................................... 560
Copyright Notice for 3rd Party Library .............................................. 561

## 20 Distributed Parallel Algorithms

20.1 Configuring a Distributed Worker Pool ......................................... 562
20.2 Writing Your Own Distributed Algorithms ...................................... 566
20.3 Distributed Algorithm Considerations ........................................... 566

## 21 Gurobi Compute Server

21.1 Setting Up and Administering a Gurobi Compute Server ...................... 569
21.2 Compute Server Usage ............................................................... 569
Client Configuration ........................................................................ 569
Job Priorities .............................................................................. 570
Performance Considerations on a Wide-Area Network (WAN) ............... 570
Callbacks .................................................................................... 571
Developing for Compute Server ...................................................... 571
Acknowledgement of 3rd Party Icons ............................................... 572
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.

This document also includes information on our Distributed Parallel Algorithms, which allow you to use multiple machines to achieve higher performance, and on Gurobi Compute Server, which allows you to offload Gurobi computations from a set of client machines onto one or more servers.

**Additional Resources**

You can consult the Gurobi Quick Start 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.
Getting Help

If you have a question that is not answered in this document, you can post it to the Gurobi Google Group. If you have a current maintenance contract with us, you can send your question to support@gurobi.com.
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 Quick Start Guide 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.

Models

You can create one or more optimization models within an environment. A model consists of a set of variables, a linear or quadratic 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.

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 or GRBaddqconstr 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/constraint indices are assigned as variables/constraints are added to the model, in a contiguous fashion. In adherence to C language conventions, indices all start at 0.

We often refer to the class of an optimization model. 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’ll sometimes also discuss a special case of QCP, the Second-Order Cone Program (SOCP). If the model contains any integer variables, semi-continuous variables, semi-integer variables, or Special Ordered Set (SOS) constraints, the model is a Mixed Integer Program (MIP). We’ll also sometimes discuss special cases of MIP, including Mixed Integer Linear Programs (MILP), Mixed Integer Quadratic Programs (MIQP), Mixed Integer Quadratically-Constrained Programs (MIQCP), and Mixed Integer Second-Order Cone Programs (MISOCP). The Gurobi Optimizer handles all of these model classes.
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 and QCP models, and the branch-and-cut algorithm to solve mixed integer models. 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 `GRBresetmodel`.

After a MIP model has been solved, you can call `GRBfixedmodel` to compute the associated fixed model. This model is identical to the input model, except that all integer variables are fixed to their values in the MIP solution. In some applications, it is useful to compute information on this continuous version of the MIP model (e.g., dual variables, sensitivity information, etc.).

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 constraint matrix and to the quadratic portion of the objective function.

The constraint matrix can be modified in a few ways. The first is to call GRBchgecoeffs 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 very important item to note about model modifications in the Gurobi optimizer is that they are performed in a lazy fashion, meaning that they don’t actually affect the model until the next call to GRBoptimize or GRBupdatemodel. This approach provides the advantage that the model remains unchanged while you are in the process of making multiple modifications. The downside, of course, is that you have to remember to call GRBupdatemodel() in order to see the effect of your changes.

If you forget to call update, your program won’t crash. The most common symptom of a missing update is an INDEX_OUT_OF_RANGE error, which indicates that the object you are trying to reference isn’t in the model yet.

Managing Parameters

The Gurobi optimizer provides a set of parameters to 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.

**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.
2.1 Environment Creation and Destruction

GRBloadenv

```c
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.

In addition to creating a new environment, 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.

**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.

GRBloadclientenv

```c
int GRBloadclientenv ( GRBenv **envP,
                        const char *logfilename,
                        const char *computeserver,
                        int port,
                        const char *password,
                        int priority,
                        double timeout )
```

Create a client environment on a compute server. Optimization models live within an environment, so this is typically the first Gurobi routine called in an application. This call specifies the compute server on which those optimization models will be solved, as well as the priority of the associated jobs.

In addition to creating a new environment, 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.

**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.
- **logfile**: The name of the log file for this environment. May be NULL (or an empty string), in which case no log file is created.
- **computeserver**: A comma-separated list of Gurobi compute servers. You can refer to compute server machines using their names or their IP addresses.
- **port**: The port number used to connect to the compute server. You should pass a -1 value, which indicates that the default port should be used, unless your server administrator has changed our recommended port settings.
- **password**: The password for gaining access to the specified compute servers. Pass an empty string if no password is required.
- **priority**: 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.
- **timeout**: Job timeout (in seconds). If the job doesn’t reach the front of the queue before the specified timeout, the call will exit with a **JOB_REJECTED** error. Use a negative value to indicate that the call should never timeout.

Example usage:

```c
GRBenv *env;
error = GRBloadclientenv(&env, "gurobi.log",
                        "server1.mydomain.com,server2.mydomain.com",
                        ",", 5, -1.0);
```

---

**GRBfreeenv**

```c
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**

```c
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 \texttt{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 \texttt{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:**

- \texttt{model}: The model for the concurrent environment.
- \texttt{num}: The concurrent environment number.

**Example usage:**

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

\textbf{GRBdiscardconcurrentenvs}

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

Discard concurrent environments for a model.

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

**Arguments:**

- \texttt{model}: The model for the concurrent environment.

**Example usage:**

```c
GRBdiscardconcurrentenvs(model);
```
2.2 Model Creation and Modification

GRBloadmodel

```c
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`, 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]`.

**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:**
```c
/* 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.

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

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

GRBcopymodel

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

Create a copy of an existing model.

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:
```c
GRBmodel *copy = GRBcopymodel(orig);
```

GRBaddconstr

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

Add a new constraint to an existing model. Note that the new constraint won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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:**

```c
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**

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

Add new constraints to an existing model. Note that the new constraints won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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.

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 an existing model. Note that the new constraint won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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: the algorithms Gurobi uses to solve quadratically constrained problems can only handle certain types of quadratic constraints. Constraints of the following forms are always accepted:

- $x^T Q x + q^T x \leq b$, 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)
- $x^T x \leq y z$, where $x$ is a vector of variables, and $y$ and $z$ are non-negative variables (a rotated Second-Order Cone)

If you add a constraint that isn’t in one of these forms (and Gurobi presolve is unable to transform the constraint into one of these forms), you’ll get an error when you try to solve the model. Constraints where the quadratic terms only involve binary variables will always be transformed into one of these forms.
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 $2x_0^2 + x_0 x_1 + x_1^2$, you would call this routine with numqzn=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 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:

```c
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

```c
int GRBaddqpterms ( GRBmodel *model,
                    int numqzn,
                    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 the new quadratic terms won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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, if you wish to input quadratic objective \(2x^2 + xy + y^2\), you would call this routine with \(numqnz=3, qrow[] = \{0, 0, 1\}, qcol[] = \{0, 1, 1\}\), and \(qval[] = \{2, 1, 1\}\).
- **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 \(2x\ y\), you can enter it as a single term, \(2\ x\ y\), or as a pair of terms, \(x\ y\) and \(y\ x\).

**Example usage:**

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

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

Add a new range constraint to an existing 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 the new constraint won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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.

Example usage:

```c
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

```c
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 an existing 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 the new constraints won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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

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

Add new Special Ordered Set (SOS) constraints to an existing model. Note that the new SOS constraints won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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 an existing model. Note that the new variable won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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

```c
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 an existing model. Note that the new variables won’t actually be added until the next call to GRBoptimize or GRBupdatemodel.

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**

```c
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 the changes won’t take effect until the next call to `GRBoptimize` or `GRBupdatemodel`.

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:**

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

**GRBdelconstrs**

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

Delete a list of constraints from an existing model. Note that the constraints won’t actually be removed until the next call to `GRBoptimize` or `GRBupdatemodel`.

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

GRBdelq
```c
int GRBdelq ( GRBmodel *model )
```
Delete all quadratic objective terms from an existing model. Note that the quadratic terms won’t actually be removed until the next call to GRBoptimize or GRBupdatemodel.

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

GRBdelqconstrs
```c
int GRBdelqconstrs ( GRBmodel *model,
                    int numdel,
                    int *ind )
```
Delete a list of quadratic constraints from an existing model. Note that the quadratic constraints won’t actually be removed until the next call to GRBoptimize or GRBupdatemodel.

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

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

Delete a list of Special Ordered Set (SOS) constraints from an existing model. Note that the SOS constraints won’t actually be removed until the next call to GRBoptimize or GRBupdatemodel.

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

GRBdelvars

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

Delete a list of variables from an existing model. Note that the variables won’t actually be removed until the next call to GRBoptimize or GRBupdatemodel.

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

GRBsetpwlobj

```c
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 example, suppose we want to define the function \( f(x) \) shown below:

![Diagram of a piecewise-linear objective function with vertices at (1,1), (3,2), and (5,4).]

The vertices of the function occur at the points (1,1), (3,2) and (5,4), so \texttt{npoints} is 3, \( x \) is \{1, 3, 5\}, and \( y \) is \{1, 2, 4\}. With these arguments 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 set of \( n \) points

\[
\begin{align*}
  x &= \{x_1, \ldots, x_n\}, & y &= \{y_1, \ldots, y_n\}
\end{align*}
\]

define the following piecewise-linear function:

\[
f(v) = \begin{cases} 
  y_1 + \frac{y_2 - y_1}{x_2 - x_1}(v - x_1), & \text{if } v \leq x_1, \\
  y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(v - x_i), & \text{if } v \geq x_i \text{ and } v \leq x_{i+1}, \\
  y_n + \frac{y_n - y_{n-1}}{x_n - x_{n-1}}(v - x_n), & \text{if } v \geq x_n.
\end{cases}
\]

The \( x \) entries must appear in non-decreasing order. Two points can have the same \( x \) coordinate — this can be useful for specifying a discrete jump in the objective function.

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.

Setting a piecewise-linear objective for a variable will set the \texttt{Obj} attribute on that variable to 0. Similarly, setting the \texttt{Obj} attribute will delete the piecewise-linear objective on that variable.

Each variable can have its own piecewise-linear objective function. They must be specified individually, even if multiple variables share the same function.

Note that a new piecewise-linear objective won’t actually be added to the model until the next call to \texttt{GRBoptimize} or \texttt{GRBupdatemodel}.
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:
```c
double x[] = {1, 3, 5};
double y[] = {1, 2, 4};
error = GRBsetpwlobj(model, var, 3, x, y);
```

GRBupdatemodel

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

GRBfreemodel

```c
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:
```c
error = GRBfreemodel(model);
```
The \texttt{size\_t} version of \texttt{GRBaddconstr}. The two arguments that count non-zero values are of type \texttt{size\_t} in this version to support models with more than 2 billion non-zero values.

Add new constraints to an existing model. Note that the new constraints won’t actually be added until the next call to \texttt{GRBoptimize} or \texttt{GRBupdatemodel}.

We recommend that you build your model one constraint at a time (using \texttt{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 \texttt{GRBgeterrormsg}.

**Arguments:**
- \texttt{model}: The model to which the new constraints should be added.
- \texttt{numconstrs}: The number of new constraints to add.
- \texttt{numnz}: The total number of non-zero coefficients in the new constraints.
- \texttt{cbeqg}: 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 \texttt{cbeqg} value, indicating the start position of the non-zeros for that constraint in the \texttt{cind} and \texttt{cval} arrays. This routine requires that the non-zeros for constraint \texttt{i} immediately follow those for constraint \texttt{i-1} in \texttt{cind} and \texttt{cval}. Thus, \texttt{cbeqg[i]} indicates both the index of the first non-zero in constraint \texttt{i} and the end of the non-zeros for constraint \texttt{i-1}. To give an example of how this representation is used, consider a case where \texttt{cbeqg[2]} = 10 and \texttt{cbeqg[3]} = 12. This would indicate that constraint 2 has two non-zero values associated with it. Their variable indices can be found in \texttt{cind[10]} and \texttt{cind[11]}, and the numerical values for those non-zeros can be found in \texttt{cval[10]} and \texttt{cval[11]}.
- \texttt{cind}: Variable indices associated with non-zero values. See the description of the \texttt{cbeqg} argument for more information.
- \texttt{cval}: Numerical values associated with constraint matrix non-zeros. See the description of the \texttt{cbeqg} argument for more information.
- \texttt{sense}: Sense for the new constraints. Options are \texttt{GRB\_LESS\_EQUAL}, \texttt{GRB\_EQUAL}, or \texttt{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

```c
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 an existing 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 the new constraints won’t actually be added until the next call to `GRBoptimize` or `GRBupdatemodel`.

**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

```c
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 an existing model. Note that the new variables won’t actually be added until the next call to `GRBoptimize` or `GRBupdatemodel`.

**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 \texttt{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 \texttt{vind[10]}
and \texttt{vind[11]}, and the numerical values for those non-zeros can be found in \texttt{vval[10]}
and \texttt{vval[11]}.

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

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

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

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

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

\textbf{vtype}: Types for the variables. Options are \texttt{GRB\_CONTINUOUS}, \texttt{GRB\_BINARY}, \texttt{GRB\_INTEGER},
\texttt{GRB\_SEMICONT}, or \texttt{GRB\_SEMIINT}. This argument can be \texttt{NULL}, in which case all variables
are assumed to be continuous.

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

\textbf{GRBXchgcoeffs}

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

The \texttt{size\_t} version of \texttt{GRBchgcoeffs}. The argument that counts non-zero values is of type
\texttt{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 the changes won’t take effect until the next call to \texttt{GRBoptimize} or \texttt{GRBupdatemodel}.

\textbf{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 \texttt{GRBgeterrormsg}.

\textbf{Arguments:}

\begin{itemize}
    \item \texttt{model}: The model to modify.
    \item \texttt{numchgs}: The number of coefficients to modify.
    \item \texttt{cind}: Constraint indices for the coefficients to modify.
    \item \texttt{vind}: Variable indices for the coefficients to modify.
    \item \texttt{val}: The new values for the coefficients. For example, if \texttt{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.
\end{itemize}
Example usage:

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

GRBXloadmodel

```c
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].

**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);
2.3 Model Solution

GRBoptimize

```c
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).

**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:**

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

GRBoptimizeasync

```c
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 GRBsinc 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:
```c
error = GRBoptimizeasync(model);

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

error = GRBsync(model);
```

GRBcomputeIIS

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

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result.

This routine populates the IISConstr, 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.

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, 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:
```c
error = GRBcomputeIIS(model);
```

GRBfeasrelax

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

60
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 \( 2 \times \text{rhspen}[i] \) to the feasibility relaxation objective for relaxobjtype=0, it would contribute \( 2 \times 2 \times \text{rhspen}[i] \) for relaxobjtype=1, and it would contribute \( \text{rhspen}[i] \) 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:**
```c
double penalties[];
error = GRBfeasrelax(model, 0, 0, NULL, NULL, penalties, NULL);
error = GRBoptimize(model);
```

**GRBfixedmodel**

```c
GRBmodel * GRBfixedmodel ( GRBmodel *model )
```

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.

**Return value:**
- This routine returns the computed model. If there is a problem, the routine returns `NULL`.

**Arguments:**
- `model`: The MIP model (with a solution loaded).

**Example usage:**
```c
GRBmodel * fixed = GRBfixedmodel(model);
```

**GRBresetmodel**

```c
int GRBresetmodel ( GRBmodel *model )
```

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.

**Example usage:**
```c
error = GRBresetmodel(model);
```

**GRBsnc**

```c
int GRBsnc ( GRBmodel *model )
```

Wait for a previous asynchronous optimization call to complete.

Calling `GRBoptimizesync` 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 `GRBsnc` call forces the calling program to wait until the asynchronous optimization completes. You **must** call `GRBsnc` before the corresponding model is freed.

The `GRBsnc` 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:**
```c
error = GRBoptimizeasync(model);

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

error = GRBsync(model);
```
2.4 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**

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

**GRBgetconstrbyname**

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

Retrieves a constraint from its name. If multiple 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 constraint should be retrieved.
- `name`: The name of the desired constraint.
- `constrnumP`: Constraint number for a constraint with the indicated name. Returns -1 if no matching name is found.
```c
int GRBgetconstrs ( GRBmodel *model,
    int *numnzP,
    int *cbeg,
    int *cind,
    double *cval,
    int start,
    int len )
```

Retrieve the non-zeros for a set of 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 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[1]` 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.

**GRBgetenv**

```c
GRBenv * GRBgetenv ( GRBmodel *model )
```
Retrieve the environment associated with a model.

**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);
```

---

**GRBgetpwllobj**

```c
int GRBgetpwllobj ( 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 the description of `GRBsetpwllobj` 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 = GRBgetpwllobj(model, var, &npoints, NULL, NULL);
/* ...allocate x and y to hold 'npoints' values... */
error = GRBgetpwllobj(model, var, &npoints, x, y);
```

---

**GRBgetq**

```c
int GRBgetq ( GRBmodel *model,
              int *numqnzP,
              int *qrow,
              int *qcol,
              double *qval )
```

66
Retrieve all quadratic objective terms. The \texttt{qrow}, \texttt{qcol}, and \texttt{qval} arguments must be large enough to hold the result. You can query the \texttt{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 \texttt{GRBgeterrormsg}.

**Arguments:**
- \texttt{model}: The model from which the quadratic objective terms should be retrieved.
- \texttt{numqnzP}: The number of quadratic objective terms retrieved.
- \texttt{qrow}: Row indices associated with quadratic terms. A quadratic term is represented using three values: a pair of indices (stored in \texttt{qrow} and \texttt{qcol}), and a coefficient (stored in \texttt{qval}). The array arguments give the corresponding values for each quadratic term. To give an example, if the quadratic terms in the model are $2x_0^2 + x_0 \times x_1 + x_1^2$, this routine would return \texttt{qrow[]} = \{0, 0, 1\}, \texttt{qcol[]} = \{0, 1, 1\}, and \texttt{qval[]} = \{2, 1, 1\}.
- \texttt{qcol}: Column indices associated with quadratic terms. See the description of the \texttt{qrow} argument for more information.
- \texttt{qval}: Numerical indices associated with quadratic terms. See the description of the \texttt{qrow} argument for more information.

**Example usage:**
```c
int qnz;
int *qrow, *qcol;
double *qval;

error = GRBgetdblattr(model, GRB_DBLE_ATTR_NUMQNZS, &qnz);
/* ...allocate qrow, qcol, qval to hold 'qnz' values... */
error = GRBgetq(model, &qnz, qrow, qcol, qval);
```

**GRBgetqconstr**

```c
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 \texttt{NULL} values for the array arguments. The routine returns the total number of linear and quadratic terms in the specified quadratic constraint in \texttt{numlnzP} and \texttt{numqnzP}, respectively. That allows you to make certain that \texttt{lind}, \texttt{lval}, \texttt{qrow}, \texttt{qcol}, and \texttt{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 $2x_0^2 + x_0 \cdot x_1 + x_1^2$, this routine would return $\star$numqnzP=3, qrow[] = {0, 0, 1}, qcol[] = {0, 1, 1}, and qval[] = {2, 1, 1}.
- **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.

**GRBgetsos**

```c
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**

```c
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**

```c
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.

**GRBXgetconstrs**

```c
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 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**

```c
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.
2.5 Input/Output

GRBreadmodel

```c
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, or .ilp. 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:**

```c
GRBmodel *model;
error = GRBreadmodel(env, "/tmp/model.mps.bz2", &model);
```

GRBread

```c
int GRBread ( GRBmodel *model,
              const char *filename )
```

Import optimization data from a file. This routine is the general entry point for importing data from a file into a model. It can be used to read start vectors for MIP models, basis files for LP models, or parameter settings. The type of data read is determined by the file suffix. File formats are described in the File Format section.

**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 start vector.
- `filename`: The path to the file to be read. The suffix on the file must be either .mst for a MIP start 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:**

```c
error = GRBread(model, "/tmp/model.mst.bz2");
```
GRBwrite

```c
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.

**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, `.ilp` for writing just the IIS associated with an infeasible model (see GRBcomputeIIS for further information), `.sol` for writing the current solution, `.mst` for writing a start vector, `.bas` for writing an LP basis, or `.prm` for writing modified parameter settings. The files can be compressed, so additional suffixes of `.gz`, `.bz2`, or `.7z` are accepted.

**Example usage:**
```c
error = GRBwrite(model, "/tmp/model.rlp.gz");
```
2.6 Attribute Management

**GRBgetattrinfo**

```c
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), or (3) SOS 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:**

```c
int datatype, attrtype, settable;
error = GRBgetattrinfo(model, "ModelName", &datatype, &attrtype, &settable);
```

**GRBgetintattr**

```c
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.

```c
int datatyp...e"ModelName", &datatyp...valueP);
```
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:**
```c
error = GRBgetintattr(model, "NumBinVars", &numbin);
```

### GRBsetintattr

```c
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:**
```c
error = GRBsetintattr(model, "ModelSense", -1);
```

### GRBgetintattrelement

```c
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:**
```c
int first_one;
error = GRBgetintattrelement(model, "VBasis", 0, &first_one);
```

### GRBsetintattrelement

```c
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:**
```c
error = GRBsetintattrelement(model, "VBasis", 0, GRB_BASIC);
```

### GRBgetintattrarray

```c
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 a 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:

```c
int cbasis[NUMCONSTRS];
error = GRBgetintattrarray(model, "CBasis", 0, NUMCONSTRS, cbasis);
```

GRBsetintattrarray

```c
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:

```c
int cbasis[] = {GRB_BASIC, GRB_BASIC, GRB_NONBASIC_LOWER, GRB_BASIC};
error = GRBsetintattrarray(model, "CBasis", 0, 4, cbasis);
```

GRBgetintattrlist

```c
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:**

```c
int desired[] = {0, 2, 4, 6};
int cbasis[4];
error = GRBgetintattrlist(model, "CBasis", 4, desired, cbasis);
```

**GRBsetintattrlist**

```c
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:**

```c
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 GRBget dblattrelement 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

```c
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.
- `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 GRBgetdblattr instead.

**Example usage:**
```c
double first_one;
error = GRBgetdblattrelement(model, "X", 0, &first_one);
```

GRBsetdblattrelement

```c
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:

```c
error = GRBsetdblattrelement(model, "Start", 0, 1.0);
```

**GRBgetdblattrarray**

```c
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:

```c
double lb[NUMVARS];
error = GRBgetdblattrarray(model, "LB", 0, cols, lb);
```

**GRBsetdblattrarray**

```c
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:
```c
double start[] = {1.0, 1.0, 0.0, 1.0};
error = GRBsetdblattrarray(model, "Start", 0, 4, start);
```

**GRBgetdblattrlist**

```c
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:
```c
int desired[] = {0, 2, 4, 6};
double x[4];
error = GRBgetdblattrlist(model, "X", 4, desired, cbasis);
```

**GRBsetdblattrlist**

```c
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:
```c
int change[] = {0, 1, 3};
double start[] = {1.0, 3.0, 2.0, 1.5};
error = GRBsetdblattrlist(model, "Start", 3, change, start);
```

GRBgetcharattrelement

```c
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:
```c
char first_one;
error = GRBgetcharattrelement(model, "VType", 0, &first_one);
```

GRBsetcharattrelement

```c
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

```c
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

```c
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:
```c
char vtypes[] = {GRB_BINARY, GRB_CONTINUOUS, GRB_INTEGER, GRB_BINARY};
error = GRBsetcharattrarray(model, "VType", 0, 4, vtypes);
```

GRBgetcharattrlist

```c
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:
```c
int desired[] = {0, 2, 4, 6};
char vtypes[4];
error = GRBgetcharattrlist(model, "VType", 4, desired, vtypes);
```

GRBsetcharattrlist

```c
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:**
```c
int change[] = {0, 1, 3};
char vtypes[] = {GRB_BINARY, GRB_BINARY, GRB_BINARY};
error = GRBsetcharattrlist(model, "Vtype", 3, change, vtypes);
```

**GRBgetstrattr**

```c
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:**
```c
char *modelname;
error = GRBgetstrattr(model, "ModelName", &modelname);
```
GRBsetstrattr

```c
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.

**Example usage:**

```c
erreur = GRBsetstrattr(model, "ModelName", "Modified name");
```

GRBgetstrattrelement

```c
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 scalar attributes only (i.e., model attributes). To modify a single element of an array attribute, use GRBsetstrattrelement instead.

**Example usage:**

```c
```
char **varname;
error = GRBgetstrattrelement(model, "VarName", 1, varname);

**GRBsetstrattrelement**

```c
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.

**Example usage:**
```c
error = GRBsetstrattrelement(model, "ConstrName", 0, "NewConstr");
```

**GRBgetstrattrarray**

```c
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.

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 **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 GRB-
  readmodel.
- **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.

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:**

```c
int desired[] = {0, 2, 4, 6};
char **varnames[4];
error = GRBgetstrattrarray(model, "VarName", 4, desired, varnames);
```

---

**GRBsetstrattrlist**

```c
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:**

```c
```
int change[] = {0, 1, 3};
char **varnames[] = {"Var0", "Var1", "Var3"};
error = GRBsetstrattrarray(model, "VarName", 3, change, varnames);
2.7 Parameter Management and Tuning

GRBtunemodel

```c
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:**
```c
error = GRBtunemodel(model);
if (error) goto QUIT;

error = GRBgetintattr(model, "TuneResultCount", &nresults);
if (error) goto QUIT;
```

GRBgettuneresult

```c
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:**
```c
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

```c
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:**
```c
error = GRBsetintparam(GRBgetenv(model), "SolutionLimit", 5);
```

GRBsetstrparam

```c
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:**
```c
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 a 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:

```c
error = GRBgetintparaminfo(GRBgetenv(model), "SolutionLimit", &current,
&minAllowedLimit, NULL, &defaultLimit);
```

**GRBgetstrparaminfo**

```c
int GRBgetstrparaminfo ( GBEnv *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:

```c
char defaultval[GRB_MAX_STRLEN];
char currentval[GRB_MAX_STRLEN];
error = GRBgetstrparaminfo(GRBgetenv(model), "LogFile", currentval, defaultval);
```
GRBreadparams

```c
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:**
```c
error = GRBreadparams(env, "/tmp/model.prm.bz2");
```

GRBwriteparams

```c
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:**
```c
error = GRBwriteparams(env, "/tmp/model.prm");
```
2.8 Monitoring Progress - Logging and Callbacks

GRBmsg

```c
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:**
```c
error = GRBmsg(env, "Add this message to the log");
```

GRBsetcallbackfunc

```c
int GRBsetcallbackfunc ( GRBmodel *model,
                          (*cb)(GRBmodel *model, void *cbdata, int
                          where, void *usrddata),
                          void *usrddata )
```

Set up a user callback function. Note that a model can only have a single callback function, so this call will replace an existing callback.

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, note that 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.

**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.
- `usrddata`: An optional pointer to user data that will be passed back to the user callback function each time it is invoked (in the `usrddata` argument).

**Example usage:**
```c
int mycallback(GRBmodel *model, void *cbdata, int where, void *usrddata);
error = GRBsetcallbackfunc(model, mycallback, NULL);
```
**GRBgetcallbackfunc**

```c
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:**

```c
int (*mycallback)(GRBmodel *model, void *cbdata, int where, void *usrdata);
error = GRBgetcallbackfunc(model, &mycallback);
```

**GRBcbget**

```c
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:**

```c
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);
}
```
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:**
```c
int major, minor, technical;
GRBversion(&major, &minor, &technical);
printf("Gurobi library version %d.%d.%d\n", major, minor, technical);
```
2.9 Modifying Solver Behavior - Callbacks

GRBcbcut

```c
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`.

When adding your own cuts, you must set parameter `PreCrush` to value 1. This setting shuts off a few presolve reductions that sometimes prevent cuts on the original model from being applied to the presolved model.

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:**
```c
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

```c
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.

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.
- `laazyrhs`: Right-hand-side value for the new lazy constraint.

**Example usage:**

```c
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

```c
int GRBcbsolution ( void *cbdata,
                    const double *solution )
```

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_MIPNODE` (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.

**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.

**Example usage:**
```c
if (where == GRB_CB_MIPNODE) {
    error = GRBcbsolution(cbdata, solution);
    if (error) return 0;
}
```

GRBterminate

```c
void GRBterminate ( GRBmodel *model )
```

Generate a request to terminate the current optimization. This routine can be called at any time during an optimization. When the optimization stops, the `Status` attribute will be equal to `GRB_INTERRUPTED`.

**Arguments:**
- `model`: The model to terminate.

**Example usage:**
```c
if (time_to_quit)
    GRBterminate(model);
```
2.10 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:**
```c
error = GRBgetintattr(model, "DOES_NOT_EXIST", &attr);
if (error)
    printf("%s\n", GRBgeterrormsg(env));
```
2.11 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**

```c
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**

```c
int GRBBSolve ( GRBmodel *model,
                GRBsvec *b,
                GRBsvec *x )
```

Computes the solution to the linear system \(B^Tx = b\), 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

```c
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

```c
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 \( \text{ind} \) and \( \text{val} \) 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**

```c
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 \( bhead[i]-cols \).
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 Quick Start Guide 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.

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, or GRBSOS). 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.

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 or GRBModel::addQConstr 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. 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’ll sometimes also discuss a special case of QCP, the Second-Order Cone Program (SOCP). If the model contains any integer variables, semi-continuous variables, or Semi-Continuous Set (SOS) constraints, the model is a Mixed Integer Program (MIP). We’ll also sometimes discuss special cases of MIP, including Mixed Integer Linear Programs (MILP), Mixed Integer Quadratic Programs (MIQP), Mixed Integer Quadratically-Constrained Programs (MIQCP), and Mixed Integer Second-Order Cone Programs (MISOCP). The Gurobi Optimizer handles all of these model classes.
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 and QCP models, and the branch-and-cut algorithm to solve mixed integer models. 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, and GRBQConstr 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 input model, except that all integer variables are fixed to their values in the MIP solution. In some applications, it is useful to compute information on this continuous version of the MIP model (e.g., dual variables, sensitivity information, etc.).

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, or GRBModel::get, and modified using GRBVar::set, GRBConstr::set, GRBQConstr::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 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 very important item to note about attribute and model modifications in the Gurobi optimizer is that they are performed in a lazy fashion, meaning that they don’t actually affect the model until the next call to optimize or update on that model object. This approach provides the advantage that the model remains unchanged while you are in the process of making multiple modifications. The downside, of course, is that you have to remember to call update in order to see the effect of your changes.

If you forget to call update, your program won’t crash. The most common symptom of a missing update is a NOT_IN_MODEL exception, which indicates that the object you are trying to reference isn’t in the model yet.

Managing Parameters

The Gurobi optimizer provides a set of parameters to 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 methods on a GRBEnv object (e.g., GRBEnv::set). Current values may also be retrieved with GRBEnv::get. Parameters can be of type int, double, or string. You can also 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.
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 `GRBModel::getEnv` to retrieve the environment associated with a particular model if you want to change a parameter for that model.

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`, `GRBSOS`, 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.

**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.
3.1 GRBEnv

Gurobi environment object. Gurobi models are always associated with an environment. You must create an environment before 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. If the constructor is called with no arguments, no log file will be written for the environment.

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.

Note that the `GRBEnv` constructor 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).

In general, you should aim to create a single Gurobi environment object 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.

---

**GRBEnv**

`GRBEnv()`

Create a Gurobi environment (with logging disabled).

**Return value:**

An environment object (with no associated log file).

**GRBEnv**

`GRBEnv(const string& logFileName)`

Create a Gurobi environment (with logging enabled).

**Arguments:**

- `logFileName`: The desired log file name.

**Return value:**

An environment object.

**GRBEnv**

`GRBEnv(const string& logFileName, const string& computeserver, int port, const string& password, int priority, double timeout)`

Create a client Gurobi environment on a compute server.
Arguments:

logFileName: The name of the log file for this environment. Pass an empty string for no log file.

computeserver: A comma-separated list of Gurobi compute servers. You can refer to compute server machines using their names or their IP addresses.

port: The port number used to connect to the compute server. You should pass a -1 value, which indicates that the default port should be used, unless your server administrator has changed our recommended port settings.

password: The password for gaining access to the specified compute servers. Pass an empty string if no password is required.

priority: 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.

timeout: Job timeout (in seconds). If the job doesn’t reach the front of the queue before the specified timeout, the constructor will throw a JOB_REJECTED exception. Use a negative value to indicate that the call should never timeout.

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.

```cpp
const 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.

```cpp
GRBEnv::getErrorMsg()
```

Query the error message for the most recent exception associated with this environment.

**Return value:**
- The error string.

```cpp
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.

```cpp
void getParamInfo ( GRB_IntParam param,
                    int* valP,
                    int* minP,
                    int* maxP,
                    int* defP )
```

Obtain detailed information about an integer parameter.
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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::getEnv` to retrieve the environment associated with a model if you would like a parameter change to affect that model.

```cpp
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.

```cpp
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.

```cpp
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.

**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.

```cpp
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.
3.2 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, or .ilp. 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.
Arguments:
  model: Model to copy.
Return value:
  New model object. Model is a clone of the input model.

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.
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.

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.

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.

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::addQConstr()

Add a quadratic constraint to a model. Multiple signatures are available.

Important note: the algorithms that Gurobi uses to solve quadratically constrained problems can only handle certain types of quadratic constraints. Constraints of the following forms are always accepted:

- $x^T Q x + q^T x \leq b$, 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)
- $x^T x \leq y z$, where x is a vector of variables, and y and z are non-negative variables (a rotated Second-Order Cone)

If you add a constraint that isn’t in one of these forms (and Gurobi presolve is unable to transform the constraint into one of these forms), you’ll get an error when you try to solve the model. Constraints where the quadratic terms only involve binary variables will always be transformed into one of these forms.
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 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 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.
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.
    names: 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.

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.

```c
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.

```c
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 the change won’t take effect until the next call to GRBModel::optimize or GRBModel::update on that model.

```c
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 the changes won’t take effect until the next call to GRBModel::optimize or GRBModel::update on that model.

```c
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 of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result.

This method populates the IISCONSTR and IISQCONSTR 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.

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

```c
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.

```cpp
void discardConcurrentEnvs()
```

**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 cost per unit violation in the lower bounds, upper bounds, and linear constraints, 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,**
    int clen,**
    const GRBVar** vars,**
    double** lbpen,**
    double** ubpen,**
    const GRBConstr** constr,**
    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.
- **clen**: The length of the list of linear constraints that 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**.
- **constr**: Linear constraints that are allowed to be violated.
- **rhspen**: Penalty for violating a linear constraint. One entry for each variable in argument **constr**.

**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.

    GRBModel  fixedModel ( )

    Return value:
    Fixed model associated with calling object.

GRBModel::get()

Query the value(s) of an attribute. Use this method for scalar model attributes, or for arrays of constraint or variable attributes.

    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.

```c
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.

```c
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.

```c
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.

```c
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 constraint from its name. If multiple constraints have the same name, this method chooses one arbitrarily.

```cpp
GRBConstr getConstrByName ( const string& name )

Arguments:
   name: The name of the desired constraint.

Return value:
   The requested constraint.
```

GRBModel::getConstrs()

Retrieve an array of all constraints in the model.

```cpp
GRBConstr* getConstrs ( )

Return value:
   An array of all constraints in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.
```

GRBModel::getEnv()

Query the environment associated with the model. Note that each model makes its own copy of the environment when it is created. To change parameters for a model, for example, you should use this method to obtain the appropriate environment object.

```cpp
GRBEnv getEnv ( )

Return value:
   The environment for the model.
```

GRBModel::getObjective()

Retrieve a quadratic model objective.

Note that the constant and linear portions of the objective can also be retrieved using the ObjCon and Obj attributes.

```cpp
GRBQuadExpr getObjective ( )

Return value:
   The model 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 the description of setPWLObj for additional information on what the values in x and y mean.

```c
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.

```c
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.

```c
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.

```c
GRBLinExpr getRow ( GRBConstr constr )
```

**Arguments:**
- `constr`: The constraint of interest.

**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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```cpp
void optimizeasync()
```

GRBModel::presolve()

Perform presolve on a model.

```cpp
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 this isn’t 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.

```cpp
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 (for a MIP start), .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 the constraint isn’t actually removed from the model until the next call to GRBModel::optimize or GRBModel::update on that model. |
| Arguments: |
| constr: The linear constraint to remove. |

| void remove ( GRBQConstr qconstr ) |
| Remove a quadratic constraint from the model. Note that the constraint isn’t actually removed from the model until the next call to GRBModel::optimize or GRBModel::update on that model. |
| Arguments: |
| qconstr: The quadratic constraint to remove. |

| void remove ( GRBSOS sos ) |
| Remove an SOS constraint from the model. Note that the SOS isn’t actually removed from the model until the next call to GRBModel::optimize or GRBModel::update on that model. |
| Arguments: |
| sos: The SOS constraint to remove. |

| void remove ( GRBVar var ) |
| Remove a variable from the model. Note that the variable isn’t actually removed from the model until the next call to GRBModel::optimize or GRBModel::update on that model. |
| Arguments: |
| var: The variable to remove. |

GRBModel::reset()

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

| void reset ( ) |
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.

```c
void setCallback ( GRBCallback* cb )
```

GRBModel::set()

Set the value(s) of an attribute. Use this method for scalar model attributes and for arrays of constraint or variable attributes.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

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.

```c
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.

```c
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::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 example, suppose we want to define the function \( f(x) \) shown below:

![Piecewise-linear objective function diagram](image)

The vertices of the function occur at the points \((1, 1)\), \((3, 2)\) and \((5, 4)\), so \( \text{npoints} \) is 3, \( x \) is \{1, 3, 5\}, and \( y \) is \{1, 2, 4\}. With these arguments 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 set of \( n \) points

\[
  x = \{x_1, \ldots, x_n\}, \quad y = \{y_1, \ldots, y_n\}
\]

define the following piecewise-linear function:

\[
f(v) = \begin{cases} 
  y_1 + \frac{y_2-y_1}{x_2-x_1}(v-x_1), & \text{if } v \leq x_1, \\
  y_i + \frac{y_{i+1}-y_i}{x_{i+1}-x_i}(v-x_i), & \text{if } v \geq x_i \text{ and } v \leq x_{i+1}, \\
  y_n + \frac{y_n-y_{n-1}}{x_n-x_{n-1}}(v-x_n), & \text{if } v \geq x_n.
\end{cases}
\]

The \( x \) entries must appear in non-decreasing order. Two points can have the same \( x \) coordinate — this can be useful for specifying a discrete jump in the objective function.
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.

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.

Each variable can have its own piecewise-linear objective function. They must be specified individually, even if multiple variables share the same function.

```c
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::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.

```c
void sync ( )
```

**GRBModel::terminate()**

Generate a request to terminate the current optimization. This method can be called at any time during an optimization.

```c
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.

```cpp
void tune ()
```

GRBModel::update()

Process any pending model modifications.

```cpp
void update ()
```

GRBModel::write()

This method is the general entry point for writing model data to a file. It can be used to write optimization models, IIS submodels, solutions, basis vectors, MIP start vectors, or parameter settings. The type of file written is determined by the file suffix. File formats are described in the File Format section.

```cpp
void write ( const string& filename )
```

**Arguments:**

- **filename**: Name of the file to write. The file type is encoded in the file name suffix. Valid suffixes for writing the model itself are .mps, .rew, .lp, or .rlp. An IIS can be written by using an .ilp suffix. Use .sol for a solution file, .mst for a MIP start, .bas for a basis file, or .prm for a parameter file. The suffix may optionally be followed by .gz, .bz2, or .7z, which produces a compressed result.
3.3 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::sameAs()

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.
3.4 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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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::sameAs()

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.
3.5 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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```c
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.

```c
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.
3.6 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.
3.7 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.

```cpp
double getValue ()
```

Return value:

Value of the expression for the current solution.
3.8 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., \( expr = 1 \times x + 2 \times y \)), or from other expressions (e.g., \( expr2 = 2 \times expr1 + x \), or \( expr3 = expr1 + 2 \times expr2 \)). 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.

Individual terms in a linear expression can be queried using the getVar, getCoef, 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.

```cpp
def addTerms(coeffs, vars, 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.

```cpp
def clear()
```

**GRBLinExpr::getConstant()**
Retrieve the constant term from a linear expression.

```cpp
def getConstant()
    # Return value:
    Constant from expression.
```

**GRBLinExpr::getCoeff()**
Retrieve the coefficient from a single term of the expression.

```cpp
def getCoeff(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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
void operator+= ( const GRBLinExpr& expr )
```

**Arguments:**
- `expr`: Expression to add.

GRBLinExpr::operator-=

Subtract an expression from the invoking expression.

```cpp
void operator-= ( const GRBLinExpr& expr )
```

**Arguments:**
- `expr`: Expression to subtract.

GRBLinExpr::operator*=

Multiply the invoking expression by a constant.

```cpp
void operator*= ( double multiplier )
```

**Arguments:**
- `multiplier`: Constant multiplier.
**GRBLinExpr::remove()**

Remove a term from a linear expression.

```cpp
void remove ( int i )
```

Remove the term stored at index $i$ of the expression.

**Arguments:**

- $i$: The index of the term to be removed.

```cpp
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).

```cpp
unsigned int size ( )
```

**Return value:**

Number of terms in the expression.
3.9 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 \( x \times x \) is a GRBQuadExpr object. Expressions can be built from constants (e.g., \( expr = 0 \)), variables (e.g., \( expr = 1 \times x \times x + 2 \times x \times y \)), or from other expressions (e.g., \( expr2 = 2 \times expr1 + x \times x \), or \( expr3 = expr1 + 2 \times expr2 \)). You can also modify existing expressions (e.g., \( expr += x \times x \), 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 \( expr = expr + x \times x \) in a loop. It will lead to runtimes that are quadratic in the number of terms in the expression.

- Using \( expr += x \times x \) (or \( expr -= x \times x \)) is much more efficient than \( expr = expr + x \times x \). 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.

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.

<table>
<thead>
<tr>
<th>Create a constant quadratic expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
</tr>
<tr>
<td>constant (optional): Constant value for expression.</td>
</tr>
</tbody>
</table>
**Return value:**
A constant expression object.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```c
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.

```c
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.

```c
void clear ( )
```

GRBQuadExpr::getCoeff()

Retrieve the coefficient from a single quadratic term of the quadratic expression.

```c
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.

```cpp
GRBLinExpr getLinExpr();
```

**Return value:** Linear expression associated with the quadratic expression.

**GRBQuadExpr::getValue()**

Compute the value of a quadratic expression for the current solution.

```cpp
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.

```cpp
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.

```cpp
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.

<table>
<thead>
<tr>
<th>GRBQuadExpr operator=( const GRBQuadExpr&amp; rhs )</th>
</tr>
</thead>
</table>

**Arguments:**
- `rhs`: Source quadratic expression.

**Return value:**
- New quadratic expression object.

GRBQuadExpr::operator+
Add one expression into another, producing a result expression.

<table>
<thead>
<tr>
<th>GRBQuadExpr operator+ ( const GRBQuadExpr&amp; rhs )</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>GRBQuadExpr operator- ( const GRBQuadExpr&amp; rhs )</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>void operator+= ( const GRBQuadExpr&amp; expr )</th>
</tr>
</thead>
</table>

**Arguments:**
- `expr`: Expression to add.
### GRBQuadExpr::operator-=
Subtract an expression from the invoking expression.

```cpp
void operator-=( const GRBQuadExpr& expr )
```

**Arguments:**
- `expr`: Expression to subtract.

### GRBQuadExpr::operator*=
Multiply the invoking expression by a constant.

```cpp
void operator*=( double multiplier )
```

**Arguments:**
- `multiplier`: Constant multiplier.

### GRBQuadExpr::remove()
Remove a quadratic term from a quadratic expression.

```cpp
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.

### GRBQuadExpr::remove(GRBVar var)
Remove all quadratic terms associated with variable `var` from the quadratic expression.

```cpp
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.

```cpp
unsigned int size ( )
```

**Return value:**
- Number of quadratic terms in the expression.
3.10 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:

```c++
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 method GRBModel::addConstr or GRBModel::addQConstr.
3.11 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.

```
<table>
<thead>
<tr>
<th>GRBColumn &amp; GRBColumn ( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return value:</td>
</tr>
<tr>
<td>An empty column object.</td>
</tr>
</tbody>
</table>
```

**GRBColumn::addTerm()**

Add a single term into a column.

```
<p>| void addTerm ( double coeff, |</p>
<table>
<thead>
<tr>
<th>GRBConstr constr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
</tr>
<tr>
<td>coeff: Coefficient for new term.</td>
</tr>
<tr>
<td>constr: Constraint for new term.</td>
</tr>
</tbody>
</table>
```

**GRBColumn::addTerms()**

Add new terms into a column.

```
<p>| void addTerms ( const double* coeff, |
| const GRBConstr* constrs, |</p>
<table>
<thead>
<tr>
<th>int count )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a list of terms into a column.</td>
</tr>
<tr>
<td>Arguments:</td>
</tr>
<tr>
<td>coeff: Coefficients for new terms.</td>
</tr>
<tr>
<td>constrs: Constraints for new terms.</td>
</tr>
<tr>
<td>count: Number of terms to add to the column.</td>
</tr>
</tbody>
</table>
```

**GRBColumn::clear()**

Remove all terms from a column.

```
| void clear ( ) |
```
**GRBColumn::getCoeff()**

Retrieve the coefficient from a single term in the column.

```cpp
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.

```cpp
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.

```cpp
void remove ( int i )
```

Remove the term stored at index i of the column.

**Arguments:**
- i: The index of the term to be removed.

```cpp
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.

```cpp
unsigned int size ( )
```

**Return value:**
Number of terms in the column.
### 3.12 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`, 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`). More sophisticated MIP callbacks might use `GRBCallback::getSolution` to retrieve values from the solution to the current node relaxation, and then use `GRBCallback::addCut` or `GRBCallback::addLazy` to add a constraint to cut off that relaxation solution, or `GRBCallback::setSolution` to import a heuristic solution built from the relaxation solution.

When solving a model using multiple threads, note that 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.

You can look at the `callback_c++.cpp` example for details of how to use Gurobi callbacks.

---

**GRBCallback()**

Callback constructor.

```c++
GRBCallback GRBCallback() {
    Return value:
    A callback object.
}
```

**GRBCallback::abort()**

Abort optimization. When the optimization stops, the `Status` attribute will be equal to `GRB_INTERRUPTED`.

```c++
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`.  

---

173
When adding your own cuts, you must set parameter PreCrush to value 1. This setting shuts off a few presolve reductions that sometimes prevent cuts on the original model from being applied to the presolved model.

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.

```cpp
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.
```

```cpp
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.

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.

```cpp
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.

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::setSolution()**

Import solution values for a heuristic solution. Only available when the `where` member variable is equal to `GRB_CB_MIPNODE`.

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. At the end of 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.

```c
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.

```c
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.
3.13 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.

custom string getMessage ()

Return value:

The error message associated with the exception.
### 3.14 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`. 

179
**operator+**

Overloaded operator on expression objects.

```cpp
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.

```cpp
GRBLinExpr operator+ ( const GRBLinExpr& expr )
```

Allow plus sign to be used before an expression.

**Arguments:**
- `expr`: Expression.

**Return value:**
Result expression.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.
3.15 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.
3.16 Parameter Enums

These enums are used to get or set Gurobi parameters. The complete list of parameters can be found in the Parameters section.

GRB_DoubleParam

This enum is used to get or set double-valued parameters (through 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 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 GRBEnv::get or GRBEnv::set). Please refer to the Parameters section to see a list of all int parameters and their functions.
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 are new to the Gurobi Optimizer, we suggest that you start with the Quick Start Guide 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.

**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, or GRBSOS). 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.

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 or GRBModel.addQConstr 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. 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’ll sometimes also discuss a special case of QCP, the *Second-Order Cone Program (SOCP)*. If the model contains any integer variables, semi-continuous variables, semi-integer variables, or Special Ordered Set (SOS) constraints, the model is a *Mixed Integer Program (MIP)*. We’ll also sometimes discuss special cases of MIP, including *Mixed Integer Linear Programs (MILP)*, *Mixed Integer Quadratic Programs (MIQP)*, *Mixed Integer Quadratically-Constrained Programs (MIQCP)*, and *Mixed Integer Second-Order Cone Programs (MISOCP)*. The Gurobi Optimizer handles all of these model classes.
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 and QCP models, and the branch-and-cut algorithm to solve mixed integer models. 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, and GRBQConstr 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 input model, except that all integer variables are fixed to their values in the MIP solution. In some applications, it is useful to compute information on this continuous version of the MIP model (e.g., dual variables, sensitivity information, etc.).

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, or GRBModel.get, and modified using GRBVar.set, GRBConstr.set, GRBQConstr.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 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 very important item to note about attribute and model modifications in the Gurobi optimizer is that they are performed in a lazy fashion, meaning that they don’t actually affect the model until the next call to optimize or update on that model object. This approach provides the advantage that the model remains unchanged while you are in the process of making multiple modifications. The downside, of course, is that you have to remember to call update in order to see the effect of your changes.

If you forget to call update, your program won’t crash. The most common symptom of a missing update is a NOT_IN_MODEL exception, which indicates that the object you are trying to reference isn’t in the model yet.

Managing Parameters

The Gurobi optimizer provides a set of parameters to 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 GRBEnv.set. Current values may also be retrieved with GRBEnv.get. Parameters can be of type int, double, or String. You can also 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.
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 `GRBModel.getEnv` to retrieve the environment associated with a particular model if you want to change a parameter for that model.

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 the 32-bit Gurobi native library with a 32-bit Java JVM, and similarly the 64-bit Gurobi native library with a 64-bit Java JVM.

**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.

**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.
4.1 GRBEnv

Gurobi environment object. Gurobi models are always associated with an environment. You must create an environment before you can 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()

Environment constructor.

Constructor for GRBEnv object. If the constructor is called with no arguments, no log file will be written for the environment.

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.

Note that the GRBEnv constructor 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).

In general, you should aim to create a single Gurobi environment object 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.

GRBEnv  GRBEnv ( )

Create a Gurobi environment (with logging disabled).

Return value:

An environment object (with no associated log file).

GRBEnv  GRBEnv ( String logFileName )

Create a Gurobi environment (with logging enabled).

Arguments:

logFileName: The desired log file name.

Return value:

An environment object.
GRBEnv GRBEnv ( String logFileName,
    String computeserver,
    int port,
    String password,
    int priority,
    double timeout )

Create a client Gurobi environment on a compute server.

Arguments:

  logFileName: The name of the log file for this environment. Pass an empty string for no
    log file.
  computeserver: A comma-separated list of Gurobi compute servers. You can refer to
    compute server machines using their names or their IP addresses.
  port: The port number used to connect to the compute server. You should pass a -1 value,
    which indicates that the default port should be used, unless your server administrator has
    changed our recommended port settings.
  password: The password for gaining access to the specified compute servers. Pass an empty
    string if no password is required.
  priority: 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.
  timeout: Job timeout (in seconds). If the job doesn’t reach the front of the queue before the
    specified timeout, the constructor will throw a JOB_REJECTED exception. Use a negative
    value to indicate that the call should never timeout.

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.

```java
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.

```java
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.

```java
String getErrorMsg ( )
```

**Return value:**
The error string.

**GRBEnv.getParamInfo()**

Obtain information about a parameter.

```java
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.

```java
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.

```java
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.

```java
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.getEnv` to retrieve the environment associated with a model if you would like a parameter change to affect that 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.

```c
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.

**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.

```c
void writeParams ( String paramfile )
```

**Arguments:**
- **paramfile**: Name of the file to which non-default parameter settings should be written. The previous contents are overwitten.
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).

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.

```java
GRBModel GRBModel ( GRBEnv env )
```

Model constructor.

**Arguments:**
- `env`: Environment for new model.

**Return value:**
- New model object. Model initially contains no variables or constraints.

```java
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, or .ilp. 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.

```java
GRBModel GRBModel ( GRBModel model )
```

Create a copy of an existing model.

**Arguments:**
- `model`: Model to copy.

**Return value:**
- New model object. Model is a clone of the input model.
**GRBModel.addConstr()**

Add a single linear constraint to a model. Multiple signatures are available.

```java
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.

```java
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.

```java
GRBConstr addConstr ( 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`: Name for new constraint.

**Return value:**
New constraint object.

```java
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.

```java
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`: 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.

```java
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.addQConstr()

Add a quadratic constraint to a model. Multiple signatures are available.

Important note: the algorithms that Gurobi uses to solve quadratically constrained problems
can only handle certain types of quadratic constraints. Constraints of the following forms are always accepted:

- \( x^TQx + q^Tx \leq b \), where \( Q \) is Positive Semi-Definite (PSD)
- \( x^Tx \leq y^2 \), where \( x \) is a vector of variables, and \( y \) is a non-negative variable (a Second-Order Cone)
• $x^T x \leq yz$, where $x$ is a vector of variables, and $y$ and $z$ are non-negative variables (a rotated Second-Order Cone)

If you add a constraint that isn’t in one of these forms (and Gurobi presolve is unable to transform the constraint into one of these forms), you’ll get an error when you try to solve the model. Constraints where the quadratic terms only involve binary variables will always be transformed into one of these forms.

```c
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 or GRB.GREATER_EQUAL).
- `rhsExpr`: Right-hand side expression for new quadratic constraint.
- `name`: Name for new constraint.

**Return value:**
New quadratic constraint object.

```c
GRBQConstr addQConstr ( 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 or GRB.GREATER_EQUAL).
- `rhsVar`: Right-hand side variable 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.

```c
GRBConstr addRange ( GRBLinExpr expr,
                      double lower,
                      double upper,
                      String name )
```

202
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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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 the change won’t take effect until the next call to `GRBModel.optimize` or `GRBModel.update` on that model.

```java
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 the changes won’t take effect until the next call to `GRBModel.optimize` or `GRBModel.update` on that model.

```java
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 of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result.

This method populates the IISCONSTR and IISQCONSTR 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.

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

```java
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.

```java
void discardConcurrentEnvs()
```

**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.

```java
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 \( 2p \) to the feasibility relaxation objective for **relaxobjtype=0**, it would contribute \( 2*2p \) 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.

```java
double feasRelax ( int relaxobjtype,
                  boolean minrelax,
                  GRBVar[] vars,
                  double[] lbpen,
                  double[] ubpen,
                  GRBConstr[] constr,
                  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**.
- **constr**: Linear constraints that are allowed to be violated.
- **rhspen**: Penalty for violating a linear constraint. One entry for each variable in argument **constr**.

**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.

```java
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.

```java
GRBModel fixedModel ( )
Return value:
Fixed model associated with calling object.
```

**GRBModel.get()**
Query the value(s) of an attribute. Use this method for scalar model attributes and for arrays of constraint or variable attributes.

```java
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```java
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.

```java
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.

```java
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.

```java
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).

```java
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.

```java
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.

```java
GRBEnv getConcurrentEnv ( int num )
```

**Arguments:**
- `num`: The concurrent environment number.

**Return value:**
The concurrent environment for the model.
GRBModel.getConstrByName()

Retrieve a constraint from its name. If multiple constraints have the same name, this method chooses one arbitrarily.

---

GRBConstr getConstrByName ( String name )

**Arguments:**
- name: The name of the desired constraint.

**Return value:**
The requested constraint.

---

GRBModel.getConstrs()

Retrieve an array of all constraints in the model.

---

GRBConstr[] getConstrs ()

**Return value:**
All constraints in the model.

---

GRBModel.getEnv()

Query the environment associated with the model. Note that each model makes its own copy of the environment when it is created. To change parameters for a model, for example, you should use this method to obtain the appropriate environment object.

---

GRBEnv getEnv ()

**Return value:**
The environment for the model.

---

GRBModel.getObjective()

Retrieve the model objective.

Note that the constant and linear portions of the objective can also be retrieved using the ObjCon and Obj attributes.

---

GRBExpr getObjective ()

**Return value:**
The model 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 the description of setPWLObj for additional information on what the values in $x$ and $y$ mean.

```java
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.

```java
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.

```java
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.

\[ \text{GRBLinExpr getRow ( GRBConstr constr )} \]

Arguments:
- \text{constr}: The constraint of interest.

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.

\[ \text{int getSOS ( GRBSOS sos, GRBVar[\_] vars, double[\_] weights, int[\_] type )} \]

Arguments:
- \text{sos}: The SOS set of interest.
- \text{vars}: A list of variables that participate in \text{sos}. Can be null.
- \text{weights}: The SOS weights for each participating variable. Can be null.
- \text{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.

\[ \text{GRBSOS[\_] getSOSs ( )} \]

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.

```java
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.

```java
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.

```java
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.

```java
void optimize( )
```
GRBModel.presolve()
Perform presolve on a model.

```java
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 this isn’t 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.

```java
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` (for a MIP start), `.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.

```java
void  remove ( GRBConstr  constr )
```

Remove a linear constraint from the model. Note that the constraint isn’t actually removed from the model until the next call to `GRBModel.optimize` or `GRBModel.update` on that model.

*Arguments:*

`constr`: The linear constraint to remove.

```java
void  remove ( GRBQConstr  qconstr )
```

Remove a quadratic constraint from the model. Note that the constraint isn’t actually removed from the model until the next call to `GRBModel.optimize` or `GRBModel.update` on that model.

*Arguments:*

`qconstr`: The quadratic constraint to remove.
void remove ( GRBSOS sos )

Remove an SOS constraint from the model. Note that the SOS isn’t actually removed from the model until the next call to GRBModel.optimize or GRBModel.update on that model.

Arguments:
  sos: The SOS constraint to remove.

void remove ( GRBVar var )

Remove a variable from the model. Note that the variable isn’t actually removed from the model until the next call to GRBModel.optimize or GRBModel.update on that model.

Arguments:
  var: The variable to remove.

GRBModel.reset()

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

void reset ( )

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 an attribute. Use this method for scalar model attributes, or for arrays of constraint or variable attributes.

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.

```c
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.

```c
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.

```c
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.

```c
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.
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.

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.

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.

Set a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:
attr: The attribute quadratic 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.
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.

```c
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.

```c
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.

```c
void set ( GRB.CharAttr attr,
          GRBQConstr[] [] [] qconstrs,
          char[] [] [] newvalues )
```

Set the value of a double-valued model attribute.

**Arguments:**
- `attr`: The attribute being modified.
- `newvalue`: The desired new value for the attribute.

```c
void set ( GRB.DoubleAttr attr,
          double newvalue )
```

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.

```java
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.

```java
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.

```java
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.

```java
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.

```c
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.

```c
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.

```c
void set ( GRB.DoubleAttr attr,
          GRBQConstr[][][] qconstrs,
          double[][][] newvalues )
```

Set the value of an int-valued model attribute.

**Arguments:**
- `attr`: The attribute being modified.
- `newvalue`: The desired new value for the attribute.

```c
void set ( GRB.IntAttr attr,
          int newvalue )
```

Set an int-valued variable attribute for an array of variables.

**Arguments:**
- `attr`: The attribute being modified.
- `vars`: An array of variables.
- `newvalues`: An array of new values for the attributes.

```c
void set ( GRB.IntAttr attr,
          GRBVar[] vars,
          int[] newvalues )
```
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.
ewvalues: 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.
ewvalues: 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.
ewvalues: 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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
void set ( GRB.StringAttr attr,
          GRBQConstr[][][][] qconstrs,
          String[][][][] newvalues )
```

**GRBModel.setObjective()**

Set the model objective equal to a linear or quadratic expression.

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.

```java
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).

```c
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.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 example, suppose we want to define the function `f(x)` shown below:

![Graph](https://via.placeholder.com/150)

The vertices of the function occur at the points (1,1), (3,2) and (5,4), so `x` is {1, 3, 5} and `y` is {1, 2, 4}. With these arguments 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 set of `n` points

\[
\mathbf{x} = \{x_1, \ldots, x_n\}, \quad \mathbf{y} = \{y_1, \ldots, y_n\}
\]

define the following piecewise-linear function:

\[
f(v) = \begin{cases} 
    y_1 + \frac{y_2-y_1}{x_2-x_1}(v-x_1), & \text{if } v \leq x_1, \\
    y_i + \frac{y_{i+1}-y_i}{x_{i+1}-x_i}(v-x_i), & \text{if } v \geq x_i \text{ and } v \leq x_{i+1}, \\
    y_n + \frac{y_n-y_{n-1}}{x_n-x_{n-1}}(v-x_n), & \text{if } v \geq x_n.
\end{cases}
\]
The $x$ entries must appear in non-decreasing order. Two points can have the same $x$ coordinate — this can be useful for specifying a discrete jump in the objective function.

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.

Setting a piecewise-linear objective for a variable will set the $\text{Obj}$ attribute on that variable to 0. Similarly, setting the $\text{Obj}$ attribute will delete the piecewise-linear objective on that variable.

Each variable can have its own piecewise-linear objective function. They must be specified individually, even if multiple variables share the same function.

```csharp
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.terminate()`

Generate a request to terminate the current optimization. This method can be called at any time during an optimization.

```csharp
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.

```csharp
void tune ( )
```

`GRBModel.update()`

Process any pending model modifications.

```csharp
void update ( )
```
GRBModel.write()

This method is the general entry point for writing model data to a file. It can be used to write optimization models, IIS submodels, solutions, basis vectors, MIP start vectors, or parameter settings. The type of file is determined by the file suffix. File formats are described in the File Format section.

| void write ( String filename ) |

Arguments:

filename: Name of the file to write. The file type is encoded in the file name suffix. Valid suffixes for writing the model itself are .mps, .rew, .lp, or .rlp. An IIS can be written by using an .ilp suffix. Use .sol for a solution file, .mst for a MIP start, .bas for a basis file, or .prm for a parameter file. The suffix may optionally be followed by .gz, .bz2, or .7z, which produces a compressed result.
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, 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.sameAs()

```java
boolean 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.

```java
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.

```java
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.

```java
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.

```java
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.
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, 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.

<table>
<thead>
<tr>
<th>char get ( GRB.CharAttr attr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query the value of a char-valued attribute.</td>
</tr>
<tr>
<td><strong>Arguments:</strong></td>
</tr>
<tr>
<td>attr: The attribute being queried.</td>
</tr>
<tr>
<td><strong>Return value:</strong></td>
</tr>
<tr>
<td>The current value of the requested attribute.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>double get ( GRB.DoubleAttr attr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query the value of a double-valued attribute.</td>
</tr>
<tr>
<td><strong>Arguments:</strong></td>
</tr>
<tr>
<td>attr: The attribute being queried.</td>
</tr>
<tr>
<td><strong>Return value:</strong></td>
</tr>
<tr>
<td>The current value of the requested attribute.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int get ( GRB.IntAttr attr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query the value of an int-valued attribute.</td>
</tr>
<tr>
<td><strong>Arguments:</strong></td>
</tr>
<tr>
<td>attr: The attribute being queried.</td>
</tr>
<tr>
<td><strong>Return value:</strong></td>
</tr>
<tr>
<td>The current value of the requested attribute.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String get ( GRB.StringAttr attr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query the value of a string-valued attribute.</td>
</tr>
<tr>
<td><strong>Arguments:</strong></td>
</tr>
<tr>
<td>attr: The attribute being queried.</td>
</tr>
<tr>
<td><strong>Return value:</strong></td>
</tr>
<tr>
<td>The current value of the requested attribute.</td>
</tr>
</tbody>
</table>
GRBConstr.sameAs()

boolean 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.
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 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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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.
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.
4.7 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.

```java
double getValue ()

Return value:
Value of the expression for the current solution.
```
4.8 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.

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 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.

```java
void addConstant ( double c )
```

**Arguments:**
- `c`: Constant to add to expression.

GRBLinExpr.addTerm()

Add a single term into a linear expression.

```java
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.

```java
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.

```java
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)
    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)
    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.

```c
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.

```c
void remove ( int i )
```

Remove the term stored at index `i` of the expression.

**Arguments:**
- `i`: The index of the term to be removed.

```c
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).

```c
int size ( )
```

**Return value:**
- Number of terms in the expression.
4.9 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.

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.

**Return value:**
An empty expression object.

---

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 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.

```java
void add ( GRBLinExpr le )
```

Add a linear expression.

**Arguments:**
- `le`: Linear expression to add.

```java
void add ( GRBQuadExpr qe )
```

Add a quadratic expression.

**Arguments:**
- `qe`: Quadratic expression to add.

**GRBQuadExpr.addConstant()**

Add a constant into a quadratic expression.

```java
void addConstant ( double c )
```

**Arguments:**
- `c`: Constant to add to expression.

**GRBQuadExpr.addTerm()**

Add a single term into a quadratic expression.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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.

```c
void clear()
```

**GRBQuadExpr.getCoeff()**

Retrieve the coefficient from a single quadratic term of the quadratic expression.

```c
double getCoeff(int i)
```

**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.

```c
GRBLinExpr getLinExpr()
```

**Return value:**

Linear expression associated with the quadratic expression.

**GRBQuadExpr.getValue()**

Compute the value of a quadratic expression for the current solution.

```c
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 )

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.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.

```java
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).

```java
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.

```java
int size ( )
```
Return value:
Number of quadratic terms in the expression.
4.10 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.

<table>
<thead>
<tr>
<th>GRBColumn GRBColumn ( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an empty column.</td>
</tr>
<tr>
<td><strong>Return value:</strong></td>
</tr>
<tr>
<td>An empty column object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GRBColumn GRBColumn ( GRBColumn orig )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy an existing column.</td>
</tr>
<tr>
<td><strong>Return value:</strong></td>
</tr>
<tr>
<td>A copy of the input column object.</td>
</tr>
</tbody>
</table>

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.

```cpp
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.

```cpp
void clear ()
```

GRBColumn.getCoeff()
Retrieve the coefficient from a single term in the column.

```cpp
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.

```cpp
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()

Retrieve the number of terms in the column.

int size ( )

Return value:
Number of terms in the column.
4.11 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`, 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`). More sophisticated MIP callbacks might use `GRBCallback.getSolution` to retrieve values from the solution to the current node relaxation, and then use `GRBCallback.addCut` or `GRBCallback.addLazy` to add a constraint to cut off that relaxation solution, or `GRBCallback.setSolution` to import a heuristic solution built from the relaxation solution.

When solving a model using multiple threads, note that 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.

You can look at the `Callback.java` example for details of how to use Gurobi callbacks.

**GRBCallback()**

Callback constructor.

```java
GRBCallback GRBCallback ( )
```

Return value:

A callback object.

**GRBCallback.abort()**

Abort optimization. When the optimization stops, the `Status` attribute will be equal to `GRB.INTERRUPTED`.

```java
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`.

---

264
When adding your own cuts, you must set parameter PreCrush to value 1. This setting shuts off a few presolve reductions that sometimes prevent cuts on the original model from being applied to the presolved model.

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.

```c
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.

**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.

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.

```c
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.
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 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.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.

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.setSolution()**

Import solution values for a heuristic solution. Only available when the `where` member variable is equal to `GRB.CB_MIPNODE`.

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. At the end of 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.

```java
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.
```

```java
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.
```
4.12 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 1.5, visit this site.

GRBException()

Exception constructor.

<table>
<thead>
<tr>
<th>GRBException</th>
<th>GRBException ( int errcode )</th>
</tr>
</thead>
</table>

Create a Gurobi exception.

**Arguments:**
- `errcode`: Error code for exception.

**Return value:**
- An exception object.

<table>
<thead>
<tr>
<th>GRBException</th>
<th>GRBException ( String errmsg )</th>
</tr>
</thead>
</table>

Create a Gurobi exception.

**Arguments:**
- `errmsg`: Error message for exception.

**Return value:**
- An exception object.

<table>
<thead>
<tr>
<th>GRBException</th>
<th>GRBException ( String errmsg, int errcode )</th>
</tr>
</thead>
</table>

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.

<table>
<thead>
<tr>
<th>int</th>
<th>getErrorCode ( )</th>
</tr>
</thead>
</table>

**Return value:**
- The error code associated with the exception.
4.13 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).

```java
// Model status codes (after call to optimize())
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;
}

// 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;

// 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;

// Numeric constants

public static final double INFINITY = 1e100;
public static final double UNDEFINED = 1e101;

// Callback constants

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 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 MIPSOL_SOL   = 4001;
public static final int MIPSOL_OBJ   = 4002;
public static final int MIPSOL_OBJBST= 4003;
public static final int MIPSOL_OJBND = 4004;
public static final int MIPSOL_NODCNT= 4005;
public static final int MIPSOL_SOLCNT= 4006;
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_OJBND = 5004;
public static final int MIPNODE_NODCNT= 5005;
public static final int MIPNODE_SOLCNT= 5006;
public static final int MSG_STRING   = 6001;
public static final int RUNTIME      = 6002;
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;

// Errors

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 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 NOT_IN_MODEL = 20001;
public static final int FAILED_TO_CREATE_MODEL = 20002;
public static final int INTERNAL = 20003;
}

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;

public static final int METHOD_AUTO = -1;
public static final int METHOD_PRIMAL = 0;
public static final int METHOD_DUAL = 1;
public static final int METHOD_BARRIER = 2;
public static final int METHOD_CONCURRENT = 3;
public static final int METHOD_DETERMINISTIC_CONCURRENT = 4;

public static final int BARORDER_AUTOMATIC = 0;
public static final int BARORDER_AMD = 1;
public static final int BARORDER_NESTEDDISSECTION = 2;

public static final int FEASRELAX_LINEAR = 0;
public static final int FEASRELAX_QUADRATIC = 1;
public static final int FEASRELAX_CARDINALITY = 2;

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 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 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 GRBEnv.get or GRBEnv.set). Please refer to the Parameters section to see a list of all string parameters and their functions.
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 Quick Start Guide 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.

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, or GRBSOS). 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.

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 or GRBModel.AddQConstr 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. 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’ll sometimes also discuss a special case of QCP, the Second-Order Cone Program (SOCP). If the model contains any integer variables, semi-continuous variables, semi-integer variables, or Special Ordered Set (SOS) constraints, the model is a Mixed Integer Program (MIP). We’ll also sometimes discuss special cases of MIP, including Mixed Integer Linear Programs (MILP), Mixed Integer Quadratic Programs (MIQP), Mixed Integer Quadratically-Constrained Programs (MIQCP), and Mixed Integer Second-Order Cone Programs (MISOCP). The Gurobi Optimizer handles all of these model classes.
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 and QCP models, and the branch-and-cut algorithm to solve mixed integer models. 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, and GRBQConstr 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 input model, except that all integer variables are fixed to their values in the MIP solution. In some applications, it is useful to compute information on this continuous version of the MIP model (e.g., dual variables, sensitivity information, etc.).

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, or GRBModel.Get, and modified using GRBVar.Set, GRBConstr.Set, GRBQConstr.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 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 very important item to note about attribute and model modifications in the Gurobi optimizer is that they are performed in a lazy fashion, meaning that they don’t actually affect the model until the next call to Optimize or Update on that model object. This approach provides the advantage that the model remains unchanged while you are in the process of making multiple modifications. The downside, of course, is that you have to remember to call Update in order to see the effect of your changes.

If you forget to call update, your program won’t crash. The most common symptom of a missing update is a NOT_IN_MODEL exception, which indicates that the object you are trying to reference isn’t in the model yet.

Managing Parameters

The Gurobi optimizer provides a set of parameters to 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 GRBEnv.Set. Current values may also be retrieved with GRBEnv.Get. Parameters can be of type int, double, or string. You can also 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.
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 `GRBModel.GetEnv` to retrieve the environment associated with a particular model if you want to change a parameter for that model.

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 Windows `PATH`. 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 32-bit Gurobi native library when you’ve selected the `x86` Platform Target, and similarly 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 Windows machine, and the 32-bit DLL on a 32-bit Windows 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.

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.
5.1 **GRBEnv**

Gurobi environment object. Gurobi models are always associated with an environment. You must create an environment before you can 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 .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.

---

**GRBEnv()**

Environment constructor.

Constructor for `GRBEnv` object. If the constructor is called with no arguments, no log file will be written for the environment.

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.

Note that the `GRBEnv` constructor 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).

In general, you should aim to create a single Gurobi environment object 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.

---

<table>
<thead>
<tr>
<th><strong>GRBEnv</strong> (<em>GRBEnv</em> () )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a Gurobi environment (with logging disabled).</td>
</tr>
<tr>
<td><strong>Return value:</strong></td>
</tr>
<tr>
<td>An environment object (with no associated log file).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>GRBEnv</strong> (<em>GRBEnv</em> ( string logFileName ) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a Gurobi environment (with logging enabled).</td>
</tr>
<tr>
<td><strong>Arguments:</strong></td>
</tr>
<tr>
<td>logFileName: The desired log file name.</td>
</tr>
<tr>
<td><strong>Return value:</strong></td>
</tr>
<tr>
<td>An environment object.</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th><strong>GRBEnv</strong> (<em>GRBEnv</em> ( string logFileName,  string computeserver,  int port,  string password,  int priority,  double timeout ) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Create a client Gurobi environment on a compute server.

**Arguments:**

- **logFileName**: The name of the log file for this environment. Pass an empty string for no log file.
- **computeServer**: A comma-separated list of Gurobi compute servers. You can refer to compute server machines using their names or their IP addresses.
- **port**: The port number used to connect to the compute server. You should pass a -1 value, which indicates that the default port should be used, unless your server administrator has changed our recommended port settings.
- **password**: The password for gaining access to the specified compute servers. Pass an empty string if no password is required.
- **priority**: 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.
- **timeout**: Job timeout (in seconds). If the job doesn’t reach the front of the queue before the specified timeout, the constructor will throw a JOB_REJECTED exception. Use a negative value to indicate that the call should never timeout.

**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.

```csharp
void Dispose()
```

**GRBEnv.ErrorMsg**

(Property) The error message for the most recent exception associated with this environment.

**GRBEnv.Get()**

Query the value of a parameter.

```csharp
double Get ( GRB.DoubleParam param )
```

Query the value of a double-valued parameter.

**Arguments:**
**get**

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.

```c
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.

```c
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.

```c
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.

```c
void GetParamInfo ( GRB.IntParam param,
                   int[] info )
```

Obtain detailed information about an integer parameter.

**Arguments:**
**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 four entries: the current value of the parameter, the minimum allowed value, the maximum allowed value, and the default value.

---

**GRBEEnv.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.

---

**GRBEEnv.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.

---

**GRBEEnv.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.

```c
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.GetEnv` to retrieve the environment associated with a model if you would like a parameter change to affect that model.

```c
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.

```c
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.

```c
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.
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.

```c
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 .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.

**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.

```csharp
GRBModel GRBModel ( GRBEnv env )
```

Model constructor.

**Arguments:**
- `env`: Environment for new model.

**Return value:**
- New model object. Model initially contains no variables or constraints.

```csharp
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, or .ilp. 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.

```csharp
GRBModel GRBModel ( GRBModel model )
```

Create a copy of an existing model.

**Arguments:**
- `model`: Model to copy.

**Return value:**
- New model object. Model is a clone of the input model.

**GRBModel.AddConstr()**

Add a single linear constraint to a model. Multiple signatures are available.
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.

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.

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.

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.

Add new linear constraints to a model. The number of added constraints is determined by the length of the input constraint objects. The length of the constraint objects is equal to the number of added constraints.

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.AddQConstr()

Add a quadratic constraint to a model. Multiple signatures are available.

Important note: the algorithms that Gurobi uses to solve quadratically constrained problems can only handle certain types of quadratic constraints. Constraints of the following forms are always accepted:

- $x^T Q x + q^T x \leq b$, 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)
- $x^T x \leq y z$, where $x$ is a vector of variables, and $y$ and $z$ are non-negative variables (a rotated Second-Order Cone)

If you add a constraint that isn’t in one of these forms (and Gurobi presolve is unable to transform the constraint into one of these forms), you’ll get an error when you try to solve the model.
Constraints where the quadratic terms only involve binary variables will always be transformed into one of these forms.

```c
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` or `GRB.GREATER_EQUAL`).
- `rhsExpr`: Right-hand side expression for new quadratic constraint.
- `name`: Name for new constraint.

**Return value:**
New quadratic constraint object.

```c
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`.

```c
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.

```c
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.

```c
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.

```c
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.
**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.

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.
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 \texttt{start} and \texttt{len} arguments allow you to specify which variables to add.

**Arguments:**

- \texttt{lb}: Lower bounds for new variables. Can be \texttt{null}, in which case the variables get lower bounds of 0.0.
- \texttt{ub}: Upper bounds for new variables. Can be \texttt{null}, in which case the variables get infinite upper bounds.
- \texttt{obj}: Objective coefficients for new variables. Can be \texttt{null}, in which case the variables get objective coefficients of 0.0.
- \texttt{type}: Variable types for new variables (\texttt{GRB.CONTINUOUS}, \texttt{GRB.BINARY}, \texttt{GRB.INTEGER}, \texttt{GRB.SEMICONT}, or \texttt{GRB.SEMIINT}). Can be \texttt{null}, in which case the variables are assumed to be continuous.
- \texttt{names}: Names for new variables. Can be \texttt{null}, in which case all variables are given default names.
- \texttt{start}: The first variable in the list to add.
- \texttt{len}: The number of variables to add.

**Return value:**

Array of new variable objects.

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 \texttt{GRBColumn} objects.

**Arguments:**

- \texttt{lb}: Lower bounds for new variables. Can be \texttt{null}, in which case the variables get lower bounds of 0.0.
- \texttt{ub}: Upper bounds for new variables. Can be \texttt{null}, in which case the variables get infinite upper bounds.
- \texttt{obj}: Objective coefficients for new variables. Can be \texttt{null}, in which case the variables get objective coefficients of 0.0.
- \texttt{type}: Variable types for new variables (\texttt{GRB.CONTINUOUS}, \texttt{GRB.BINARY}, \texttt{GRB.INTEGER}, \texttt{GRB.SEMICONT}, or \texttt{GRB.SEMIINT}). Can be \texttt{null}, in which case the variables are assumed to be continuous.
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 the change won't take effect until the next call to GRBModel.Optimize or GRBModel.Update on that model.

```c
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 the changes won't take effect until the next call to GRBModel.Optimize or GRBModel.Update on that model.

```c
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 of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result.

This method populates the IISCONSTR and IISQCONSTR constraint attributes, the IISSOS SOS attribute, and the IISSLB, and IISSUB 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.

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.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 rhs\(_{\text{pen}}\) value \(p\) is violated by 2.0, it would contribute \(2*p\) to the feasibility relaxation objective for \(\text{relaxobjtype}=0\), it would contribute \(2*2*p\) for \(\text{relaxobjtype}=1\), it would contribute \(p\) for \(\text{relaxobjtype}=2\).

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[] constr, 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.
constr: Linear constraints that are allowed to be violated.
rhspen: Penalty for violating a linear constraint. One entry for each variable in argument constr.

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.

Return value:
Fixed model associated with calling object.

GRBModel.Get()

Query the value(s) of an attribute. Use this method for scalar model attributes and for arrays of constraint or variable attributes.

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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.
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 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[]  conhrs )

Query an int-valued constraint attribute for an array of constraints.

Arguments:
attr: The attribute being queried.
conhrs: The constraints whose attribute values are being queried.

Return value:
The current values of the requested attribute for each input constraint.
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.

```c
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.

```c
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.

```c
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.

```c
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.
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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 constraint from its name. If multiple constraints have the same name, this method chooses one arbitrarily.

```
    GRBConstr GetConstrByName ( string name )
```

**Arguments:**
- `name`: The name of the desired constraint.

**Return value:**
The requested constraint.

**GRBModel.GetConstrs()**

Retrieve an array of all constraints in the model.
GRBConstr[] GetConstrs ( )

Return value:
All constraints in the model.

GRBModel.GetEnv()

Query the environment associated with the model. Note that each model makes its own copy of the environment when it is created. To change parameters for a model, for example, you should use this method to obtain the appropriate environment object.

GRBEnv GetEnv ( )

Return value:
The environment for the model.

GRBModel.GetObjective()

Retrieve the model objective.

Note that the constant and linear portions of the objective can also be retrieved using the ObjCon and Obj attributes.

GRBExpr GetObjective ( )

Return value:
The model 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 the description of SetPWLObj 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.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.

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.

```csharp
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.

```csharp
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.

```csharp
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.

```csharp
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.

```csharp
void Optimize()
```
GRBModel.Presolve()

Perform presolve on a model.

```cpp
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 this isn’t 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.

```cpp
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 (for a MIP start), .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 constraint from a model.

```cpp
void Remove ( GRBConstr constr )
```

Remove a constraint from the model. Note that the constraint isn’t actually removed from the model until the next call to GRBModel.Optimize or GRBModel.Update on that model.

```cpp
void Remove ( GRBQConstr qconstr )
```

Remove a quadratic constraint from the model. Note that the quadratic constraint isn’t actually removed from the model until the next call to GRBModel.Optimize or GRBModel.Update on that model.

```cpp
void Remove ( GRBSOS sos )
```

Remove an SOS constraint from the model. Note that the SOS isn’t actually removed from the model until the next call to GRBModel.Optimize or GRBModel.Update on that model.

```cpp
void Remove ( GRBVar var )
```
Remove a variable from the model. Note that the variable isn’t actually removed from the
model until the next call to GRBModel.Optimize or GRBModel.Update on that model.

Arguments:

  var: The variable to remove.

GRBModel.Reset()

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

  void Reset ( )

GRBModel.SetCallback()

Set the callback object for a model. The Callback() method on this object will be called period-
ically 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 an attribute. Use this method for scalar model attributes, or for arrays of
constraint or variable attributes.

  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.
\textbf{start}: The index of the first variable of interest in the list.
\textbf{len}: The number of variables.

\begin{verbatim}
void Set ( GRB.CharAttr attr,
          GRBVar[,] vars,
          char[,] newvalues )
Set a char-valued variable attribute for a two-dimensional array of variables.
\textbf{Arguments}:
\begin{itemize}
  \item \textbf{attr}: The attribute being modified.
  \item \textbf{vars}: A two-dimensional array of variables whose attribute values are being modified.
  \item \textbf{newvalues}: The desired new values for the attribute for each input variable.
\end{itemize}
\end{verbatim}

\begin{verbatim}
void Set ( GRB.CharAttr attr,
          GRBVar[,][,] vars,
          char[,][,] newvalues )
Set a char-valued variable attribute for a three-dimensional array of variables.
\textbf{Arguments}:
\begin{itemize}
  \item \textbf{attr}: The attribute being modified.
  \item \textbf{vars}: A three-dimensional array of variables whose attribute values are being modified.
  \item \textbf{newvalues}: The desired new values for the attribute for each input variable.
\end{itemize}
\end{verbatim}

\begin{verbatim}
void Set ( GRB.CharAttr attr,
          GRBConstr[] constrs,
          char[] newvalues )
Set a char-valued constraint attribute for an array of constraints.
\textbf{Arguments}:
\begin{itemize}
  \item \textbf{attr}: The attribute being modified.
  \item \textbf{constrs}: The constraints whose attribute values are being modified.
  \item \textbf{newvalues}: The desired new values for the attribute for each input constraint.
\end{itemize}
\end{verbatim}

\begin{verbatim}
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.
\textbf{Arguments}:
\begin{itemize}
  \item \textbf{attr}: The attribute being modified.
  \item \textbf{constrs}: A one-dimensional array of constraints whose attribute values are being modified.
  \item \textbf{newvalues}: The desired new values for the attribute for each input constraint.
  \item \textbf{start}: The index of the first constraint of interest in the list.
  \item \textbf{len}: The number of constraints.
\end{itemize}
\end{verbatim}
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.
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.

```c
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.

```c
void Set ( GRB.CharAttr attr,
          GRBQConstr[,,,] qconstrs,
          char[,,,] newvalues )
```

Set the value of a double-valued model attribute.

**Arguments:**
- `attr`: The attribute being modified.
- `newvalue`: The desired new value for the attribute.

```c
void Set ( GRB.DoubleAttr attr,
          double newvalue )
```

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.

```c
void Set ( GRB.DoubleAttr attr,
          GRBVar[] vars,
          double[] newvalues )
```

Set a double-valued variable attribute for a sub-array of variables.

**Arguments:**
- `attr`: The attribute being modified.

```c
void Set ( GRB.DoubleAttr attr,
          GRBVar[] vars,
          double[] newvalues,
          int start,
          int len )
```
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.
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.

```c
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.

```c
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.

```c
void Set ( GRB.StringAttr attr,
          GRBVar[:,:,:] vars,
          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.

```c
void Set ( GRB.StringAttr attr,
          GRBConstr[] constrs,
          string[] newvalues )
```

Set a string-valued constraint attribute for a sub-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.
- **start**: The index of the first constraint of interest in the list.
- **len**: The number of constraints.

```c
void Set ( GRB.StringAttr attr,
          GRBConstr[], constrs,
          string[] newvalues,
          int start,
          int len )
```
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

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.

```c
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).

```c
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.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 example, suppose we want to define the function $f(x)$ shown below:

The vertices of the function occur at the points $(1, 1)$, $(3, 2)$ and $(5, 4)$, so $x$ is {1, 3, 5} and $y$ is {1, 2, 4}. With these arguments 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 set of $n$ points

$$x = \{x_1, \ldots, x_n\}, \quad y = \{y_1, \ldots, y_n\}$$

define the following piecewise-linear function:

$$f(v) = \begin{cases} 
    y_1 + \frac{y_2-y_1}{x_2-x_1}(v-x_1), & \text{if } v \leq x_1, \\
    y_i + \frac{y_{i+1}-y_i}{x_{i+1}-x_i}(v-x_i), & \text{if } v \geq x_i \text{ and } v \leq x_{i+1}, \\
    y_n + \frac{y_{n-1}-y_n}{x_n-x_{n-1}}(v-x_n), & \text{if } v \geq x_n.
\end{cases}$$

The $x$ entries must appear in non-decreasing order. Two points can have the same $x$ coordinate — this can be useful for specifying a discrete jump in the objective function.

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.

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.
Each variable can have its own piecewise-linear objective function. They must be specified individually, even if multiple variables share the same function.

```c
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.Terminate()**

Generate a request to terminate the current optimization. This method can be called at any time during an optimization.

```c
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.

```c
void Tune ( )
```

---

**GRBModel.Update()**

Process any pending model modifications.

```c
void Update ( )
```

---

**GRBModel.Write()**

This method is the general entry point for writing model data to a file. It can be used to write optimization models, IIS submodels, solutions, basis vectors, MIP start vectors, or parameter settings. The type of file is determined by the file suffix. File formats are described in the File Format section.

```c
void Write ( string filename )
```

**Arguments:**

- **filename**: Name of the file to write. The file type is encoded in the file name suffix. Valid suffixes for writing the model itself are .mps, .rew, .lp, or .rlp. An IIS can be written by using an .ilp suffix. Use .sol for a solution file, .mst for a MIP start, .bas for a basis file, or .prm for a parameter file. The suffix may optionally be followed by .gz, .bz2, or .7z, which produces a compressed result.
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.

```c
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.

```c
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.

```c
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.

```c
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.SameAs()

```c
bool SameAs ( GRBVar var2 )
```

328
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.

```c
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.

```c
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.

```c
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.

```c
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.
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.SameAs()

```cpp
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.

```java
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.
```

```java
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.
```

```java
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.
```

```java
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.
```
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 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.

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.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.
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 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.

```c
int Get ( GRB.IntAttr attr )
```

Arguments:
- attr: The attribute being queried.

Return value:
- The current value of the requested attribute.
5.7 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.
5.8 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., \( \text{expr} = 0 \)), variables (e.g., \( \text{expr} = 1 \times x + 2 \times y \)), or from other expressions (e.g., \( \text{expr2} = 2 \times \text{expr1} + x \), or \( \text{expr3} = \text{expr1} + 2 \times \text{expr2} \)). You can also modify existing expressions (e.g., \( \text{expr} += x \), or \( \text{expr2} -= \text{expr1} \)).

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.

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 \( \text{expr} = \text{expr} + x \times x \) or \( \text{expr} += x \times x \) 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.

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.

\[
\begin{align*}
\text{GRBLinExpr} & \quad \text{GRBLinExpr ( )} \\
& \quad \text{Create an empty linear expression.} \\
& \quad \text{Return value:} \\
& \quad \text{An empty expression object.} \\
\text{GRBLinExpr} & \quad \text{GRBLinExpr ( GRBLinExpr \ orig )} \\
& \quad \text{Copy an existing linear expression.} \\
& \quad \text{Arguments:} \\
& \quad \quad \text{orig: Existing expression to copy.} \\
& \quad \text{Return value:} \\
& \quad \quad \text{A copy of the input expression object.}
\end{align*}
\]
**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.

```c
void Add ( GRBLinExpr le )
```

**Arguments:**
- `le`: Linear expression to add.

**GRBLinExpr.AddConstant()**

Add a constant into a linear expression.

```c
void AddConstant ( double c )
```

**Arguments:**
- `c`: Constant to add to expression.

**GRBLinExpr.AddTerm()**

Add a single term into a linear expression.

```c
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.

```c
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.

```c
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.
```cpp
void Clear();
```

GRBLinExpr.Constant
(Property) The constant term from the linear expression.

GRBLinExpr.GetCoeff()
Retrieve the coefficient from a single term of the expression.
```cpp
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.
```cpp
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.
```cpp
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).

GRBLinExpr.Value

(Property) The value of an expression for the current solution.
5.9 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.

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() | GRBLinExpr | orig |
Initialize a quadratic expression from an existing linear expression.

**Arguments:**
- `orig`: Existing linear expression to copy.

**Return value:**
A 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 \((\text{coeff} \times \text{var})\) 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.

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.

```java
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.
```

```java
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.

```java
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.
```

```java
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.
**GRBQuadExpr.Value**

(Property) The value of an expression for the current solution.
5.10 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.
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 property.

GRBColumn()

Column constructor. Create an empty column, or copy an existing column.

| GRBColumn  GRBColumn ( ) |
| Returns:   |
| Create an empty column. |

| GRBColumn  GRBColumn ( GRBColumn orig ) |
| Returns:   |
| Copy an existing column. |

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.
5.12 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.

\textbf{operator} <=

Create an inequality constraint.

\begin{verbatim}
GRBTempConstr operator <= ( GRBQuadExpr lhsExpr, 
                           GRBQuadExpr rhsExpr )
\end{verbatim}

\textbf{Arguments:}
\begin{itemize}
  \item \texttt{lhsExpr}: Left-hand side of inequality constraint.
  \item \texttt{rhsExpr}: Right-hand side of inequality constraint.
\end{itemize}

\textbf{Return value:}
A constraint of type \texttt{GRBTempConstr}. The result is typically immediately passed to method \texttt{GRBModel.AddConstr}.

\textbf{operator} >=

Create an inequality constraint.

\begin{verbatim}
GRBTempConstr operator >= ( GRBQuadExpr lhsExpr, 
                           GRBQuadExpr rhsExpr )
\end{verbatim}

\textbf{Arguments:}
\begin{itemize}
  \item \texttt{lhsExpr}: Left-hand side of inequality constraint.
  \item \texttt{rhsExpr}: Right-hand side of inequality constraint.
\end{itemize}

\textbf{Return value:}
A constraint of type \texttt{GRBTempConstr}. The result is typically immediately passed to method \texttt{GRBModel.AddConstr}.

\textbf{operator} ==

Create an equality constraint.

\begin{verbatim}
GRBTempConstr operator ==( GRBLinExpr lhsExpr, 
                          GRBLinExpr rhsExpr )
\end{verbatim}

\textbf{Arguments:}
\begin{itemize}
  \item \texttt{lhsExpr}: Left-hand side of equality constraint.
  \item \texttt{rhsExpr}: Right-hand side of equality constraint.
\end{itemize}

\textbf{Return value:}
A constraint of type \texttt{GRBTempConstr}. The result is typically immediately passed to method \texttt{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`: Coefficient.
- `var`: Variable.
**Return value:**
A linear expression that is equal to the sum of the constant and the variable arguments.

```c
GRBLinExpr operator + ( GRBVar var, double a )
```

**Arguments:**
- `var`: Variable.
- `a`: Coefficient.

**Return value:**
A linear expression that is equal to the sum of the constant and the variable arguments.

**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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.
<table>
<thead>
<tr>
<th>GRBQuadExpr</th>
<th>GRBQuadExpr ( GRBVar var )</th>
</tr>
</thead>
</table>

**Arguments:**
- **value:** Desired value for quadratic expression.

**Return value:**
A quadratic expression that is equal to specified variable.

<table>
<thead>
<tr>
<th>GRBQuadExpr</th>
<th>GRBQuadExpr ( GRBLinExpr expr )</th>
</tr>
</thead>
</table>

**Arguments:**
- **expr:** Desired value for quadratic expression.

**Return value:**
A quadratic expression that is equal to specified linear expression.
5.13 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`, 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`). More sophisticated MIP callbacks might use `GRBCallback.GetSolution` to retrieve values from the solution to the current node relaxation, and then use `GRBCallback.AddCut` or `GRBCallback.AddLazy` to add a constraint to cut off that relaxation solution, or `GRBCallback.SetSolution` to import a heuristic solution built from the relaxation solution.

When solving a model using multiple threads, note that 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.

You can look at the `callback_cs.cs` example for details of how to use Gurobi callbacks.

**GRBCallback()**

Callback constructor.

<table>
<thead>
<tr>
<th>GRBCallback</th>
<th>GRBCallback ( )</th>
</tr>
</thead>
</table>

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.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`.

When adding your own cuts, you must set parameter `PreCrush` to value 1. This setting shuts off a few presolve reductions that sometimes prevent cuts on the original model from being applied to the presolved model.
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.

```c
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.
```

```c
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.

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.

```c
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.
```

```c
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.

```c
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.

```c
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.OPTIMAL`.

```c
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.

```c
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.
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.

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.SetSolution()

Import solution values for a heuristic solution. Only available when the where member variable is equal to GRB.Callback.MIPNODE.

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. At the end of 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.

```c
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.
```

```c
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.
```
5.14 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.
5.15 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())

public class Status
{
    public const int LOADED = 1;
    public const int OPTIMAL = 2;
    public const int INFEASIBLE = 3;
    public const int INF_OR_UNBD = 4;
    public const int UNBOUNDED = 5;
    public const int CUTOFF = 6;
    public const int ITERATION_LIMIT = 7;
    public const int NODE_LIMIT = 8;
    public const int TIME_LIMIT = 9;
    public const int SOLUTION_LIMIT = 10;
    public const int INTERRUPTED = 11;
    public const int NUMERIC = 12;
    public const int SUBOPTIMAL = 13;
    public const int INPROGRESS = 14;
}

// Basis status info

public const int BASIC = 0;
public const int NONBASIC_LOWER = -1;
public const int NONBASIC_UPPER = -2;
public const int SUPERBASIC = -3;

// Constraint senses

public const char LESS_EQUAL = '<';
public const char GREATER_EQUAL = '>';  
public const char EQUAL = '=';

// Variable types
```
public const char CONTINUOUS = 'C';
public const char BINARY = 'B';
public const char INTEGER = 'I';
public const char SEMICONT = 'S';
public const char SEMIINT = 'N';

// Objective sense
public const int MINIMIZE = 1;
public const int MAXIMIZE = -1;

// SOS types
public const int SOS_TYPE1 = 1;
public const int SOS_TYPE2 = 2;

// Numeric constants
public const double INFINITY = 1e100;
public const double UNDEFINED = 1e101;

// Limits
public const int MAX_STRLEN = 512;

// Callback constants

public class Callback
{
    public const int POLLING = 0;
    public const int PRESOLVE = 1;
    public const int SIMPLEX = 2;
    public const int MIP = 3;
    public const int MIPSOL = 4;
    public const int MIPNODE = 5;
    public const int MESSAGE = 6;
    public const int BARRIER = 7;

    public const int PRE_COLDEL = 1000;
    public const int PRE_ROWDEL = 1001;
    public const int PRE_SENCHG = 1002;
    public const int PRE_BNDCHG = 1003;
    public const int PRE_COECHG = 1004;
    public const int SPX_ITRCNT = 2000;
    public const int SPX_OBJVAL = 2001;
public const int SPX_PRIMINF = 2002;
public const int SPX_DUALINF = 2003;
public const int SPX_ISPERT = 2004;
public const int MIP_OBJBST = 3000;
public const int MIP_OBJBND = 3001;
public const int MIP_NODCNT = 3002;
public const int MIP_SOLCNT = 3003;
public const int MIP_CUTCNT = 3004;
public const int MIP_NODLFT = 3005;
public const int MIP_ITRCNT = 3006;
public const int MIPSOL_SOL = 4001;
public const int MIPSOL_OBJ = 4002;
public const int MIPSOL_OBJBST = 4003;
public const int MIPSOL_OBJBND = 4004;
public const int MIPSOL_NODCNT = 4005;
public const int MIPSOL_SOLCNT = 4006;
public const int MIPNODE_STATUS = 5001;
public const int MIPNODE_REL = 5002;
public const int MIPNODE_OBJBST = 5003;
public const int MIPNODE_OBJBND = 5004;
public const int MIPNODE_NODCNT = 5005;
public const int MIPNODE_SOLCNT = 5006;
public const int BARRIER_ITRCNT = 7001;
public const int BARRIER_PRIMOBJ = 7002;
public const int BARRIER_DUALOBJ = 7003;
public const int BARRIER_PRIMINF = 7004;
public const int BARRIER_DUALINF = 7005;
public const int BARRIER_COMPL = 7006;
public const int MSG_STRING = 6001;
public const int RUNTIME = 6002;
}

// Errors

public class Error {
    public const int OUT_OF_MEMORY = 10001;
    public const int NULL_ARGUMENT = 10002;
    public const int INVALID_ARGUMENT = 10003;
    public const int UNKNOWN_ATTRIBUTE = 10004;
    public const int DATA_NOT_AVAILABLE = 10005;
    public const int INDEX_OUT_OF_RANGE = 10006;
    public const int UNKNOWN_PARAMETER = 10007;
    public const int VALUE_OUT_OF_RANGE = 10008;
    public const int NO_LICENSE = 10009;
public const int SIZE_LIMIT_EXCEEDED = 10010;
public const int CALLBACK = 10011;
public const int FILE_READ = 10012;
public const int FILE_WRITE = 10013;
public const int NUMERIC = 10014;
public const int IIS_NOT_INFEASIBLE = 10015;
public const int NOT_FOR_MIP = 10016;
public const int OPTIMIZATION_IN_PROGRESS = 10017;
public const int DUPLICATE = 10018;
public const int NODEFILE = 10019;
public const int Q_NOT_PSD = 10020;
public const int QCP_EQUALITY_CONSTRAINT = 10021;
public const int NETWORK = 10022;
public const int JOB_REJECTED = 10023;
public const int NOT_SUPPORTED = 10024;
public const int EXCEED_2B_NONZEROS = 10025;
public const int INVALID_PIECEWISE_OBJ = 10026;
public const int NOT_IN_MODEL = 20001;
public const int FAILED_TO_CREATE_MODEL = 20002;
public const int INTERNAL = 20003;

}

public const int METHOD_AUTO = -1;
public const int METHOD_PRIMAL = 0;
public const int METHOD_DUAL = 1;
public const int METHOD_BARRIER = 2;
public const int METHOD_CONCURRENT = 3;
public const int METHOD_DETERMINISTIC_CONCURRENT = 4;

public const int FEASRELAX_LINEAR = 0;
public const int FEASRELAX_QUADRATIC = 1;
public const int FEASRELAX_CARDINALITY = 2;

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 `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 `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 `GRBEnv.Get` or `GRBEnv.Set`). Please refer to the Parameters section to see a list of all string parameters and their functions.
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.

**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. Other useful global functions are `system`, which allows you to issue shell commands from within the Gurobi shell, and `models`, which gives you a list of the currently loaded models. 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`), 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`, `QConstr`, or `SOS`). 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.

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` to add additional variables, and `Model.addConstr` or `Model.addQConstr` to add additional constraints.

Linear constraints are specified by building linear expressions (objects of class `LinExpr`), 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`) instead.

Models are dynamic entities; you can always add or remove variables or constraints.

We often refer to the class of an optimization model. 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’ll sometimes also discuss a special case of QCP, the Second-Order Cone Program (SOCP). If the model contains any integer variables, semi-continuous variables, semi-integer variables, or Special Ordered Set (SOS) constraints, the model is a Mixed Integer Program (MIP). We’ll also sometimes discuss special cases of MIP, including Mixed Integer
Linear Programs (MILP), Mixed Integer Quadratic Programs (MIQP), Mixed Integer Quadratically-Constrained Programs (MIQCP), and Mixed Integer Second-Order Cone Programs (MISOCP). The Gurobi Optimizer handles all of these model classes.

**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 and QCP models, and the branch-and-cut algorithm to solve mixed integer models. 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 input model, except that all integer variables are fixed to their values in the MIP solution. In some applications, it is useful to compute information on this continuous version of the MIP model (e.g., dual variables, sensitivity information, etc.).

**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), linear constraints (Constr.getAttr/Constr.setAttr), quadratic constraints (QConstr.getAttr/QConstr.setAttr), SOSs (SOS.getAttr), 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` or `QuadExpr` 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 `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 very important item to note about attribute and model modifications in the Gurobi optimizer is that they are performed in a lazy fashion, meaning that they don’t actually affect the model until the next call to `optimize` or `update` on that model object. This approach provides the advantage that the model remains unchanged while you are in the process of making multiple modifications. The downside, of course, is that you have to remember to call `update` in order to see the effect of your changes.

If you forget to call update, your program won’t crash. The most common symptom of a missing update is a `NOT_IN_MODEL` exception, which indicates that the object you are trying to reference isn’t in the model yet.

Managing Parameters

The Gurobi optimizer provides a set of parameters to 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.

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.

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.
6.1 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.

models()

models ( )

Print a list of loaded models.
Note that this function will only list models stored in global variables. Models stored in Python data structures (lists, dictionaries, etc.), or inside user classes aren’t listed.

Example usage:
    a = Model("a")
    b = Model("b")
    models()

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 dictionaries.

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:
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:**

```python
expr = quicksum([2*x, 3*y+1, 4*z*z])
expr = quicksum(model.getVars())
```

read()

| read ( filename, env=None )

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`, or `.ilp`. 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**: Compute Server client environment in which to create the model. This argument should only be used when you have explicitly created a Compute Server client environment (using the `Env` constructor).

**Return value:**

Model object containing the model that was read from the input file.

**Example usage:**

```python
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:**
    ```python
    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:**
    ```python
    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:**
    ```python
    setParam("Cuts", 2)
    setParam("Heu*", 0.5)
    setParam("*Interval", 10)
    ```

system()

    system ( command )

    Issue a system command.
    **Arguments:**
    - command: A string containing the desired system command.
    **Example usage:**
    ```python
    ```
```system("ls")
system("rm junk")
```

```writeParams()

<table>
<thead>
<tr>
<th>writeParams</th>
<th>( filename )</th>
</tr>
</thead>
</table>

Write all modified parameters to a file. The file is written in PRM format.

**Example usage:**

```python
setParam("Heu*", 0.5)
writeParams("params.prm")  # file will contain changed parameter
system("cat params.prm")
```
6.2 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), `addConstr` (adds a new constraint), and `update` (integrates model changes into the model).

Model()

```python
Model ( name='', env=None )
```

Model constructor.

**Arguments:**
- `name`: Name of new model.
- `env`: Compute Server client environment in which to create the model. This argument should only be used when you have explicitly created a Compute Server client environment (using the `Env` constructor).

**Return value:**
New model object. Model initially contains no variables or constraints.

**Example usage:**
```python
m = Model("NewModel")
x0 = m.addVar()
```

Model.addConstr()

```python
addConstr ( lhs, sense, rhs, name='' )
```

Add a linear constraint to a model.

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 new linear constraint. Can be a constant, a `Var`, or a `LinExpr`.
- `sense`: Sense for new linear constraint (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).
- `rhs`: Right-hand side for new linear constraint. Can be a constant, a `Var`, or a `LinExpr`.
- `name`: Name for new constraint.

**Return value:**
New constraint object.

**Example usage:**
```python
model.addConstr(x + 2*y, GRB.EQUAL, 3*z, "c0")
model.addConstr(x + y <= 2.0, "c1")
```
Model.addQConstr()

def addQConstr(lhs, sense, rhs, name=''' '):
    Add a quadratic constraint to a model.
    Important note: the algorithms that Gurobi uses to solve quadratically constrained problems
    can only handle certain types of quadratic constraints. Constraints of the following forms are always
    accepted:
    - $x^TQx + q^Tx \leq b$, 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)
    - $x^T x \leq yz$, where $x$ is a vector of variables, and $y$ and $z$ are non-negative variables (a rotated
      Second-Order Cone)
    If you add a constraint that isn’t in one of these forms (and Gurobi presolve is unable to transform
    the constraint into one of these forms), you’ll get an error when you try to solve the model. Constraints
    where the quadratic terms only involve binary variables will always be transformed
    into one of these forms.
    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 new quadratic constraint. Can be a constant, a Var, a LinExpr, or
      a QuadExpr.
    - sense: Sense for new quadratic constraint (GRB.LESS_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.
    Return value:
    New quadratic constraint object.
    Example usage:
    ```python
    model.addQConstr(x*x + y*y, GRB.LESS_EQUAL, z*z, "c0")
    model.addQConstr(x*x + y*y <= 2.0, "c1")
    ```

Model.addRange()

def 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.
    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.

**Return value:**
New constraint object.

**Example usage:**
```python
# 1 <= x + y <= 2
model.addRange(x + y, 1.0, 2.0, "range0")
```

**Model.addSOS()**

<table>
<thead>
<tr>
<th>addSOS ( type, vars, wts=None )</th>
</tr>
</thead>
</table>

Add an SOS constraint to the model.

**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:**
```python
model.addSOS(GRB.SOS_TYPE1, [x, y, z], [1, 2, 4])
```

**Model.addVar()**

<table>
<thead>
<tr>
<th>addVar ( lb=0.0, ub=GRB.INFINITY, obj=0.0, vtype=GRB.CONTINUOUS, name='', column=None )</th>
</tr>
</thead>
</table>

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.
- **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:**
```python
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.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 the change won’t take effect until the next call to Model.update or Model.optimize.

Argument:
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.computeIIS()

computeIIS ( void )

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result.

This method populates the IISCONSTR and IISQCONSTR 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.

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

Example usage:

model.computeIIS()
model.write("model.ilp")

Model.copy()

copy ( )

Copy a model.

Return value:
Copy of model.

Example usage:

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:**

```python
env0 = model.getConcurrentEnv(0)
env1 = model.getConcurrentEnv(1)

env0.setParam('Method', 0)
env1.setParam('Method', 1)

model.optimize()

model.discardConcurrentEnvs()
```

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 con-
tribute 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.
ubpen: Penalty for violating a variable upper bound. One entry for each variable in argument vars.
lbpen: Penalty for violating a variable lower bound. One entry for each variable in argument vars.
constr: Linear constraints that are allowed to be violated.
rhspen: Penalty for violating a linear constraint. One entry for each variable in argument constr.

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:

```python
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()

```
def 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.

Return value:
    Fixed model associated with calling object.

Example usage:

    fixed = model.fixed()
```

Model.getAttr()

```
def 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.

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", m.getVars())
print model.getAttr("Pi", m.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(x)
```

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: The concurrent environment number.

**Return value:**

The concurrent environment for the model.

**Example usage:**

```python
env0 = model.getConcurrentEnv(0)
env1 = model.getConcurrentEnv(1)

env0.setParam('Method', 0)
env0.setParam('Method', 1)

model.optimize()

model.discardConcurrentEnvs()
```

**Model.getConstrByName()**

| getConstrByName ( name ) |

Retrieve a constraint from its name. If multiple 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:**

```python
c0 = model.getConstrByName("c0")
```

**Model.getConstrs()**

| getConstrs ( ) |

Retrieve a list of all constraints in the model.

**Return value:**

All constraints in the model.

**Example usage:**
constrs = model.getConstrs()
c0 = constrs[0]

**Model.getObjective()**

getObjective ( )

Retrieve the model objective (as a linear or quadratic expression).

**Return value:**
The model objective. A LinExpr object for a linear model, or a QuadExpr object for a quadratic model.

**Example usage:**

```python
obj = model.getObjective()
print obj.getValue()
```

**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 the description of setPWLObj for additional information on how the points relate to the overall function.

**Arguments:**
- \( \text{var} \): 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:**

```python
> 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:**

```python
qconstrs = model.getQConstrs()
qc0 = qconstrs[0]
```
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.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.

Return value:
A QuadExpr object that captures the left-hand side of the quadratic constraint.

Example usage:
print model.getQCRow(m.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.

Return value:
A LinExpr object that captures the set of variables that participate in the constraint.

Example usage:
print model.getRow(c0)

386
**Model.getSOS()**

```java
getSOS ()
```

Retrieve a list of all SOS constraints in the model.

**Return value:**
All SOS constraints in the model.

**Example usage:**
```python
sos = model.getSOS()
```

**Model.getTuneResult()**

```java
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:**
```python
model.tune()
for i in range(model.tuneResultCount):
    model.getTuneResult(i)
    model.write('tune'+str(i)+'.prm')
```

**Model.getVarByName()**

```java
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:**
```python
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 )

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.

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.presolve()

presolve ( )

Perform presolve on a model.

Return value:
Presolved version of original model.

Example usage:
```python
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 associate 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:**
```python
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
```

```python
Model.printQuality()

```printQuality ()```

Print statistics about the quality of the computed solution (constraint violations, integrality violations, etc.).

**Example usage:**
```python
model.optimize()
model.printQuality()
```

```python
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:**
```python
model.printStats()
```
**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`.

When adding your own cuts, you must set parameter `PreCrush` to value 1. This setting shuts off a few presolve reductions that sometimes prevent cuts on the original model from being applied to the presolved model.

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.

**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:**

```python
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:**
```python
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 list of variables or a single variable.

Return value:
    The values of the specified variables in the node relaxation for the current node.

Example usage:

def mycallback(model, where):
    if where == GRB.callback.MIPNODE:
        print model.cbGetNodeRel(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 (see the Callback Codes section for more information).

Arguments:
    vars: The variables whose solution values are desired. Can be a list of variables or a single variable.

Return value:
    The values of the specified variables in the solution.

Example usage:

def mycallback(model, where):
    if where == GRB.callback.MIPSOL:
        print model.cbGetSolution(model.getVars())

model.optimize(mycallback)
```
Model.cbLazy()

Model.cbLazy() adds 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.CB_MIPSOL` callback, or `cbGetNodeRel` from a `GRB.CB_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.

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.

**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:**

```python
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.cbSetSolution()

Model.cbSetSolution() imports solution values for a heuristic solution. Only available when the where value on the callback function is equal to `GRB.CB_MIPNODE`. (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. At the end of 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.
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:

```python
def mycallback(model, where):
    if where == GRB.callback.MIPNODE:
        model.cbSetSolution(vars, newsolution)

model.optimize(mycallback)
```

**Model.read()**

```python
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 this isn’t 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` (for a MIP start), `.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:

```python
model.read('input.bas')
model.read('input.mst')
```

**Model.relax()**

```python
relax ( )
```

Create the relaxation of a MIP model. Transforms integer variables into continuous variables, and removes SOS constraints.

Return value:
Relaxed version of model.

Example usage:

```python
r = model.relax()
```
Model.remove()

remove ( item )

Remove a variable, linear constraint, quadratic constraint, or SOS from a model.

**Arguments:**
- **item**: The item to remove from the model. The item can be a Var, a Constr, QConstr, or an SOS

**Example usage:**
```
model.remove(model.getVars()[0])
model.remove(model.getConstrs()[0])
model.remove(model.getQConstrs()[0])
model.remove(model.getSOS()[0])
```

Model.reset()

reset ( )

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

**Example usage:**
```
model.reset()
```

Model.setAttr()

setAttr ( attrname, newvalue )

Set the value of an attribute. Note that attribute changes are handled in a lazy fashion. The effect of a change isn’t visible until the next call to Model.update or Model.optimize.

Call this method with two arguments to set a model attribute. Call it with three arguments to set the values of the attribute for a list of variables or constraints.

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

**Arguments:**
- **attrname**: Name of attribute to set.
- **newvalue**: Desired new value of attribute.

**Example usage:**
```
model.setAttr("objCon", 0)
model.setAttr(GRB.attr.objCon, 0)
model.setAttr("LB", m.getVars(), [0]*model.numVars)
model.setAttr("RHS", m.getConstrs(), [1.0]*model.numConstrs)
model.objcon = 0
```

Model.setObjective()

setObjective ( expr, sense=None )
Set the model objective equal to a linear or quadratic expression.
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:**
```python
model.setObjective(x + y, GRB.MAXIMIZE)
model.setObjective(x*x + y*y)
```

**Model.setPWLObj()**

```python
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 example, suppose we want to define the function $f(x)$ shown below:

![Graph of piecewise-linear function](image)

The vertices of the function occur at the points (1, 1), (3, 2) and (5, 4), so $x$ is [1, 3, 5] and $y$ is [1, 2, 4]. With these arguments 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 set of $n$ points

$$x = [x_1, \ldots, x_n], \quad y = [y_1, \ldots, y_n]$$
define the following piecewise-linear function:

\[
 f(v) = \begin{cases} 
 y_1 + \frac{y_2-y_1}{x_2-x_1}(v-x_1), & \text{if } v \leq x_1, \\
 y_i + \frac{y_{i+1}-y_i}{x_{i+1}-x_i}(v-x_i), & \text{if } v \geq x_i \text{ and } v \leq x_{i+1}, \\
 y_n + \frac{y_{n-1}-y_n}{x_n-x_{n-1}}(v-x_n), & \text{if } v \geq x_n. 
\end{cases}
\]

The \( x \) entries must appear in non-decreasing order. Two points can have the same \( x \) coordinate — this can be useful for specifying a discrete jump in the objective function.

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.

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.

Each variable can have its own piecewise-linear objective function. They must be specified individually, even if multiple variables share the same function.

**Arguments:**

- \( \text{var} \): 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:**

```python
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 \( \text{m} \), you can do either \( \text{m.setParam('MIPGap', 0)} \) or \( \text{m.params.MIPGap}=0 \).

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:**

- \( \text{paramname} \): 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.
- \( \text{newvalue} \): Desired new value for parameter. Can be 'default', which indicates that the parameter should be reset to its default value.

**Example usage:**

```python
model.setParam("heu*", 0.5)
model.setParam(GRB.param.heuristics, 0.5)
model.setParam("heu*", "default")
```
Model.terminate()

| terminate ( ) |

Generate a request to terminate the current optimization. This method is typically called from Model.cbGet. 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.

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 model data to a file. It can be used to write optimization models, IIS submodels, solutions, basis vectors, MIP start vectors, or parameter settings. The type of file is determined by the file suffix. File formats are described in the File Format section.

Arguments:
filename: Name of the file to write. The file type is encoded in the file name suffix. Valid suffixes for writing the model itself are .mps, .rew, .lp, or .rlp. An IIS can be written by using an .ilp suffix. Use .sol for a solution file, .mst for a MIP start, .bas for a basis file, or .prm for a parameter file. The suffix may optionally be followed by .gz, .bz2, or .7z, which produces a compressed result.

Example usage:
model.write("out.mst")
model.write("out.sol")
### 6.3 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` or `Model.optimize` 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:

\[
\text{expr1} = x + 2 \times y + 3 \times z + 4.0 \\
\text{expr2} = x \times x + 2 \times x \times y + 3 \times z + 4.0
\]

The first expression is linear, while the second is quadratic. An expressions is typically then passed to `setObjective` (to set the optimization objective) or `addConstr` (to add a constraint).

#### Var.getAttr()

<table>
<thead>
<tr>
<th>getAttr ( attrname )</th>
</tr>
</thead>
</table>

Query the value of a variable attribute. The full list of available attributes can be found in the Attributes section.

**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()

<table>
<thead>
<tr>
<th>sameAs ( var2 )</th>
</tr>
</thead>
</table>

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:**
```python
print model.getVars()[0].sameAs(model.getVars()[1])
```

**Var.setAttr()**

```
| setAttr  ( attrname, newvalue ) |
```

Set the value of a variable attribute. Note that attribute changes are handled in a lazy fashion. The effect of a change isn’t visible until the next call to `Model.update` or `Model.optimize`. The full list of available attributes can be found in the `Attributes` section.

**Arguments:**
- **attrname**: The attribute being modified.
- **newvalue**: The desired new value of the attribute.

**Example usage:**
```python
var.setAttr(GRB.attr.ub, 0.0)
var.setAttr("ub", 0.0)
```
6.4 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 or Model.optimize 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.

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.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 attribute changes are handled in a lazy fashion. The effect of a change isn’t visible until the next call to Model.update or Model.optimize. The full list of available attributes can be found in the Attributes section.

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)
6.5 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 or Model.optimize 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.

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 attribute changes are handled in a lazy fashion. The effect of a change isn’t visible until the next call to Model.update or Model.optimize.

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

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)
6.6 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 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 SOS.getAttr method.

**SOS.getAttr()**

```python
def getAttr( attrname )
```

Query the value of an SOS attribute. The full list of available attributes can be found in the Attributes section.

**Arguments:**
- `attrname`: The attribute being queried.

**Return value:**
- The current value of the requested attribute.

**Example usage:**
```python
print sos.getAttr(GRB.attr.IISSOS)
```
6.7 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 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., \( expr = 1 \times x + 2 \times y \)), or from other expressions (e.g., \( expr2 = 2 \times expr1 + x \), or \( expr3 = expr1 + 2 \times expr2 \)). 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 /. In Python parlance, we’ve defined the following LinExpr functions: add, radd, iadd, sub, rsub, isub, mul, rmul, imul, and div.

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 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.

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:

- While the Python sum function can be used to build expressions, it should be avoided. Its cost is quadratic in the length of the expression.
- For similar reasons, you should avoid using \( expr = expr + x \) in a loop. Building large expressions in this way also leads to quadratic runtimes.
- The quicksum function is much quicker than sum, as are loops over \( expr += x \) or expr.add(x). These approaches are fast enough for most programs, but they may still be expensive for very large expressions.
- The two most efficient ways to build large linear expressions are addTerms or the LinExpr() constructor.

Individual terms in a linear expression can be queried using the getVar, getCoef, 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. Note that 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:

define expr = LinExpr(2.0)
define expr = LinExpr(2*x)
define expr = LinExpr([1.0, 2.0], [x, y])
define 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:
define e1 = x + y
define 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:
define expr = x + 2 * y
define 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__()**

```python
__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:**
```python
m.addConstr(x + y == 1)
```

**LinExpr.__le__()**

```python
__le__ ( )
```

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:**
```python
m.addConstr(x + y <= 1)
```

**LinExpr.__ge__()**

```python
__ge__ ( arg )
```

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:**
```python
m.addConstr(x + y >= 1)
```
6.8 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 \( x \times x \) is a QuadExpr object. Expressions can be built from constants (e.g., \( \text{expr} = 0 \)), variables (e.g., \( \text{expr} = 1 \times x \times x + 2 \times x \times y \)), or from other expressions (e.g., \( \text{expr2} = 2 \times \text{expr1} \times x \times x \), or \( \text{expr3} = \text{expr1} + 2 \times \text{expr2} \)). You can also modify existing expressions (e.g., \( \text{expr} += x \times x \), or \( \text{expr2} -= \text{expr1} \)).

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\_\_, \_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 \text{sum} function. You can also use \text{add} or \text{addTerms} to modify expressions. Terms can be removed from an expression using \text{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:

- While the Python \text{sum} function can be used to build expressions, it should be avoided. Its cost is quadratic in the length of the expression.

- For similar reasons, you should avoid using \text{expr} = \text{expr} + x\times x\) in a loop. Building large expressions in this way also leads to quadratic runtimes.

- The \text{quicksum} function is much quicker than \text{sum}, as are loops over \text{expr} += x\times x\) or \text{expr}.add(x\times x). These approaches are fast enough for most programs, but they may still be expensive for very large expressions.

- The most efficient way to build a large quadratic expression is with a single call to \text{addTerms}.

Individual quadratic terms in a quadratic expression can be queried using the \text{getVar1}, \text{getVar2}, and \text{getCoeff} methods. You can query the number of quadratic terms in the expression using the \text{size} method. To query the constant and linear terms associated with a quadratic expression, use \text{getLinExpr} to obtain the linear portion of the quadratic expression, and then use the \text{getVar}, \text{getCoeff}, and \text{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 \text{getVar1} and \text{getVar2}).

\textbf{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:**
```python
e = QuadExpr()
e = QuadExpr(2*x)
e = 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:**
```python
e = x * x + 2 * y * y
e.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:**
```python
e = x * x + 2 * y * y
e.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:**
```
e = x * x + 2 * y * y + z
print e.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:**
```python
e = x * x + 2 * y * y + z
le = e.getLinExpr()
```

**QuadExpr.getValue()**

```python
g getValue ( )
```

Compute the value of an expression using the current solution.

**Return value:**
The value of the expression.

**Example usage:**
```python
obj = model.getObjective()
print obj.getValue()
```

**QuadExpr.getVar1()**

```python
g 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:**
```python
e = x * x + 2 * y * y + z
print e.getVar1(1)
```

**QuadExpr.getVar2()**

```python
g 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:**
```python
e = x * x + 2 * y * y + z
print e.getVar2(1)
```

**QuadExpr.remove()**

```python
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:**
\[ e = x \times x + 2 \times y \times y + z \]
\[ e . r e m o v e ( x ) \]

**QuadExpr.size()**

```python
size ( )
```

Retrieve the number of quadratic terms in the expression.

**Return value:**
Number of quadratic terms in the expression.

**Example usage:**
```python
e = x \times x + 2 \times y \times y + z
print e . s i z e ()
```

**QuadExpr.__eq__()**

```python
__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:**
```python
m . a d d C o n s t r ( x \times x + y \times y == 1 )
```

**QuadExpr.__le__()**

```python
__le__ ( )
```

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:**
```python
m . a d d C o n s t r ( x \times x + y \times y <= 1 )
```

**QuadExpr.__ge__()**

```python
__ge__ ( arg )
```

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:**
```python
m . a d d C o n s t r ( x \times x + y \times y >= 1 )
```
6.9 **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, LinExpr, and QuadExpr objects (e.g., ==, <=, and >=). You will generally never store objects of this class in your own variables.

Consider the following examples:

```java
model.addConstr(x + y <= 1);
model.addQConstr(x*x + y*y <= 1);
```

The overloaded <= operator creates an object of type TempConstr, which is then immediately passed to method Model.addConstr or Model.addQConstr.
6.10 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:
  col = Column()
  col = Column(3, c1)
  col = Column([[1.0, 2.0], [c1, c2]])

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()
6.11 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, 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. Your callback can call other callback methods to terminate the optimization, retrieve variable values for the current node relaxation, retrieve values for the new MIP solution, insert a new heuristic solution into the MIP search, or add a new MIP cutting plane.

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, note that 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.

You can look at callback.py in the examples directory for details of how to use Gurobi callbacks.
6.12 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.
6.13 Env

Gurobi environment object. Note that environments play a much smaller role in the Python interface than they do in other Gurobi language APIs. You will rarely need to work with environments. The two primary exceptions are when you use concurrent environments in one of the concurrent optimizers, or when you manage multiple Gurobi Compute Servers.

Env()

```
Env ( logfilename='', computeServers='', port=GRB.DEFAULT_CS_PORT,
     password='', priority=0, timeout=-1 )
```

Env constructor. Creates a client environment on a compute server. If all compute servers are at capacity, this command will cause a job to be placed in the compute server queue, and the command will return an environment once capacity is available.

Client environments have limited uses in the Python environment. You can use a client environment as an argument to the Model constructor, to indicate that a model should be constructed on a Compute Server, or as an argument to the global read function, to indicate that the result of reading the file should be placed on a Compute Server.

Arguments:

- **logfilename**: Name of the log file for this environment. Pass an an empty string if you don’t want a log file.
- **computeServers**: Comma-separated list of compute servers. Servers can be identified by name or by IP address.
- **port**: Port number on compute server. Use the default value unless your server administrator has informed you that a different value should be used.
- **password**: User password on compute server. Obtain this from your Compute Server administrator.
- **priority**: Job priority on the compute server. Higher priority jobs are pulled from the job queue before lower priority jobs. A special value of 100 indicates that the job should run immediately.
- **timeout**: Job queue timeout. After the specified timeout (in seconds) has elapsed, this command will give up and return a Gurobi exception. Use a negative value to indicate that the call should never timeout.

Return value:

- New environment object.

Example usage:

```
env = Env("client.log", "server1.mycompany.com,server2.mycompany.com")
m = read("misc07.mps", env)
m.optimize()
```

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, 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:**
```
    env.setParam("Cuts", 2)
    env.setParam("Heu*", 0.5)
    env.setParam("*Interval", 10)
```

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")
```
6.14  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).

# Model status codes (after call to optimize())

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

# Basis status info

BASIC = 0
NONBASIC_LOWER = -1
NONBASIC_UPPER = -2
SUPERBASIC = -3

# 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

# Numeric constants
INFINITY = 1e100
UNDEFINED = 1e101

# Other constants
DEFAULT_CS_PORT = 61000

# 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
**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`).
6.15  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.

While you can use a tuplelist anywhere you would normally use a list, we suggest that you only use them when you wish to populate the list with tuples and use the sub-list selection facilities provided by select.

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 should provide one value for each field in the member tuples. A '*' value indicates that any value is appropriate in that field.

Arguments:

pattern: Pattern to match for a member tuple.

Example usage:
l.select(1, 3, '*', 6)
l.select('A', '*', 'C')

tuplelist.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()
The Gurobi MATLAB® interface allows you to build an optimization model, pass the model to Gurobi, and obtain the optimization result, all from within the MATLAB environment. 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.

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.

**The Gurobi MATLAB API**

The Gurobi MATLAB interface is quite concise. It consists of just four MATLAB functions: `gurobi`, `gurobi_read`, `gurobi_write` and `gurobi_setup`. 
7.1 Solving models with the Gurobi MATLAB interface

The Gurobi MATLAB interface can be used to solve optimization problems of the following form:

\[
\begin{align*}
\text{minimize} & \quad x^T Q x + c^T x + \alpha \\
\text{subject to} & \quad Ax = b \\
& \quad \ell \leq x \leq u \\
& \quad \text{some } x_j \text{ integral} \\
& \quad \text{some } x_k \text{ lie within second order cones} \\
& \quad x^T Q c x + q^T x \leq \beta \\
& \quad \text{some } x_i \text{ in SOS}
\end{align*}
\]

subject to

- \(Ax = b\) (linear constraints)
- \(\ell \leq x \leq u\) (bound constraints)
- some \(x_j\) integral (integrality constraints)
- some \(x_k\) lie within second order cones (cone constraints)
- \(x^T Q c x + q^T x \leq \beta\) (quadratic constraints)
- some \(x_i\) in SOS (special ordered set constraints)

Many of the model components listed here are optional. For example, integrality constraints may be omitted. We'll discuss the details of how models are represented shortly.

The function `gurobi`, described next, allows you to take a model represented using MATLAB matrices and solve it with the Gurobi Optimizer.

\[\text{gurobi()}\]

---

The two arguments are MATLAB `struct` variables, each consisting of multiple fields. The first argument contains the optimization model to be solved. The second contains an optional set of Gurobi parameters to be modified during the solution process. The return value of this function is a `struct`, also consisting of multiple fields. It contains the result of performing the optimization on the specified model. We’ll now discuss the details of each of these data structures.

**The optimization model**

As we’ve mentioned, the `model` argument to the `gurobi` function is a `struct` variable, containing multiple fields that represent the various parts of the optimization model. Several of these fields are optional. Note that you refer to a field of a MATLAB `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 following is an enumeration of all of the fields of the `model` argument that Gurobi will take into account when optimizing the model:

- **A**: The linear constraint matrix. This must be a sparse matrix.
- **obj**: The linear objective vector (\(c\) in the problem statement). You must specify one value for each column of \(A\). This must be a dense vector.
- **sense**: 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. This must be a char array.
- **rhs**: The right-hand side vector for the linear constraints (\(b\) in the problem statement). You must specify one value for each row of \(A\). This must be a dense vector.
- **lb** (optional): The lower bound vector. When present, you must specify one value for each column of \(A\). This must be a dense vector. When absent, each variable has a lower bound of 0.
ub (optional): The upper bound vector. When present, you must specify one value for each column of $A$. This must be a dense vector. When absent, the variables have infinite upper bounds.

vtype (optional): The variable types. This char array 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. 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.

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).

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 array, 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 $A$. This must be a dense vector.

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 $A$. This must be a dense vector.

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 $A$. $Q$ must be a sparse matrix.

cones (optional): Second-order cone constraints. A struct array. Each element in the array defines a single cone constraint: $x(k)^2 \geq \text{sum}(x(idx).^2)$, $x(k) \geq 0$. The constraint is defined via model.cones.index = [k idx], with the first entry in index corresponding to the index of the variable on the left-hand side of the constraint, and the remaining entries corresponding to the indices of the variables on the right-hand side of the constraint. model.cones.index must be a dense vector.

quadcon (optional): The quadratic constraints. A struct array. When present, each element in the array defines a single quadratic constraint: $x^TQc x + q^T x \leq \beta$. The Qc matrix must be a square matrix whose row and column counts are equal to the number of columns of $A$. Qc must be a sparse matrix. It is stored in model.quadcon.Qc. The $q$ vector defines the linear terms in the constraint. You must specify a value for $q$ for each column of $A$. This must be a
dense vector. It is stored in `model.quadcon.q`. The scalar beta defines the right-hand side of the constraint. It is stored in `model.quadcon.rhs`.

**sos** (optional): The Special Ordered Set (SOS) constraints. A struct array. When present, each element in the array defines a single SOS constraint. A SOS constraint can be of type 1 or 2. This is specified via `model.sos.type`. 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-zeros values, they must be contiguous in the ordered set. The members of an SOS constraint are specified by placing their indices in `model.sos.index`. Optional weights associated with SOS members may be defined in `model.sos.weight`.

**pwlobj** (optional): The piecewise-linear objective functions. A struct array. When present, each element in the array defines a piecewise-linear objective function of a single variable. The index of the variable whose objective function is being defined is stored in `model.pwlobj.var`. The $x$ values for the points that define the piecewise-linear function are stored in `model.pwlobj.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.y`.

**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. This must be a dense vector. Note that you can leave the start value for a variable undefined—the MIP solver will attempt to fill in values for the undefined start values. This may be done by setting the start value for that variable to `nan`.

**varnames** (optional): The variable names. A cell array of strings. When present, each element of the array defines the name of a variable. You must specify a name for each column of $A$.

**constrnames** (optional): The constraint names. A cell array of strings. When present, each element of the array defines the name of a constraint. You must specify a name for each row of $A$.

If any of the mandatory fields listed above are missing, the `gurobi` function will return an error.

Below is an example that demonstrates the construction of a simple optimization model:

```matlab
model.A = sparse([1 2 3; 1 1 0]);
model.obj = [1 1 2];
model.modelsense = 'max';
model.rhs = [4; 1];
model.sense = '<>';
```

**Parameters**

The optional `params` argument to the `gurobi` function is also a `struct`, potentially containing multiple fields. 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. Gurobi parameters allow users to modify the default behavior of the Gurobi optimization algorithms. You can find a complete list of the available Gurobi parameters [here](https://www.gurobi.com/products/gurobi-optimizer/parameters).

To create a struct that would set the Gurobi `method` parameter to 2 you would do the following:

```matlab
params.method = 2;
```
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, and the status of the optimization. The following is a list of fields that might be available in the returned result. We’ll discuss the circumstances under which each will be available after presenting the list.

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.

runtime: The elapsed wall-clock time (in seconds) for the optimization.

x: The computed solution. This array contains one entry for each column of A.

slack: The constraint slack for the computed solution. This array contains one entry for each row of A.

qcslack: The quadratic constraint slack in the current solution. This array contains one entry for second-order cone constraint and one entry for each quadratic constraint. The slacks for the second-order cone constraints appear before the slacks for the quadratic constraints.

pi: Dual values for the computed solution (also known as shadow prices). This array contains one entry for each row of A.

qcpi: The dual values associated with the quadratic constraints. This array contains one entry for each second-order cone constraint and one entry for each quadratic constraint. The dual values for the second-order cone constraints appear before the dual values for the quadratic constraints.

rc: Variable reduced costs for the computed solution. This array 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 array. If you wish to use an advanced start later, you would simply copy the vbasis and cbasis arrays into the corresponding fields for the next model. This array contains one entry for each column of A.

cbasis: Constraint basis status values for the computed optimal basis. This array contains one entry for each row 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.

farkasdual: Farkas infeasibility proof. This is a dual unbounded vector. Adding this vector to any feasible solution of the dual model yields a new solution that is also feasible but improves the dual objective.
farkasproof: Magnitude of infeasibility violation in Farkas infeasibility proof. A Farkas infeasibility proof identifies a new constraint, obtained by taking a linear combination of the constraints in the model, that can never be satisfied. (the linear combination is available in the farkasdual field). This attribute indicates the magnitude of the violation of this aggregated constraint.

objbound: Best available bound on solution (lower bound for minimization, upper bound for maximization).

itercount: Number of simplex iterations performed.

baritercount: Number of barrier iterations performed.

nodecount: Number of branch-and-cut nodes explored.

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.

The following is an example of how the results of the gurobi call might be extracted and output:

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
7.2 Reading and writing models with the Gurobi MATLAB interface

The MATLAB interface contains functions to read and write model files.

**gurobi_read()**

| gurobi_read ( filename ) |

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, or .ilp (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. 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.

**Return value:**
A model struct containing multiple named fields. See the gurobi function for a description of these fields and their contents.

**Example usage:**

```matlab
model = gurobi_read('etamacro.mps');
result = gurobi(model)
```

**gurobi_write()**

| gurobi_write ( model, filename ) |

Writes a model to a file.

**Arguments:**
- **model**: The model struct must contain a valid Gurobi model. See the gurobi function for a description of model’s required fields and values.
- **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, 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.

**Example usage:**

```matlab
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');
```
7.3 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 6.0 in the default location, you should run

```
>> cd c:/Users/jones/gurobi600/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/gurobi600/win64/examples/matlab
>> mip1
```

If the Gurobi package was successfully installed, you should see the following output:

```
status: 'OPTIMAL'
versioninfo: [1x1 struct]
  objval: 3
  runtime: 0.0386
  x: [3x1 double]
  slack: [2x1 double]
  objbound: 3
  itercount: 0
  baritercount: 0
  nodecount: 0

  x 1
  y 0
  z 1

Obj: 3.000000e+00
```
The Gurobi R interface allows you to build an optimization model, pass the model to Gurobi, and obtain the optimization result, all from within the R environment. For those of you who are not familiar with R, it is a free language for statistical computing. Please visit the R Project web site for more information.

The Gurobi R interface can be used to solve optimization problems of the following form:

minimize \( x^T Q x + c^T x + \alpha \)
subject to \( Ax = b \) \hspace{1cm} \text{(linear constraints)}
\( \ell \leq x \leq u \) \hspace{1cm} \text{(bound constraints)}
some \( x_j \) integral \hspace{1cm} \text{(integrality constraints)}
some \( x_k \) lie within second order cones \hspace{1cm} \text{(cone constraints)}

Many of the model components listed here are optional. For example, integrality constraints or second order cone constraints may be omitted. We’ll discuss the details of how models are represented shortly.

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.

The Gurobi R API

The Gurobi R interface is quite concise. It consists of a single R function that takes a pair of arguments:

\[
gurobi \quad ( \text{model, params=NULL} )
\]

The two arguments are R list variables, each consisting of multiple named components. The first argument contains the optimization model to be solved. The second contains an optional list of Gurobi parameters to be modified during the solution process. The return value of this function is a list, also consisting of multiple named components. It contains the result of performing the optimization on the specified model. We’ll now discuss the details of each of these lists.

The optimization model

As we’ve mentioned, the model argument to the gurobi() function is a list variable, containing multiple named components that represent the various parts of the optimization model. Several of these components are optional. Note that you refer to a named component of an R list variable by appending a dollar sign followed by the component name to the list variable name. For example, model$A refers to component A of list variable model.

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:

A: The linear constraint matrix. This 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.
obj: The linear objective vector (the $c$ vector in the problem statement above). You must specify one value for each column of $A$.

sense: The senses of the linear constraints. Allowed values are '=' , '<=', or '>='. You must specify one value for each row of $A$.

rhs: The right-hand side vector for the linear constraints (the $b$ vector in the problem statement above). You must specify one value for each row of $A$.

lb (optional): The lower bound vector. When present, you must specify one value for each column of $A$. When absent, each variable has a 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 type vector. 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 $A$. When absent, each variable is treated as being continuous.

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 above).

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.

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 array, 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 $A$.

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 $A$. 
cones (optional): Second-order cone constraints. A list of lists. Each member list defines a single cone constraint: \( \sum x_i^2 \leq y^2 \). The first integer in the list gives the column index for variable \( y \), and the remainder give the column indices for the \( x \) variables.

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:

```r
model <- list()
model$A <- matrix(c(1,1,0,0,1,1), nrow=2, byrow=T)
model$obj <- c(1,1,2)
model$modelsense <- "max"
model$rhs <- c(1,1)
model$sense <- c("<=" , '<="')
```

You can also build \( A \) as a sparse matrix, using either `sparseMatrix` or `simple_triplet_matrix`:

```r
model$A <- spMatrix(2, 3, c(1, 1, 2), c(1, 2, 2), c(1, 1, 1))
model$A <- simple_triplet_matrix(c(1, 1, 2, 2), c(1, 2, 2, 3), c(1, 1, 1))
```

Note that the Gurobi interface allows you to specify a scalar value for any 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$rhs <- 1` would be equivalent to `model$rhs <- c(1,1)` in the example).

The optional `params` argument to the `gurobi()` function is also a list of named components. For each component, the name should be the name of a Gurobi parameter, and the associated value should be the desired value of that parameter. Gurobi parameters allow users to modify the default behavior of the Gurobi optimization algorithms. You can find a complete list of the available Gurobi parameters here.

To create a list that would set the Gurobi `Method` parameter to 2 and the `ResultFile` parameter parameter to `model.mps`, you would do the following:

```r
params <- list(Method=2, 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`, or `.rlp`, to indicate the desired file format (see the file formats section for details on Gurobi file formats).

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, and the status of the optimization. The following is a list of components that might be available in the result list. We'll discuss the circumstances under which each will be available after presenting the list.
**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.

**x**: The computed solution. This array contains one entry for each column of $A$.

**slack**: Constraint slacks for the computed solution. This array contains one entry for each row of $A$.

**pi**: Dual values for the computed solution (also known as shadow prices). This array contains one entry for each row of $A$.

**rc**: Variable reduced costs for the computed solution. This array 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 array. If you wish to use an advanced start later, you would simply copy the $vbasis$ and $cbasis$ arrays into the corresponding components for the next model. This array contains one entry for each column of $A$.

**cbasis**: Constraint basis status values for the computed optimal basis. This array contains one entry for each row of $A$.

The **status** 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**, **x**, and **slack** components will be present. For linear and quadratic programs, if a solution is available, then the **pi** and **rc** components will also be present. Finally, if the final solution is a basic solution (computed by simplex), then **vbasis** and **cbasis** will be present.

The following is an example of how the results of the `gurobi()` call might be extracted and output:

```r
result <- gurobi(model, params)
print(result$objval)
print(result$x)
```

**Installing the R package**

To use our R interface, you’ll need to install the Gurobi package in R. The R command for doing this is:

```r
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 6.0.0 is `/opt/gurobi600/linux64` for Linux, `c:\gurobi600` for Windows).
for Windows, and /Library/gurobi600 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_6.0-0_R_x86_64-pc-linux-gnu.tar.gz, the Windows package is in file gurobi_6.0-0.zip, and the Mac package is in file gurobi_6.0-0.tgz).

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:

```r
> 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 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 type: model attributes take scalar values, while variable, constraint, and SOS attributes contain one entry per variable, constraint, or SOS 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 \texttt{VBasis} and \texttt{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 (\texttt{SAObjLow}, \texttt{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).

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumVars</td>
<td>Number of variables</td>
</tr>
<tr>
<td>NumConstrs</td>
<td>Number of linear constraints</td>
</tr>
<tr>
<td>NumSOS</td>
<td>Number of SOS constraints</td>
</tr>
<tr>
<td>NumQConstrs</td>
<td>Number of quadratic constraints</td>
</tr>
<tr>
<td>NumNZs</td>
<td>Number of non-zero coefficients in the constraint matrix</td>
</tr>
<tr>
<td>NumQNZs</td>
<td>Number of non-zero quadratic objective terms</td>
</tr>
<tr>
<td>NumQCNZs</td>
<td>Number of non-zero terms in quadratic constraints</td>
</tr>
<tr>
<td>NumIntVars</td>
<td>Number of integer variables</td>
</tr>
<tr>
<td>NumBinVars</td>
<td>Number of binary variables</td>
</tr>
<tr>
<td>NumPWLObjVars</td>
<td>Number of variables with piecewise-linear objective functions.</td>
</tr>
<tr>
<td>ModelName</td>
<td>Model name</td>
</tr>
<tr>
<td>ModelSense</td>
<td>Model sense (minimization or maximization)</td>
</tr>
<tr>
<td>ObjCon</td>
<td>Constant offset for objective function</td>
</tr>
<tr>
<td>ObjVal</td>
<td>Objective value for current solution</td>
</tr>
<tr>
<td>ObjBound</td>
<td>Best available objective bound (lower bound for minimization, upper bound for maximization)</td>
</tr>
<tr>
<td>ObjBoundC</td>
<td>Best available objective bound, without rounding (lower bound for minimization, upper bound for maximization)</td>
</tr>
<tr>
<td>MIPGap</td>
<td>Current relative MIP optimality gap</td>
</tr>
<tr>
<td>Runtime</td>
<td>Runtime for most recent optimization</td>
</tr>
<tr>
<td>Status</td>
<td>Current optimization status</td>
</tr>
<tr>
<td>SolCount</td>
<td>Number of solutions found</td>
</tr>
<tr>
<td>IterCount</td>
<td>Number of simplex iterations performed in most recent optimization</td>
</tr>
<tr>
<td>BarIterCount</td>
<td>Number of barrier iterations performed in most recent optimization</td>
</tr>
<tr>
<td>NodeCount</td>
<td>Number of branch-and-cut nodes explored in most recent optimization</td>
</tr>
<tr>
<td>IsMIP</td>
<td>Indicates whether the model is a MIP</td>
</tr>
<tr>
<td>IsQP</td>
<td>Indicates whether the model is a QP/MIQP</td>
</tr>
<tr>
<td>IsQCP</td>
<td>Indicates whether the model is a QCP/MIQCP</td>
</tr>
<tr>
<td>IISMinimal</td>
<td>Indicates whether the current IIS is minimal</td>
</tr>
<tr>
<td>MaxCoeff</td>
<td>Maximum constraint matrix coefficient (in absolute value)</td>
</tr>
<tr>
<td>MinCoeff</td>
<td>Minimum (non-zero) constraint matrix coefficient (in absolute value)</td>
</tr>
<tr>
<td>MaxBound</td>
<td>Maximum finite variable bound</td>
</tr>
<tr>
<td>MinBound</td>
<td>Minimum finite variable bound</td>
</tr>
<tr>
<td>MaxObjCoeff</td>
<td>Maximum linear objective coefficient (in absolute value)</td>
</tr>
<tr>
<td>MinObjCoeff</td>
<td>Minimum (non-zero) linear objective coefficient (in absolute value)</td>
</tr>
<tr>
<td>MaxRHS</td>
<td>Maximum constraint right-hand side (in absolute value)</td>
</tr>
<tr>
<td>MinRHS</td>
<td>Minimum (non-zero) constraint right-hand side (in absolute value)</td>
</tr>
<tr>
<td>Kappa</td>
<td>Estimated basis condition number</td>
</tr>
<tr>
<td>KappaExact</td>
<td>Exact basis condition number</td>
</tr>
<tr>
<td>FarkasProof</td>
<td>Magnitude of infeasibility violation in Farkas infeasibility proof</td>
</tr>
<tr>
<td>TuneResultCount</td>
<td>Number of improved parameter sets found by tuning tool</td>
</tr>
</tbody>
</table>
Variable attributes:
These attributes provide information that is associated with specific variables.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>Lower bound</td>
</tr>
<tr>
<td>UB</td>
<td>Upper bound</td>
</tr>
<tr>
<td>Obj</td>
<td>Linear objective coefficient</td>
</tr>
<tr>
<td>VType</td>
<td>Variable type (continuous, binary, integer, etc.)</td>
</tr>
<tr>
<td>VarName</td>
<td>Variable name</td>
</tr>
<tr>
<td>X</td>
<td>Value in the current solution</td>
</tr>
<tr>
<td>Xn</td>
<td>Value in a sub-optimal MIP solution</td>
</tr>
<tr>
<td>RC</td>
<td>Reduced cost</td>
</tr>
<tr>
<td>Start</td>
<td>MIP start value (for constructing an initial MIP solution)</td>
</tr>
<tr>
<td>BranchPriority</td>
<td>Branching priority</td>
</tr>
<tr>
<td>VBasis</td>
<td>Basis status</td>
</tr>
<tr>
<td>PStart</td>
<td>Simplex start vector</td>
</tr>
<tr>
<td>IISLB</td>
<td>Indicates whether the lower bound participate in the IIS</td>
</tr>
<tr>
<td>IISUB</td>
<td>Indicates whether the upper bound participate in the IIS</td>
</tr>
<tr>
<td>PWLObjCvx</td>
<td>Indicates whether the variable has a convex piecewise-linear objective</td>
</tr>
<tr>
<td>SAObjLow</td>
<td>Objective coefficient sensitivity information</td>
</tr>
<tr>
<td>SAObjUp</td>
<td>Objective coefficient sensitivity information</td>
</tr>
<tr>
<td>SALBLow</td>
<td>Lower bound sensitivity information</td>
</tr>
<tr>
<td>SALBUp</td>
<td>Lower bound sensitivity information</td>
</tr>
<tr>
<td>SAUBLow</td>
<td>Upper bound sensitivity information</td>
</tr>
<tr>
<td>SAUBUp</td>
<td>Upper bound sensitivity information</td>
</tr>
<tr>
<td>UnbdRay</td>
<td>Unbounded ray</td>
</tr>
</tbody>
</table>
**Linear constraint attributes:**

These attributes provide information that is associated with specific linear constraints.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sense</td>
<td>Constraint sense (‘&lt;’, ‘&gt;’, or ‘=’)</td>
</tr>
<tr>
<td>RHS</td>
<td>Right-hand side value</td>
</tr>
<tr>
<td>ConstrName</td>
<td>Constraint name</td>
</tr>
<tr>
<td>Pi</td>
<td>Dual value (also known as the <em>shadow price</em>)</td>
</tr>
<tr>
<td>Slack</td>
<td>Slack in the current solution</td>
</tr>
<tr>
<td>CBasis</td>
<td>Basis status</td>
</tr>
<tr>
<td>DStart</td>
<td>Simplex start vector</td>
</tr>
<tr>
<td>Lazy</td>
<td>Determines whether a constraint is treated as a lazy constraint</td>
</tr>
<tr>
<td>IISConstr</td>
<td>Indicates whether the constraint participates in the IIS</td>
</tr>
<tr>
<td>SARHSLow</td>
<td>Right-hand-side sensitivity information</td>
</tr>
<tr>
<td>SARHSUp</td>
<td>Right-hand-side sensitivity information</td>
</tr>
<tr>
<td>FarkasDual</td>
<td>Farkas infeasibility proof</td>
</tr>
</tbody>
</table>
**SOS attributes:**

These attributes provide information that is associated with specific Special-Ordered Set (SOS) constraints.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IISSOS</td>
<td>Indicates whether the SOS constraint participates in the IIS</td>
</tr>
</tbody>
</table>
**Quadratic constraint attributes:**

These attributes provide information that is associated with specific quadratic constraints.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QCSense</td>
<td>Constraint sense (‘&lt;’, ‘&gt;’, or ‘=’).</td>
</tr>
<tr>
<td>QCRHS</td>
<td>Right-hand side.</td>
</tr>
<tr>
<td>QCName</td>
<td>Quadratic constraint name.</td>
</tr>
<tr>
<td>QCPi</td>
<td>Dual value.</td>
</tr>
<tr>
<td>QCSlack</td>
<td>Slack in the current solution.</td>
</tr>
<tr>
<td>IISQConstr</td>
<td>Indicates whether the quadratic constraint participates in the IIS.</td>
</tr>
</tbody>
</table>
Solution quality attributes:

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundVio</td>
<td>Maximum (unscaled) bound violation</td>
</tr>
<tr>
<td>BoundSVio</td>
<td>Maximum (scaled) bound violation</td>
</tr>
<tr>
<td>BoundVioIndex</td>
<td>Index of variable with the largest (unscaled) bound violation</td>
</tr>
<tr>
<td>BoundSVioIndex</td>
<td>Index of variable with the largest (scaled) bound violation</td>
</tr>
<tr>
<td>BoundVioSum</td>
<td>Sum of (unscaled) bound violations</td>
</tr>
<tr>
<td>BoundSVioSum</td>
<td>Sum of (scaled) bound violations</td>
</tr>
<tr>
<td>ConstrVio</td>
<td>Maximum (unscaled) constraint violation</td>
</tr>
<tr>
<td>ConstrSVio</td>
<td>Maximum (scaled) constraint violation</td>
</tr>
<tr>
<td>ConstrVioIndex</td>
<td>Index of constraint with the largest (unscaled) violation</td>
</tr>
<tr>
<td>ConstrSVioIndex</td>
<td>Index of constraint with the largest (scaled) violation</td>
</tr>
<tr>
<td>ConstrVioSum</td>
<td>Sum of (unscaled) constraint violations</td>
</tr>
<tr>
<td>ConstrSVioSum</td>
<td>Sum of (scaled) constraint violations</td>
</tr>
<tr>
<td>ConstrResidual</td>
<td>Maximum (unscaled) primal constraint error</td>
</tr>
<tr>
<td>ConstrSResidual</td>
<td>Maximum (scaled) primal constraint error</td>
</tr>
<tr>
<td>ConstrResidualIndex</td>
<td>Index of constraint with the largest (unscaled) primal constraint error</td>
</tr>
<tr>
<td>ConstrSResidualIndex</td>
<td>Index of constraint with the largest (scaled) primal constraint error</td>
</tr>
<tr>
<td>ConstrResidualSum</td>
<td>Sum of (unscaled) primal constraint errors</td>
</tr>
<tr>
<td>ConstrSResidualSum</td>
<td>Sum of (scaled) primal constraint errors</td>
</tr>
<tr>
<td>DualVio</td>
<td>Maximum (unscaled) reduced cost violation</td>
</tr>
<tr>
<td>DualSVio</td>
<td>Maximum (scaled) reduced cost violation</td>
</tr>
<tr>
<td>DualVioIndex</td>
<td>Index of variable with the largest (unscaled) reduced cost violation</td>
</tr>
<tr>
<td>DualSVioIndex</td>
<td>Index of variable with the largest (scaled) reduced cost violation</td>
</tr>
<tr>
<td>DualVioSum</td>
<td>Sum of (unscaled) reduced cost violations</td>
</tr>
<tr>
<td>DualSVioSum</td>
<td>Sum of (scaled) reduced cost violations</td>
</tr>
<tr>
<td>DualResidual</td>
<td>Maximum (unscaled) dual constraint error</td>
</tr>
<tr>
<td>DualSResidual</td>
<td>Maximum (scaled) dual constraint error</td>
</tr>
<tr>
<td>DualResidualIndex</td>
<td>Index of variable with the largest (unscaled) dual constraint error</td>
</tr>
<tr>
<td>DualSResidualIndex</td>
<td>Index of variable with the largest (scaled) dual constraint error</td>
</tr>
<tr>
<td>DualResidualSum</td>
<td>Sum of (unscaled) dual constraint errors</td>
</tr>
<tr>
<td>DualSResidualSum</td>
<td>Sum of (scaled) dual constraint errors</td>
</tr>
<tr>
<td>ComplVio</td>
<td>Maximum complementarity violation</td>
</tr>
<tr>
<td>ComplVioIndex</td>
<td>Index of variable with the largest complementarity violation</td>
</tr>
<tr>
<td>ComplVioSum</td>
<td>Sum of complementarity violations</td>
</tr>
<tr>
<td>IntVio</td>
<td>Maximum integrality violation</td>
</tr>
<tr>
<td>IntVioIndex</td>
<td>Index of variable with the largest integrality violation</td>
</tr>
<tr>
<td>IntVioSum</td>
<td>Sum of integrality violations</td>
</tr>
</tbody>
</table>

### 9.1 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.

**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.

**NumNZs**

- **Type:** int
- **Modifiable:** No

  The number of non-zero coefficients in the linear constraints of the model.

  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 +1.0 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.

**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.

**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.

**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 solutions found during 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.
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, and semi-integer variables, and SOS constraints.
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.

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 (finite) 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.

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
Magnitude of infeasibility violation in Farkas infeasibility proof (for infeasible linear models
only). A Farkas infeasibility proof identifies a new constraint, obtained by taking a linear com-
bination of the constraints in the model, that can never be satisfied (the linear combination is
available in the FarkasDual attribute). This attribute indicates the magnitude of the violation of
this aggregated constraint. 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 tools 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.

9.2 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 in-
terfaces will throw an exception.

LB

Type: double
Modifiable: Yes
Variable lower bound. Note that any value less than -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 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.

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.

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.

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.

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 continuous models.

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 SubMIPNodes parameter to a larger value if you want Gurobi to work harder to try to complete the partial start.

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.

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 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. The default variable branch priority value is zero.

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.

For examples of how to query or modify attributes, refer to our Attribute Examples.

**PStart**

Type: double
Modifiable: Yes

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. If you’d like to retract a previously specified start, set any PStart value to GRB_UNDEFINED.

Note that any model modifications 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.

Only affects LP models; it will be ignored for QP, QCP, or MIP models.

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). Only available after you have computed an IIS.

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). Only available after you have computed an IIS.

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 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 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.

**SAUBLow**

*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.
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.

### 9.3 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.

**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.

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.

**Pi**

*Type:* double  
*Modifiable:* No  
The constraint dual value in the current solution (also known as the *shadow price*).  
Given a linear programming problem

\[
\begin{align*}
\text{minimize} & \quad c'x \\
\text{subject to} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
\]

and a corresponding dual problem

\[
\begin{align*}
\text{maximize} & \quad b'y \\
\text{subject to} & \quad A'y \leq c \\
& \quad y \geq 0
\end{align*}
\]

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 \( \geq 0 \).
- Dual values for \( \leq \) constraints are \( \leq 0 \).
- Dual values for \( = \) constraints are unconstrained.

For models with a maximization sense, the senses of the dual values are reversed: the dual is \( \geq 0 \) for a \( \leq \) constraint and \( \leq 0 \) for a \( \geq \) constraint.

Only available for 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.

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 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.

Only affects LP models; it will be ignored for QP, QCP, or MIP models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

Lazy

**Type:** int  
**Modifiable:** No

Determines whether a linear constraint is treated as a lazy constraint. 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 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 of 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 are also pulled in.

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.

**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  
Farkas infeasibility proof (for infeasible linear models only). Provides a dual unbounded vector. Adding this vector into any feasible solution to the dual model yields a new solution that is also feasible but improves the dual objective. Only available when parameter InfUnbdInfo is set to 1.

For examples of how to query or modify attributes, refer to our Attribute Examples.

### 9.4 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.
9.5 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.
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.
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.
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.

9.6 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, 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.

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.

BoundViolIndex

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, constraint violations are 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.
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.
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 \texttt{DualVio} and the latter as \texttt{DualResidual}.

\texttt{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.

\texttt{DualSResidual} reports the maximum (scaled) dual constraint error.

Only available for continuous models.

For examples of how to query or modify attributes, refer to our Attribute Examples.

\texttt{DualResidualIndex} reports the index of the 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.

\texttt{DualSResidualIndex} reports the index of the 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.

\texttt{DualResidualSum} reports the 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.

### 9.7 Attribute Examples

The same attributes exist in all of the Gurobi APIs, but the approaches used to query and modify them, and the means by which you refer to them vary. 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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GRB_DBL_ATTR_LB</td>
</tr>
<tr>
<td>C++</td>
<td>GRB_DoubleAttr_LB</td>
</tr>
<tr>
<td>Java</td>
<td>GRB.DoubleAttr.LB</td>
</tr>
<tr>
<td>.NET</td>
<td>GRB.DoubleAttr.LB</td>
</tr>
<tr>
<td>Python</td>
<td>GRB.attr.lb, or just var.lb</td>
</tr>
</tbody>
</table>

To query the value of this attribute for an individual variable in the different API’s, you would do the following:

<table>
<thead>
<tr>
<th>Language</th>
<th>Attribute Query Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GRBgetdblattrelement(model, GRB_DBL_ATTR_LB, var_index, &amp;value);</td>
</tr>
<tr>
<td>C++</td>
<td>var.get(GRB_DoubleAttr_LB)</td>
</tr>
<tr>
<td>Java</td>
<td>var.get(GRB.DoubleAttr.LB)</td>
</tr>
<tr>
<td>.NET</td>
<td>var.Get(GRB.DoubleAttr.LB)</td>
</tr>
<tr>
<td>Python</td>
<td>var.getAttr(GRB.attr.lb), or just var.lb</td>
</tr>
</tbody>
</table>

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:

```c
#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 API’s:

- C
- C++
- C#
- Java
- Python
- Visual Basic

You can 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:

```c
int cols;
error = GRBgetintattr(m, GRB_INT_ATTR_NUMVARS, &cols);
```

You can also use the name of the attribute directly:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:

```c++
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:

```c++
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:

```c++
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`:

```c++
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:

```c++
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 using the `Get` method:

```csharp
    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:

```csharp
    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:

```csharp
    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`:

```csharp
    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:

```csharp
    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).

**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:

```java
    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:

```java
    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:

```java
    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`:

```java
    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:

```python
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:

```python
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:

```python
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:

```python
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`:

```python
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:

```python
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 using the Get method:

```vbnet
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:

```vbnet
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:

```csharp
vars = m.GetVars()
For j As Integer = 0 To cols - 1
    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:

```csharp
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:

```csharp
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).
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.

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).

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarIterLimit</td>
<td>Barrier iteration limit</td>
</tr>
<tr>
<td>Cutoff</td>
<td>Objective cutoff</td>
</tr>
<tr>
<td>IterationLimit</td>
<td>Simplex iteration limit</td>
</tr>
<tr>
<td>NodeLimit</td>
<td>MIP node limit</td>
</tr>
<tr>
<td>SolutionLimit</td>
<td>MIP feasible solution limit</td>
</tr>
<tr>
<td>TimeLimit</td>
<td>Time limit</td>
</tr>
</tbody>
</table>

**Tolerances:** These parameters control the allowable feasibility or optimality violations.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarConvTol</td>
<td>Barrier convergence tolerance</td>
</tr>
<tr>
<td>BarQCPConvTol</td>
<td>Barrier QCP convergence tolerance</td>
</tr>
<tr>
<td>FeasibilityTol</td>
<td>Primal feasibility tolerance</td>
</tr>
<tr>
<td>IntFeasTol</td>
<td>Integer feasibility tolerance</td>
</tr>
<tr>
<td>MarkowitzTol</td>
<td>Threshold pivoting tolerance</td>
</tr>
<tr>
<td>MIPGap</td>
<td>Relative MIP optimality gap</td>
</tr>
<tr>
<td>MIPGapAbs</td>
<td>Absolute MIP optimality gap</td>
</tr>
<tr>
<td>OptimalityTol</td>
<td>Dual feasibility tolerance</td>
</tr>
<tr>
<td>PSDTol</td>
<td>Positive semi-definite tolerance</td>
</tr>
</tbody>
</table>
**Simplex:** These parameters control the operation of the simplex algorithms.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfUnbdInfo</td>
<td>Generate additional info for infeasible/unbounded models</td>
</tr>
<tr>
<td>NormAdjust</td>
<td>Simplex pricing norm</td>
</tr>
<tr>
<td>ObjScale</td>
<td>Objective scaling</td>
</tr>
<tr>
<td>PerturbValue</td>
<td>Simplex perturbation magnitude</td>
</tr>
<tr>
<td>Quad</td>
<td>Quad precision computation in simplex</td>
</tr>
<tr>
<td>ScaleFlag</td>
<td>Model scaling</td>
</tr>
<tr>
<td>Sifting</td>
<td>Sifting within dual simplex</td>
</tr>
<tr>
<td>SiftMethod</td>
<td>LP method used to solve sifting sub-problems</td>
</tr>
<tr>
<td>SimplexPricing</td>
<td>Simplex variable pricing strategy</td>
</tr>
</tbody>
</table>

**Barrier:** These parameters control the operation of the barrier solver.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarCorrectors</td>
<td>Central correction limit</td>
</tr>
<tr>
<td>BarHomogeneous</td>
<td>Barrier homogeneous algorithm</td>
</tr>
<tr>
<td>BarOrder</td>
<td>Barrier ordering algorithm</td>
</tr>
<tr>
<td>Crossover</td>
<td>Barrier crossover strategy</td>
</tr>
<tr>
<td>CrossoverBasis</td>
<td>Crossover initial basis construction strategy</td>
</tr>
<tr>
<td>QCPDual</td>
<td>Compute dual variables for QCP models</td>
</tr>
</tbody>
</table>

**MIP:** These parameters control the operation of the MIP algorithms.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BranchDir</td>
<td>Branch direction preference</td>
</tr>
<tr>
<td>ConcurrentJobs</td>
<td>Enables distributed concurrent solver</td>
</tr>
<tr>
<td>ConcurrentMIP</td>
<td>Enables concurrent MIP solver</td>
</tr>
<tr>
<td>ConcurrentSettings</td>
<td>Comma-separated list of .prm files - used to create concurrent environments</td>
</tr>
<tr>
<td>DistributedMIPJobs</td>
<td>Enables the distributed MIP solver</td>
</tr>
<tr>
<td>Heuristics</td>
<td>Turn MIP heuristics up or down</td>
</tr>
<tr>
<td>ImproveStartGap</td>
<td>Trigger solution improvement</td>
</tr>
<tr>
<td>ImproveStartNodes</td>
<td>Trigger solution improvement</td>
</tr>
<tr>
<td>ImproveStartTime</td>
<td>Trigger solution improvement</td>
</tr>
<tr>
<td>MinRelNodes</td>
<td>Minimum relaxation heuristic control</td>
</tr>
<tr>
<td>MIPFocus</td>
<td>Set the focus of the MIP solver</td>
</tr>
<tr>
<td>MIQCPMethod</td>
<td>Method used to solve MIQCP models</td>
</tr>
<tr>
<td>NodefileDir</td>
<td>Directory for MIP node files</td>
</tr>
<tr>
<td>NodefileStart</td>
<td>Memory threshold for writing MIP tree nodes to disk</td>
</tr>
<tr>
<td>NodeMethod</td>
<td>Method used to solve MIP node relaxations</td>
</tr>
<tr>
<td>PumpPasses</td>
<td>Feasibility pump heuristic control</td>
</tr>
<tr>
<td>RINS</td>
<td>RINS heuristic</td>
</tr>
<tr>
<td>SolutionNumber</td>
<td>Sub-optimal MIP solution retrieval</td>
</tr>
<tr>
<td>SubMIPNodes</td>
<td>Nodes explored by sub-MIP heuristics</td>
</tr>
<tr>
<td>Symmetry</td>
<td>MIP symmetry detection</td>
</tr>
<tr>
<td>VarBranch</td>
<td>Branch variable selection strategy</td>
</tr>
<tr>
<td>ZeroObjNodes</td>
<td>Zero objective heuristic control</td>
</tr>
</tbody>
</table>
**Tuning:** These parameters control the operation of the parameter tuning tool.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TuneJobs</td>
<td>Enables distributed tuning</td>
</tr>
<tr>
<td>TuneOutput</td>
<td>Tuning output level</td>
</tr>
<tr>
<td>TuneResults</td>
<td>Number of improved parameter sets returned</td>
</tr>
<tr>
<td>TuneTimeLimit</td>
<td>Time limit for tuning</td>
</tr>
<tr>
<td>TuneTrials</td>
<td>Perform multiple runs on each parameter set to limit the effect of random noise</td>
</tr>
</tbody>
</table>

**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).

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cuts</td>
<td>Global cut generation control</td>
</tr>
<tr>
<td>CliqueCuts</td>
<td>Clique cut generation</td>
</tr>
<tr>
<td>CoverCuts</td>
<td>Cover cut generation</td>
</tr>
<tr>
<td>FlowCoverCuts</td>
<td>Flow cover cut generation</td>
</tr>
<tr>
<td>FlowPathCuts</td>
<td>Flow path cut generation</td>
</tr>
<tr>
<td>GUBCoverCuts</td>
<td>GUB cover cut generation</td>
</tr>
<tr>
<td>ImpliedCuts</td>
<td>Implied bound cut generation</td>
</tr>
<tr>
<td>MIPSepCuts</td>
<td>MIP separation cut generation</td>
</tr>
<tr>
<td>MIRCuts</td>
<td>MIR cut generation</td>
</tr>
<tr>
<td>ModKCuts</td>
<td>Mod-k cut generation</td>
</tr>
<tr>
<td>NetworkCuts</td>
<td>Network cut generation</td>
</tr>
<tr>
<td>SubMIPCuts</td>
<td>Sub-MIP cut generation</td>
</tr>
<tr>
<td>ZeroHalfCuts</td>
<td>Zero-half cut generation</td>
</tr>
<tr>
<td>CutAggPasses</td>
<td>Constraint aggregation passes performed during cut generation</td>
</tr>
<tr>
<td>CutPasses</td>
<td>Root cutting plane pass limit</td>
</tr>
<tr>
<td>GomoryPasses</td>
<td>Root Gomory cut pass limit</td>
</tr>
</tbody>
</table>

**Distributed algorithms:** Parameters that are used to control our distributed parallel algorithms (distributed MIP, distributed concurrent, and distributed tuning).

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkerPassword</td>
<td>Password for distributed workers</td>
</tr>
<tr>
<td>WorkerPool</td>
<td>List of available distributed workers</td>
</tr>
</tbody>
</table>
### Other: Other parameters.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>AggFill</td>
<td>Allowed fill during presolve aggregation</td>
</tr>
<tr>
<td>Aggregate</td>
<td>Presolve aggregation control</td>
</tr>
<tr>
<td>DisplayInterval</td>
<td>Frequency at which log lines are printed</td>
</tr>
<tr>
<td>DualReductions</td>
<td>Disables dual reductions in presolve</td>
</tr>
<tr>
<td>FeasRelaxBigM</td>
<td>Big-M value for feasibility relaxations</td>
</tr>
<tr>
<td>IISMethod</td>
<td>IIS method</td>
</tr>
<tr>
<td>InputFile</td>
<td>File to be read before optimization commences</td>
</tr>
<tr>
<td>LazyConstraints</td>
<td>Programs that add lazy constraints must set this parameter</td>
</tr>
<tr>
<td>LogFile</td>
<td>Log file name</td>
</tr>
<tr>
<td>LogToConsole</td>
<td>Console logging</td>
</tr>
<tr>
<td>Method</td>
<td>Algorithm used to solve continuous models</td>
</tr>
<tr>
<td>NumericFocus</td>
<td>Set the numerical focus</td>
</tr>
<tr>
<td>OutputFlag</td>
<td>Solver output control</td>
</tr>
<tr>
<td>PreCrush</td>
<td>Allows presolve to translate constraints on the original model to equivalent constraints on the presolved model</td>
</tr>
<tr>
<td>PreDepRow</td>
<td>Presolve dependent row reduction</td>
</tr>
<tr>
<td>PreDual</td>
<td>Presolve dualization</td>
</tr>
<tr>
<td>PrePasses</td>
<td>Presolve pass limit</td>
</tr>
<tr>
<td>PreQLinearize</td>
<td>Presolve Q matrix linearization</td>
</tr>
<tr>
<td>Presolve</td>
<td>Presolve level</td>
</tr>
<tr>
<td>PreSOS1BigM</td>
<td>Controls SOS1 conversion to binary form</td>
</tr>
<tr>
<td>PreSOS2BigM</td>
<td>Controls SOS2 conversion to binary form</td>
</tr>
<tr>
<td>PreSparsify</td>
<td>Presolve sparsify reduction</td>
</tr>
<tr>
<td>ResultFile</td>
<td>Result file written upon completion of optimization</td>
</tr>
<tr>
<td>Seed</td>
<td>Modify the random number seed</td>
</tr>
<tr>
<td>Threads</td>
<td>Number of parallel threads to use</td>
</tr>
</tbody>
</table>
10.1 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, so that you can get an 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 same optimal basis each time you run your program. 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. 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, and applying more aggressive scaling (2) can sometimes improve the numerical properties of the scaled model. 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, 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 to solve the model. The default is to use all cores in the machine. 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. Another common termination choice for MIP models is to set the MIPGap parameter. The MIPGap 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. 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.

**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).

**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, but 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 Cuts parameter sets 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. 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 Resolve 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 whether presolve performs constraint aggregation. 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 nonzero values in the constraint matrix.

**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.

### 10.2 Parameter Descriptions

**AggFill**

<table>
<thead>
<tr>
<th>Type: int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value: -1</td>
</tr>
<tr>
<td>Minimum value: -1</td>
</tr>
<tr>
<td>Maximum value: MAXINT</td>
</tr>
</tbody>
</table>

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.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**Aggregate**

<table>
<thead>
<tr>
<th>Type: int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value: 1</td>
</tr>
<tr>
<td>Minimum value: 0</td>
</tr>
<tr>
<td>Maximum value: 1</td>
</tr>
</tbody>
</table>

Enables or disables aggregation in presolve. 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**

<table>
<thead>
<tr>
<th>Type: double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value: 1e-8</td>
</tr>
<tr>
<td>Minimum value: 0.0</td>
</tr>
<tr>
<td>Maximum value: 1.0</td>
</tr>
</tbody>
</table>

Barrier convergence tolerance
The barrier solver terminates when the relative difference between the primal and dual objective values is less than the specified tolerance (with a \texttt{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.

\textbf{Note:} Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

\begin{itemize}
  \item \textbf{BarCorrectors}
    \begin{itemize}
      \item \textbf{Type:} int
      \item \textbf{Default value:} -1
      \item \textbf{Minimum value:} -1
      \item \textbf{Maximum value:} MAXINT
    \end{itemize}

    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.

    \textbf{Note:} Barrier only

    For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

  \item \textbf{BarHomogeneous}
    \begin{itemize}
      \item \textbf{Type:} int
      \item \textbf{Default value:} -1
      \item \textbf{Minimum value:} -1
      \item \textbf{Maximum value:} 1
    \end{itemize}

    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.

    \textbf{Note:} Barrier only

    For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

  \item \textbf{BarOrder}
    \begin{itemize}
      \item \textbf{Type:} int
      \item \textbf{Default value:} -1
      \item \textbf{Minimum value:} -1
      \item \textbf{Maximum value:} 1
    \end{itemize}

    For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.
\end{itemize}
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

<table>
<thead>
<tr>
<th>Type:</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>1e-6</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0.0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>1.0</td>
</tr>
</tbody>
</table>

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.

**BarIterLimit**

Barrier iteration limit

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>MAXINT</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>

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.

**BranchDir**

Preferred branch direction

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>0</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>1</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>2</td>
</tr>
</tbody>
</table>

Clique cut generation

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.

### ConcurrentJobs

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>0</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>

Distributed concurrent optimizer job count

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 WorkerPool parameter to provide a list of available distributed 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 Algorithm discussion for additional details.

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

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.

**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 set of .prm files

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
For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**CoverCuts**

- **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**

- **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:

<table>
<thead>
<tr>
<th>Parameter value</th>
<th>First push</th>
<th>Second push</th>
<th>Cleanup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dual</td>
<td>Primal</td>
<td>Primal</td>
</tr>
<tr>
<td>2</td>
<td>Dual</td>
<td>Primal</td>
<td>Dual</td>
</tr>
<tr>
<td>3</td>
<td>Primal</td>
<td>Dual</td>
<td>Primal</td>
</tr>
<tr>
<td>4</td>
<td>Primal</td>
<td>Dual</td>
<td>Dual</td>
</tr>
</tbody>
</table>

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

Type: int
Default value: 0
Minimum value: 0
Maximum value: 1

Crossover basis construction strategy

Determines the initial basis construction strategy for crossover. The default value (0) chooses an initial basis quickly. A value of 1 can take much longer, but often produces a more numerically stable start basis.

Note: Barrier only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

Cutoff

Type: double
Default value: Infinity for minimization, -Infinity for maximization
Minimum value: -Infinity
Maximum value: Infinity

Objective cutoff

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 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.

CutAggPasses

Type: int
Default value: -1
Minimum value: -1
Maximum value: MAXINT

Constraint aggregation passes in cut generation

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.

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.

CutPasses

Type: int
Default value: -1
Minimum value: -1
Maximum value: MAXINT

Cutting plane passes
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.

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.

**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

<table>
<thead>
<tr>
<th>Global cut control</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type:</strong> int</td>
</tr>
<tr>
<td><strong>Default value:</strong> -1</td>
</tr>
<tr>
<td><strong>Minimum value:</strong> -1</td>
</tr>
<tr>
<td><strong>Maximum value:</strong> 3</td>
</tr>
</tbody>
</table>

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. 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.

### Disconnected

<table>
<thead>
<tr>
<th>Disconnected component strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type:</strong> int</td>
</tr>
<tr>
<td><strong>Default value:</strong> -1</td>
</tr>
<tr>
<td><strong>Minimum value:</strong> -1</td>
</tr>
<tr>
<td><strong>Maximum value:</strong> 2</td>
</tr>
</tbody>
</table>

A MIP 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

<table>
<thead>
<tr>
<th>Frequency of log lines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type:</strong> int</td>
</tr>
<tr>
<td><strong>Default value:</strong> 5</td>
</tr>
<tr>
<td><strong>Minimum value:</strong> 1</td>
</tr>
<tr>
<td><strong>Maximum value:</strong> MAXINT</td>
</tr>
</tbody>
</table>

Determines the frequency at which log lines are printed (in seconds).
For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**DistributedMIPJobs**

Type: `int`

Default value: 0

Minimum value: 0

Maximum value: `MAXINT`

Distributed MIP job count

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 `WorkerPool` parameter to provide the list of available machines.

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 Algorithm discussion for additional details.

**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**

Type: `int`

Default value: 1

Minimum value: 0

Maximum value: 1

Controls dual reductions

Determines whether dual reductions are performed in presolve. 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**

Type: `double`

Default value: `1e-6`

Minimum value: `1e-9`

Maximum value: `1e-2`

Primal feasibility tolerance

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 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.

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.
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**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>2</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Type:</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>0.05</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

**IISMethod**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>1</td>
</tr>
</tbody>
</table>

Chooses the IIS method to use. Method 0 is often faster, while method 1 can produce a smaller IIS. 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**

**Type:** int  
**Default value:** -1  
**Minimum value:** -1  
**Maximum value:** 2

Implied bound cut generation 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**

**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**

**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.

**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:** LP only

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 file), a simplex basis (a .bas file), or a set of parameter settings (a .prm file) from the Gurobi command line.

**Note:** Command-line only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.
**IntFeasTol**

Type: double

Integer feasibility tolerance

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**

Type: double

Simplex iteration limit

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.

**LazyConstraints**

Type: int

Programs that use lazy constraints must set this parameter

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 theLazy 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.

**LogFile**

Type: string

Name for Gurobi log file

Default value: ""

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.
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**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>1</td>
</tr>
</tbody>
</table>

Enables or disables console logging. 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.

**MarkowitzTol**

<table>
<thead>
<tr>
<th>Type:</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>0.0078125</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>1e-4</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>0.999</td>
</tr>
</tbody>
</table>

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.

**Method**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>4</td>
</tr>
</tbody>
</table>

Algorithm used to solve continuous models or the root node of a MIP model. Options are: -1=automatic, 0=primal simplex, 1=dual simplex, 2=barrier, 3=concurrent, 4=deterministic concurrent.

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 node. Only the simplex and barrier algorithms are available for continuous QP models. Only primal and dual simplex are available for solving the root of an MIQP model. Only barrier is available for continuous QCP models.

Concurrent optimizers run multiple solvers on multiple threads simultaneously, and choose the one that finishes first. Deterministic concurrent (Method=4) gives the exact same result each time, while 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.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**MinRelNodes**

- **Type:** int
- **Default value:** -1
- **Minimum value:** -1
- **Maximum value:** MAXINT

Minimum relaxation heuristic

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.

**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**

- **Type:** int
- **Default value:** 0
- **Minimum value:** 0
- **Maximum value:** 3

MIP solver focus

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 relative gap between the lower and upper objective bound is less than MIPGap times the upper 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.

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 absolute 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.

MISepCuts

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:** MIQCP only

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**MIRCuts**

- **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.

**ModKCuts**

- **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.

**NetworkCuts**

- **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.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.
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.

**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 GBytes) 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: 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**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>2</td>
</tr>
</tbody>
</table>

Method used to solve MIP node relaxations

Algorithm used for MIP node relaxations (0=primal simplex, 1=dual simplex, 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.

**NormAdjust**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>3</td>
</tr>
</tbody>
</table>

Choose simplex pricing norm.

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**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>0</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>3</td>
</tr>
</tbody>
</table>

Numerical focus

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.
**ObjScale**

Objective scaling

**Type:** double  
**Default value:** 0.0  
**Minimum value:** -1  
**Maximum value:** Infinity

Divides the model objective by the specified value to avoid numerical errors that may result from very large 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 \( \text{ObjScale} = -0.5 \) would scale by the square root of the largest objective coefficient).

Objective scaling can be useful when the objective contains extremely large values, but it can also lead to large dual violations, so it should be used sparingly.

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 \( \text{OptimalityTol} \) 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.

**PerturbValue**

Simplex perturbation

**Type:** double  
**Default value:** 0.0002  
**Minimum value:** 0  
**Maximum value:** 0.01

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.
**PreCrush**

Controls presolve reductions that affect user cuts

Type: int  
Default value: 0  
Minimum value: 0  
Maximum value: 1

Allows presolve to translate constraints on the original model to equivalent constraints on the presolved model. You must turn this parameter on when you are using callbacks to add your own cuts.

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:** LP only

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.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**PreQLinearize**

Type: int

Default value: -1

Minimum value: -1

Maximum value: 1

Controls presolve Q matrix linearization. Option 1 attempts to linearize quadratic constraints or a quadratic objective, potentially transforming an MIQP or MIQCP into an MILP. Option 0 shuts off the transformation. The default setting (-1) choose automatically. The automatic setting works well, but there are cases where forcing Q linearization can be beneficial.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**Presolve**

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**

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.

**PreSOS2BigM**

Threshold for SOS2-to-binary reformulation

Type: double  
Default value: 0  
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 0 disables the reformulation. You can set the parameter to -1 to choose an automatic approach, 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.

**PreSparsify**

Controls the presolve sparsify reduction

Type: int  
Default value: -1  
Minimum value: -1  
Maximum value: 1

Controls the presolve sparsify reduction. This reduction can sometimes significantly reduce the number of nonzero values in the presolved model. Value 0 shuts off the reduction, while value 1 forces it on. 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.

**PSDTo1**

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:** QP/QCP/MIQP/MIQCP only
For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**PumpPasses**

Type: int
Default value: 0
Minimum value: 0
Maximum value: MAXINT

Passes of the feasibility pump heuristic

Number of passes of the feasibility pump 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.

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**

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**

Type: int
Default value: -1
Minimum value: -1
Maximum value: 1

Controls quad precision in simplex

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.

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. Possible suffixes are .sol (the solution vector), .bas (the simplex basis), or .mst (the solution vector on the integer variables). 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.

**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:** 0
- **Maximum value:** 2

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 more aggressive scaling option (ScaleFlag=2) can sometimes improve performance for particularly numerically difficult models.

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.
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

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>2</td>
</tr>
</tbody>
</table>

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 root node 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

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>3</td>
</tr>
</tbody>
</table>

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.
### SolutionLimit

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>MAXINT</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>

MIP solution limit

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).

**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

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>0</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>

Select a sub-optimal MIP solution

When querying attribute \( X_n \) 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.

**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

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>2</td>
</tr>
</tbody>
</table>

Sub-MIP cut generation

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

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>500</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>

Nodes explored in sub-MIP heuristics
Limits the number of nodes explored by the RINS heuristic. Exploring more nodes can produce better solutions, but it generally takes longer.

**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**

**MIP symmetric detection**

- **Type:** int
- **Default value:** -1
- **Minimum value:** -1
- **Maximum value:** 2

Controls MIP symmetry detection. A value of -1 corresponds to an automatic setting. Other options are off (0), conservative (1), or aggressive (2).

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.

**Threads**

**Thread count**

- **Type:** int
- **Default value:** 0
- **Minimum value:** 0
- **Maximum value:** NProc

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 all of the cores in the machine, but it may choose to use fewer.

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 uncontended.

Another situation where reducing the thread count can be helpful is when memory is tight. Each thread can consume a significant amount of memory.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.
**TimeLimit**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time limit</td>
<td>double</td>
<td>Infinity</td>
<td>0</td>
<td>Infinity</td>
</tr>
</tbody>
</table>

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).

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**TuneJobs**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed tuning job count</td>
<td>int</td>
<td>0</td>
<td>0</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>

Enables distributed parallel tuning, which can significantly increase the performance of the tuning tool. A value of \( n \) causes the tuning tool to distribute tuning work among \( n \) parallel jobs. These jobs are distributed among a set of machines. Use the WorkerPool parameter to provide a list of available distributed worker machines.

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.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**TuneOutput**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuning output level</td>
<td>int</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numer of improved parameter sets returned</td>
<td>int</td>
<td>-1</td>
<td>1</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>
The tuning tool often finds multiple parameter sets that produce better results than the baseline settings. This parameter controls how many of these sets should be retained when tuning is complete. The default value retains the best results that were found for each count of changed parameters. In other words, it retains the best result for one changed parameter, for two changed parameter, etc. Results that aren’t on the efficient frontier are discarded.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

**TuneTimeLimit**

<table>
<thead>
<tr>
<th>Type:</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>Infinity</td>
</tr>
</tbody>
</table>

Tuning tool time limit

Limits total tuning runtime (in seconds). The default setting (-1) 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**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>3</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>

Perform multiple runs on each parameter set to limit the effect of random noise

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.

**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.

**VarBranch**

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>-1</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>3</td>
</tr>
</tbody>
</table>

Branch variable selection strategy

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.

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 workers listed in the WorkerPool parameter.

For examples of how to query or modify parameter values from our different APIs, refer to our Parameter Examples.

WorkerPool

Pool of machines to use 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 comma-separated list of machines that can be used as workers. These machines must be running Gurobi Remote Services. You can refer to these workers using their names or their IP addresses. You should specify the access password, if there is one, in the WorkerPassword parameter.

To give an example, if you have two machines named server1.mydomain.com and server2.mydomain.com, with IP addresses 192.168.1.100 and 192.168.1.101, you could set the WorkerPool to "server1.mydomain.com,server2.mydomain.com" or "192.168.1.100,192.168.1.101".

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

<table>
<thead>
<tr>
<th>Type:</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value:</td>
<td>0</td>
</tr>
<tr>
<td>Minimum value:</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value:</td>
<td>MAXINT</td>
</tr>
</tbody>
</table>

Zero-objective heuristic

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.

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.

10.3 Parameter Examples

While the meanings of the various Gurobi parameters remain constant between the different language API’s, the methods used to query or modify them vary. Refer to the following sections for detailed examples of how to work with parameters from our various API’s:

- 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:
The routine you use to modify a parameter value depends on the type of the parameter. For a double-valued parameter, you would use \texttt{GRBsetdblparam}. Recall that models get their own environments once they are created, so you’ll generally need to get the environment for a model before setting a parameter on that model.

To set the \texttt{TimeLimit} parameter for a model, you’d do:

```c
modelenv = GRBgetenv(model);
...
error = GRBsetdblparam(modelenv, GRB\_DBL\_PAR\_TIMELIMIT, 100.0);
```

If you’d prefer to use a string for the parameter name, you can also do:

```c
error = GRBsetdblparam(modelenv, "TimeLimit", 100.0);
```

The case of the string is ignored, as are underscores. Thus, \texttt{TimeLimit} and \texttt{TIME\_LIMIT} are equivalent.

Use \texttt{GRBgetdblparam} to query the current value of a parameter:

```c
double currentvalue;
error = GRBgetdblparam(modelenv, "TimeLimit", &currentvalue);
```

### C++ Parameter Examples

In the C++ interface, parameters are grouped by datatype into three \texttt{enums}: \texttt{GRB\_DoubleParam}, \texttt{GRB\_IntParam}, and \texttt{GRB\_StringParam}. You refer to a specific parameter by appending the parameter name to the enum name. For example, the \texttt{Threads} parameter is \texttt{GRB\_IntParam\_Threads}.

To modify a parameter, you use \texttt{GRBEnv::set}. Recall that models get their own environments once they are created, so you’ll generally need to get the environment for a model before setting a parameter on that model.

To set the \texttt{TimeLimit} parameter for a model, you’d do:

```cpp
GRBModel *m = ...;
m->getEnv().set(GRB\_DoubleParam\_TimeLimit, 100.0);
```

Use \texttt{GRBEnv::get} to query the current value of a parameter:

```cpp
currentlimit = m.getEnv().get(GRB\_DoubleParam\_TimeLimit);
```

### C# Parameter Examples

In the C# interface, parameters are grouped by datatype into three \texttt{enums}: \texttt{GRB\_DoubleParam}, \texttt{GRB\_IntParam}, and \texttt{GRB\_StringParam}. You would refer to the integer \texttt{Threads} parameter as \texttt{GRB\_IntParam\_Threads}.

To modify a parameter, you use \texttt{GRBEnv.Set}. Recall that models get their own environments once they are created, so you’ll generally need to get the environment for a model before setting a parameter on that model.

To set the \texttt{TimeLimit} parameter for a model, you’d do:
GRBModel m = ...;
m.GetEnv().Set(GRB.DoubleParam.TimeLimit, 100.0);

Use GRBEnv.Get to query the current value of a parameter:
currentlimit = m.GetEnv().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 GRBEnv.set. Recall that models get their own environments once they are created, so you’ll generally need to get the environment for a model before setting a parameter on that model.

To set the TimeLimit parameter for a model, you’d do:

GRBModel m = ...;
m.getEnv().set(GRB.DoubleParam.TimeLimit, 100.0);

Use GRBEnv.get to query the current value of a parameter:
currentlimit = m.getEnv().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:

```python
m.params.timelimit = 100.0
```

...or...

```python
m.params.TIME_LIMIT = 100.0
```

You can also use the `Model.setParam` method:

```python
m.setParam(GRB.Param.TimeLimit, 100.0)
```

If you’d prefer to use a string for the parameter name, you can also do:

```python
m.setParam("TimeLimit", 100.0);
```

To query the current value of a parameter, use:

```python
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:

```r
params <- list(TimeLimit=100)
```

The case of the parameter name is ignored, as are underscores. Thus, you could also do:

```r
params <- list(timeLimit = 100)
```

...or...

```r
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, you use `GRBEnv.Set`. Recall that models get their own environments once they are created, so you’ll generally need to get the environment for a model before setting a parameter on that model.

To set the `TimeLimit` parameter for a model, you’d do:

```vbnet
GRBModel m = ...;
m.GetEnv().Set(GRB.DoubleParam.TimeLimit, 100.0);
```

Use `GRBEnv.Get` to query the current value of a parameter:

```vbnet
currentlimit = m.GetEnv().Get(GRB.DoubleParam.TimeLimit);
```
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 pre-defined 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 \texttt{OPTIMAL} in the following ways from the available Gurobi interfaces:

<table>
<thead>
<tr>
<th>Language</th>
<th>Status Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>\texttt{GRB_OPTIMAL}</td>
</tr>
<tr>
<td>C++</td>
<td>\texttt{GRB_OPTIMAL}</td>
</tr>
<tr>
<td>Java</td>
<td>\texttt{GRB.Status.OPTIMAL}</td>
</tr>
<tr>
<td>.NET</td>
<td>\texttt{GRB.Status.OPTIMAL}</td>
</tr>
<tr>
<td>Python</td>
<td>\texttt{GRB.OPTIMAL}</td>
</tr>
</tbody>
</table>

Possible status codes are as follows:

<table>
<thead>
<tr>
<th>Status code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{LOADED}</td>
<td>1</td>
<td>Model is loaded, but no solution information is available.</td>
</tr>
<tr>
<td>\texttt{OPTIMAL}</td>
<td>2</td>
<td>Model was solved to optimality (subject to tolerances), and an optimal solution is available.</td>
</tr>
<tr>
<td>\texttt{INFEASIBLE}</td>
<td>3</td>
<td>Model was proven to be infeasible.</td>
</tr>
<tr>
<td>\texttt{INF_OR_UNBD}</td>
<td>4</td>
<td>Model was proven to be either infeasible or unbounded. To obtain a more definitive conclusion, set the \texttt{DualReductions} parameter to 0 and reoptimize.</td>
</tr>
<tr>
<td>\texttt{UNBOUNDED}</td>
<td>5</td>
<td>Model was proven to be unbounded. \textit{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.}</td>
</tr>
<tr>
<td>\texttt{CUTOFF}</td>
<td>6</td>
<td>Optimal objective for model was proven to be worse than the value specified in the \texttt{Cutoff} parameter. No solution information is available.</td>
</tr>
<tr>
<td>\texttt{ITERATION_LIMIT}</td>
<td>7</td>
<td>Optimization terminated because the total number of simplex iterations performed exceeded the value specified in the \texttt{IterationLimit} parameter, or because the total number of barrier iterations exceeded the value specified in the \texttt{BarIterLimit} parameter.</td>
</tr>
<tr>
<td>\texttt{NODE_LIMIT}</td>
<td>8</td>
<td>Optimization terminated because the total number of branch-and-cut nodes explored exceeded the value specified in the \texttt{NodeLimit} parameter.</td>
</tr>
<tr>
<td>\texttt{TIME_LIMIT}</td>
<td>9</td>
<td>Optimization terminated because the time expended exceeded the value specified in the \texttt{TimeLimit} parameter.</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SOLUTION_LIMIT</td>
<td>10</td>
<td>Optimization terminated because the number of solutions found reached the value specified in the SolutionLimit parameter.</td>
</tr>
<tr>
<td>INTERRUPTED</td>
<td>11</td>
<td>Optimization was terminated by the user.</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>12</td>
<td>Optimization was terminated due to unrecoverable numerical difficulties.</td>
</tr>
<tr>
<td>SUBOPTIMAL</td>
<td>13</td>
<td>Unable to satisfy optimality tolerances; a sub-optimal solution is available.</td>
</tr>
<tr>
<td>INPROGRESS</td>
<td>14</td>
<td>An asynchronous optimization call was made, but the associated optimization run is not yet complete.</td>
</tr>
</tbody>
</table>
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).

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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Callback constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GRB_CB_SIMPLEX</td>
</tr>
<tr>
<td>C++</td>
<td>GRB_CB_SIMPLEX</td>
</tr>
<tr>
<td>Java</td>
<td>GRB.Callback.SIMPLEX</td>
</tr>
<tr>
<td>.NET</td>
<td>GRB.Callback.SIMPLEX</td>
</tr>
<tr>
<td>Python</td>
<td>GRB.Callback.SIMPLEX</td>
</tr>
</tbody>
</table>

Possible `where` values are:

<table>
<thead>
<tr>
<th>where</th>
<th>Numeric value</th>
<th>Optimizer status</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLING</td>
<td>0</td>
<td>Periodic polling callback</td>
</tr>
<tr>
<td>PRESOLVE</td>
<td>1</td>
<td>Currently performing presolve</td>
</tr>
<tr>
<td>SIMPLEX</td>
<td>2</td>
<td>Currently in simplex</td>
</tr>
<tr>
<td>MIP</td>
<td>3</td>
<td>Currently in MIP</td>
</tr>
<tr>
<td>MIPSOL</td>
<td>4</td>
<td>Found a new MIP incumbent</td>
</tr>
<tr>
<td>MIPNODE</td>
<td>5</td>
<td>Currently exploring a MIP node</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>6</td>
<td>Printing a log message</td>
</tr>
<tr>
<td>BARRIER</td>
<td>7</td>
<td>Currently in barrier</td>
</tr>
</tbody>
</table>

Allowable `what` values depend on the value of the `where` argument. Valid combinations are:

<table>
<thead>
<tr>
<th>what</th>
<th>where</th>
<th>Result type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNTIME</td>
<td>Any except POLLING</td>
<td>double</td>
<td>Elapsed solver runtime (seconds).</td>
</tr>
<tr>
<td>PRE_COLDEL</td>
<td>PRESOLVE</td>
<td>int</td>
<td>The number of columns removed by presolve to this point.</td>
</tr>
<tr>
<td>PRE_ROWDEL</td>
<td>PRESOLVE</td>
<td>int</td>
<td>The number of rows removed by presolve to this point.</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>PRE_SENCHG</td>
<td>PRESOLVE</td>
<td>int The number of constraint senses changed by presolve to this point.</td>
<td></td>
</tr>
<tr>
<td>PRE_BNDCHG</td>
<td>PRESOLVE</td>
<td>int The number of variable bounds changed by presolve to this point.</td>
<td></td>
</tr>
<tr>
<td>PRE_COECHG</td>
<td>PRESOLVE</td>
<td>int The number of coefficients changed by presolve to this point.</td>
<td></td>
</tr>
<tr>
<td>SPX_ITRCNT</td>
<td>SIMPLEX</td>
<td>double Current simplex iteration count.</td>
<td></td>
</tr>
<tr>
<td>SPX_OBJVAL</td>
<td>SIMPLEX</td>
<td>double Current simplex objective value.</td>
<td></td>
</tr>
<tr>
<td>SPX_PRIMINF</td>
<td>SIMPLEX</td>
<td>double Current primal infeasibility.</td>
<td></td>
</tr>
<tr>
<td>SPX_DUALINF</td>
<td>SIMPLEX</td>
<td>double Current dual infeasibility.</td>
<td></td>
</tr>
<tr>
<td>SPX_ISPERT</td>
<td>SIMPLEX</td>
<td>int Is problem current perturbed?</td>
<td></td>
</tr>
<tr>
<td>MIP_OBJBST</td>
<td>MIP</td>
<td>double Current best objective.</td>
<td></td>
</tr>
<tr>
<td>MIP_OBJBND</td>
<td>MIP</td>
<td>double Current best objective bound.</td>
<td></td>
</tr>
<tr>
<td>MIP_NODCNT</td>
<td>MIP</td>
<td>double Current explored node count.</td>
<td></td>
</tr>
<tr>
<td>MIP_SOLCNT</td>
<td>MIP</td>
<td>int Current count of feasible solutions found.</td>
<td></td>
</tr>
<tr>
<td>MIP_CUTCNT</td>
<td>MIP</td>
<td>int Current count of cutting planes applied.</td>
<td></td>
</tr>
<tr>
<td>MIP_NODLFT</td>
<td>MIP</td>
<td>double Current unexplored node count.</td>
<td></td>
</tr>
<tr>
<td>MIP_ITRCNT</td>
<td>MIP</td>
<td>double Current simplex iteration count.</td>
<td></td>
</tr>
<tr>
<td>MIPSOL_SOL</td>
<td>MIPSOL</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>MIPSOL_OBJ</td>
<td>MIPSOL</td>
<td>double Objective value for new solution.</td>
<td></td>
</tr>
<tr>
<td>MIPSOL_OBJBST</td>
<td>MIPSOL</td>
<td>double Current best objective.</td>
<td></td>
</tr>
<tr>
<td>MIPSOL_OBJBND</td>
<td>MIPSOL</td>
<td>double Current best objective bound.</td>
<td></td>
</tr>
<tr>
<td>MIPSOL_NODCNT</td>
<td>MIPSOL</td>
<td>double Current explored node count.</td>
<td></td>
</tr>
<tr>
<td>MIPSOL_SOLCNT</td>
<td>MIPSOL</td>
<td>int Current count of feasible solutions found.</td>
<td></td>
</tr>
<tr>
<td>MIPNODE_STATUS</td>
<td>MIPNODE</td>
<td>int Optimization status of current MIP node (see the Status Code section for further information).</td>
<td></td>
</tr>
<tr>
<td>MIPNODE_OBJBST</td>
<td>MIPNODE</td>
<td>double Current best objective.</td>
<td></td>
</tr>
<tr>
<td>MIPNODE_OBJBND</td>
<td>MIPNODE</td>
<td>double Current best objective bound.</td>
<td></td>
</tr>
<tr>
<td>MIPNODE_NODCNT</td>
<td>MIPNODE</td>
<td>double Current explored node count.</td>
<td></td>
</tr>
<tr>
<td>MIPNODE_SOLCNT</td>
<td>MIPNODE</td>
<td>int Current count of feasible solutions found.</td>
<td></td>
</tr>
</tbody>
</table>
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.

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.
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.

Note that the POLLING callback does not allow any additional information to be retrieved. It is provided in order to allow interactive applications to regain control frequently, so that they can maintain application responsiveness.
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 pre-defined 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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GRB_ERROR_OUT_OF_MEMORY</td>
</tr>
<tr>
<td>C++</td>
<td>GRB_ERROR_OUT_OF_MEMORY</td>
</tr>
<tr>
<td>Java</td>
<td>GRB.Error.OUT_OF_MEMORY</td>
</tr>
<tr>
<td>.NET</td>
<td>GRB.Error.OUT_OF_MEMORY</td>
</tr>
<tr>
<td>Python</td>
<td>GRB.Error.OUT_OF_MEMORY</td>
</tr>
</tbody>
</table>

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.

Possible error codes are:

<table>
<thead>
<tr>
<th>Error code</th>
<th>Error number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT_OF_MEMORY</td>
<td>10001</td>
<td>Available memory was exhausted</td>
</tr>
<tr>
<td>NULL_ARGUMENT</td>
<td>10002</td>
<td>NULL input value provided for a required argument</td>
</tr>
<tr>
<td>INVALID_ARGUMENT</td>
<td>10003</td>
<td>An invalid value was provided for a routine argument</td>
</tr>
<tr>
<td>UNKNOWN_ATTRIBUTE</td>
<td>10004</td>
<td>Tried to query or set an unknown attribute</td>
</tr>
<tr>
<td>DATA_NOT_AVAILABLE</td>
<td>10005</td>
<td>Attempted to query or set an attribute that could not be accessed at that time</td>
</tr>
<tr>
<td>INDEX_OUT_OF_RANGE</td>
<td>10006</td>
<td>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</td>
</tr>
<tr>
<td>UNKNOWN_PARAMETER</td>
<td>10007</td>
<td>Tried to query or set an unknown parameter</td>
</tr>
<tr>
<td>VALUE_OUT_OF_RANGE</td>
<td>10008</td>
<td>Tried to set a parameter to a value that is outside the parameter’s valid range</td>
</tr>
<tr>
<td>NO_LICENSE</td>
<td>10009</td>
<td>Failed to obtain a valid license</td>
</tr>
<tr>
<td>Condition</td>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>SIZE_LIMIT_EXCEEDED</td>
<td>10010</td>
<td>Attempted to solve a model that is larger than the limit for a demo license</td>
</tr>
<tr>
<td>CALLBACK</td>
<td>10011</td>
<td>Problem in callback</td>
</tr>
<tr>
<td>FILE_READ</td>
<td>10012</td>
<td>Failed to read the requested file</td>
</tr>
<tr>
<td>FILE_WRITE</td>
<td>10013</td>
<td>Failed to write the requested file</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>10014</td>
<td>Numerical error during requested operation</td>
</tr>
<tr>
<td>IIS_NOT_INFEASIBLE</td>
<td>10015</td>
<td>Attempted to perform infeasibility analysis on a feasible model</td>
</tr>
<tr>
<td>NOT_FOR_MIP</td>
<td>10016</td>
<td>Requested operation not valid for a MIP model</td>
</tr>
<tr>
<td>OPTIMIZATION_IN_PROGRESS</td>
<td>10017</td>
<td>Tried to query or modify a model while optimization was in progress</td>
</tr>
<tr>
<td>DUPLICATES</td>
<td>10018</td>
<td>Constraint, variable, or SOS contained duplicated indices</td>
</tr>
<tr>
<td>NODEFILE</td>
<td>10019</td>
<td>Error in reading or writing a node file during MIP optimization</td>
</tr>
<tr>
<td>Q_NOT_PSD</td>
<td>10020</td>
<td>Q matrix in QP model is not positive semi-definite</td>
</tr>
<tr>
<td>QCP_EQUALITY_CONSTRAINT</td>
<td>10021</td>
<td>QCP equality constraint specified (only inequalities are supported)</td>
</tr>
<tr>
<td>NETWORK</td>
<td>10022</td>
<td>Problem communicating with the Gurobi Compute Server</td>
</tr>
<tr>
<td>JOB_REJECTED</td>
<td>10023</td>
<td>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)</td>
</tr>
<tr>
<td>NOT_SUPPORTED</td>
<td>10024</td>
<td>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)</td>
</tr>
<tr>
<td>EXCEED_2B_NONZEROS</td>
<td>10025</td>
<td>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.</td>
</tr>
<tr>
<td>INVALID_PIECEWISE_OBJ</td>
<td>10026</td>
<td>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.</td>
</tr>
<tr>
<td>NOT_IN_MODEL</td>
<td>20001</td>
<td>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</td>
</tr>
<tr>
<td>FAILED_TO_CREATE_MODEL</td>
<td>20002</td>
<td>Failed to create the requested model</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>20003</td>
<td>Internal Gurobi error</td>
</tr>
</tbody>
</table>
The Gurobi optimizer works with a variety of file formats. The MPS, REW, LP, RLP, and ILP 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 the MIP model to start with a known 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 format holds a solution vector. It can be written once the model has been optimized. 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.

14.1 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 white space 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.

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
```
Note that an N in the type field indicates that the row is a free row. The first free row is traditionally used as the objective function, while additional free rows are ignored.

**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. 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 $X_{50}$ has a right-hand side value of 310, and $X_{51}$ 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.

**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:

- **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

Consider the following example:

```
BOUNDS
  UP BND X50 80.
  LO BND X51 20.
  FX BND X52 30.
```

In this BOUNDS section, variable $X_{50}$ gets a upper bound of 80 (lower bound is unchanged at 0, $X_{51}$ gets a lower bound of 20 (infinite upper bound is unchanged), and $X_{52}$ 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 \((10X01^2 + 2X01 \times X02 + 2X02 \times X01 + 2X02^2)/2\) (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 is the optional **QCMATRIX** section, which contains quadratic terms for quadratic constraints. Each quadratic constraint gets its own header. The header includes the QCMATRIX label (starting in column 1), followed by the name of the quadratic constraint (starting in column 12). The lines that follow each represent a single non-zero in the corresponding quadratic constraint. The names of the two participating variables appear first (in columns 5 and 15), followed by the coefficient value (in column 26). Here’s an example of a quadratic expression:

```
QCMATRIX qc0
    X01   X01   10.0
    X01   X02   1.0
    X02   X02   5.0
```

Note that the linear terms, the sense, and the right-hand side value for the quadratic constraint appear in the **COLUMNS**, **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
```
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 or more quadratic terms. Each term is described on one line, which indicates 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 three terms would indicate that quadratic constraint **QC0** contains terms 10\(X01^2\), 4\(X01\times X02\), and 2\(X02^2\). Linear terms for constraint **QC0** would appear in the **COLUMNS** section. Note that a **QCMATRIX** block must contain a symmetric matrix, so for example an \(X01\times X02\) term must be accompanied by a matching \(X02\times X01\) term.

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.

14.2 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. Specifically, column number \(j\) gets name **C000j**, where the number of zeros is chosen to make the total name length at least 8 characters. Similarly, row number \(i\) gets name **R000i**.

14.3 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 white space 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. The name should be no longer than 255 characters, and to avoid confusing the LP parser, it should not begin with a number, or with any of the characters +, -, *, <, >, =, or :.

Note that white space is 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 headers, on its own line: minimize, maximize, minimum, maximum, min, or max. Capitalization in the header is ignored. The header is then followed by a 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., \( 2x^2 \)) or product terms (e.g., \( 3x * y \)) are accepted. Coefficients on the quadratic terms are optional.

For each variable with a piecewise-linear objective, the objective section will include a \_\_pwl(x) term, where \( x \) is the name of the variable. You should view these as comments; they are ignored by the LP reader. 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:

\[
\text{Minimize} \\
\text{obj: } 3.1x + 4.5y + 10z + \left[ x^2 + 2xy + 3y^2 \right] / 2
\]

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:

\[
c0: 2.5x + 2.3y + 5.3z \leq 8.1
\]

The following is a valid quadratic constraint:

\[
qc0: 3.1x + 4.5y + 10z + \left[ x^2 + 2xy + 3y^2 \right] \leq 10
\]

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, 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

\[
c0: 2.5x + 2.3y + 5.3z \leq 8.1
\]

Lazy constraints are handled differently from other constraints by the MIP solver. A lazy constraint only becomes active when the MIP solver finds a candidate solution that violates the constraint.

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:

```
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.

**End statement**

The last line in an LP format file should be an `End` statement.

### 14.4 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 \(c_i\). 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 \(B7(2)\).

### 14.5 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.

### 14.6 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.

### 14.7 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 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:

```plaintext
# 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.

### 14.8 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 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 **ENDDATA** 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. If the first field is **LL**, **UL**, or **BS**, the variable named 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

---

537
in the second field is basic, while the variable named in the third field is non-basic at its lower or upper bound, respectively.

The following is a simple example:

```plaintext
NAME example.bas
XL x1 c1
XU x2 c2
BS c3
UL x3
LL x4
```

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.

**14.9 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:

```plaintext
# Solution file
x  1.0
y  0.5
z  0.2
```

**14.10 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:

```plaintext
# 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.
The Gurobi optimizer produces a log that allows you to track the progress of the optimization. By default, the log is put to both the screen and to a file. Screen output can be controlled using the `OutputFlag` parameter, and file output can be controlled using the `LogFile` parameter.

The format of the log depends on the algorithm that is used to solve the model (simplex, barrier, sifting, or branch-and-cut). We now describe the contents of the log for each algorithm.

15.1 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 df1001:

Presolve removed 2381 rows and 3347 columns
Presolve time: 0.12 sec.
Presolved: 3690 Rows, 8883 Columns, 31075 Nonzeros

The example output shows that presolve was able to remove 2381 rows and 3347 columns, and it required 0.12 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:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Objective</th>
<th>Primal Inf.</th>
<th>Dual Inf.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.7748600e+04</td>
<td>6.627132e+03</td>
<td>0.000000e+00</td>
<td>0s</td>
</tr>
<tr>
<td>9643</td>
<td>1.1574611e+07</td>
<td>1.418653e+03</td>
<td>0.000000e+00</td>
<td>5s</td>
</tr>
<tr>
<td>14440</td>
<td>1.1607748e+07</td>
<td>4.793500e+00</td>
<td>0.000000e+00</td>
<td>10s</td>
</tr>
<tr>
<td>15213</td>
<td>1.1266396e+07</td>
<td>0.000000e+00</td>
<td>0.000000e+00</td>
<td>11s</td>
</tr>
</tbody>
</table>

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 constraints), and the time taken (in seconds) since the start of the optimization.
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 an initial 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 15213 iterations and 10.86 seconds
Optimal objective 1.126639605e+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

15.2 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 df1001:

Presolve removed 2381 rows and 3347 columns
Presolve time: 0.12 sec.
Presolved: 3690 Rows, 8883 Columns, 31075 Nonzeros

The example output shows that presolve was able to remove 2381 rows and 3347 columns, and
it required 0.12 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 reduced 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:**
- Dense cols : 10
- Free vars : 3
- $AA'\text{ NZ }$ : 9.353e+04
- Factor NZ : 1.139e+06 (roughly 14 MBytes of memory)
- Factor Ops : 7.388e+08 (roughly 2 seconds per iteration)

The first line indicates how many columns from the constraint matrix were treated as dense. The second line 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 sometimes useful to know that they are present. Note that these lines are only printed when the model contains dense columns or free variables.

The next 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 final 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.

**Progress Section**

The third section of the Gurobi barrier output provides information on the progress of the barrier method:

<table>
<thead>
<tr>
<th>Iter</th>
<th>Primal</th>
<th>Dual</th>
<th>Residual</th>
<th>Primal</th>
<th>Dual</th>
<th>Compl</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.11502515e+13</td>
<td>-3.03102251e+08</td>
<td>7.65e+05</td>
<td>9.29e+07</td>
<td>2.68e+09</td>
<td>2s</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4.40523949e+12</td>
<td>-8.22101865e+09</td>
<td>3.10e+05</td>
<td>4.82e+07</td>
<td>1.15e+09</td>
<td>3s</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.18016996e+12</td>
<td>-2.25095257e+10</td>
<td>7.39e+04</td>
<td>1.15e+07</td>
<td>3.37e+08</td>
<td>4s</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.24969338e+11</td>
<td>-2.09167762e+10</td>
<td>1.01e+04</td>
<td>2.16e+06</td>
<td>5.51e+07</td>
<td>5s</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.63336765e+10</td>
<td>-1.44308755e+10</td>
<td>8.13e+02</td>
<td>4.30e+05</td>
<td>9.09e+06</td>
<td>6s</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.25266057e+10</td>
<td>-4.06364070e+09</td>
<td>1.52e+02</td>
<td>8.13e+04</td>
<td>2.21e+06</td>
<td>7s</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.53128732e+09</td>
<td>-1.27023188e+09</td>
<td>9.52e+00</td>
<td>1.61e+04</td>
<td>3.23e+05</td>
<td>9s</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5.70973983e+08</td>
<td>-8.11694302e+08</td>
<td>2.10e+00</td>
<td>5.99e+03</td>
<td>1.53e+05</td>
<td>10s</td>
<td></td>
</tr>
</tbody>
</table>
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.

**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...

```
1592 DPushes remaining with DInf 0.0000000e+00 2s
0 DPushes remaining with DInf 2.8167333e-06 2s
180 PPushes remaining with PInf 0.0000000e+00 2s
0 PPushes remaining with PInf 0.0000000e+00 2s

Push phase complete: Pinf 0.0000000e+00, Dinf 2.8167333e-06 2s
```

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
1776 1.1266396e+07 0.0000000e+00 0.0000000e+00 2s
```

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 7212 iterations and 48.38 seconds**  
**Optimal objective 1.126639605e+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**

### 15.3 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:

<table>
<thead>
<tr>
<th>Iter</th>
<th>Pivots</th>
<th>Primal Obj</th>
<th>Dual Obj</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>infinity</td>
<td>2.0000000e+01</td>
<td>11s</td>
</tr>
<tr>
<td>1</td>
<td>4662</td>
<td>1.5220652e+03</td>
<td>2.7034420e+02</td>
<td>12s</td>
</tr>
<tr>
<td>2</td>
<td>8917</td>
<td>1.3127217e+03</td>
<td>4.6530259e+02</td>
<td>13s</td>
</tr>
<tr>
<td>3</td>
<td>16601</td>
<td>1.1651147e+03</td>
<td>6.4767742e+02</td>
<td>17s</td>
</tr>
<tr>
<td>4</td>
<td>30060</td>
<td>1.0881514e+03</td>
<td>7.8842688e+02</td>
<td>29s</td>
</tr>
<tr>
<td>5</td>
<td>45169</td>
<td>1.0618879e+03</td>
<td>8.8656855e+02</td>
<td>46s</td>
</tr>
<tr>
<td>6</td>
<td>59566</td>
<td>1.0549766e+03</td>
<td>9.5404159e+02</td>
<td>64s</td>
</tr>
<tr>
<td>7</td>
<td>73614</td>
<td>1.0540577e+03</td>
<td>1.0172213e+03</td>
<td>82s</td>
</tr>
</tbody>
</table>
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.

15.4 MIP Logging

The MIP log can be divided into three sections: the presolve section, the simplex 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: 0.00s
Presolved: 12 Rows, 148 Columns, 1615 Nonzeros

In this example, presolve was able to remove 3 columns. The final line shows the size of the model that is passed to the branch-and-cut algorithm.

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 these lines:

Found heuristic solution: objective 93644.999
Found heuristic solution: objective 87658.484
Found heuristic solution: objective 80811.127

These indicate that the Gurobi heuristics found three integer feasible solutions 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, 43 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 relaxation log...

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Objective</th>
<th>Primal Inf.</th>
<th>Dual Inf.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8370</td>
<td>5.6894789e+02</td>
<td>3.032449e+05</td>
<td>0.000000e+00</td>
<td>5s</td>
</tr>
</tbody>
</table>

544
Root relaxation: objective 5.762316e+02, 34230 iterations, 28.47 seconds

To be more precise, this more detailed log is triggered whenever the root relaxation requires more than the DisplayInterval parameter value (5 seconds by default).

The next section provides progress information on the branch-and-cut tree search:

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Current Node</th>
<th>Objective Bounds</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expl</td>
<td>Unexpl</td>
<td>Obj</td>
<td>Depth</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>38893.904</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>45476.147</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>38903.750</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>38926.214</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>38950.968</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>2</td>
<td>38952.279</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>2</td>
<td>40005.054</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>38952.279</td>
<td>0</td>
</tr>
<tr>
<td>96386</td>
<td>22115</td>
<td>cutoff</td>
<td>37</td>
</tr>
<tr>
<td>203266</td>
<td>12649</td>
<td>cutoff</td>
<td>30</td>
</tr>
</tbody>
</table>

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: 6
- Cover: 5
- MIR: 8

Explored 226525 nodes (854805 simplex iterations) in 11.15 seconds
Thread count was 2 (of 2 available processors)

Optimal solution found (tolerance 1.00e-04)
Best objective 4.0005054142e+04, best bound 4.0001112908e+04, gap 0.0099%

In this example, the Gurobi solver required just over 11 seconds to solve the model to optimality, and it used two processors to do so (the processor count can be limited with the Threads parameter). The gap between the best feasible solution objective and the best bound is just under 0.01%, which produces an Optimal termination status, since the achieved gap is smaller than the default MIPGap parameter value.

### 15.5 Distributed MIP Logging

Logging for distributed MIP is very similar to the standard MIP logging. The main differences are in the progress section. The header for the standard MIP logging looks like this:

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Current Node</th>
<th>Objective Bounds</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expl</td>
<td>Unexpl</td>
<td>Obj Depth IntInf</td>
<td>Incumbent</td>
</tr>
</tbody>
</table>

By contrast, the distributed MIP header looks like this:

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Utilization</th>
<th>Objective Bounds</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expl</td>
<td>Unexpl</td>
<td>Active Sync Comm</td>
<td>Incumbent</td>
</tr>
</tbody>
</table>

You’ll note that columns three through five show different information. In the distributed MIP log, these columns give information about the utilization of the distributed workers, expressed as percentages. The first of these columns shows the fraction of the preceding time period (the time since the previous progress log line) that the workers spent actively processing MIP nodes. The second column shows the time the workers spent waiting for other workers to complete tasks assigned to them. The final column shows the time spent communicating data between machines. The numbers sum to 100%.

Here is an example of a distributed MIP progress log:

---

546
One thing you may find in the progress section is that node counts may not increase monotonically. In this example, a solution found at node 261 is reported before a solution found at node 0. 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:
Active: 13.73s (75%)
Sync: 1.16s (6%)
Comm: 3.45s (19%)

This is an aggregated view of the utilization data that is displayed in the progress log lines. In this example, the workers spent 75% of runtime actively working on MIP nodes, 6% waiting to
synchronize with other workers, and 19% communicating data between machines.
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 administer Gurobi Remote Services and Gurobi Compute Server. The syntax for this usage is:

```
gurobi_cl [--command=]*
```

A list of supported commands can be found 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.

16.1 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 [parameter=value]* modelfile
```
**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.

**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.

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), 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, a `.mst` file populates the `Start` attribute, and a `.ord` file populates the `BranchPriority` attribute.

Again, you should consult the File Formats section for more information on supported file formats.

---

**16.2 Gurobi Remote Services and Compute Server Administration**

The command-line tool can also be used to administer Gurobi Remote Services and Gurobi Compute Server. The format of an administrative command is simply:

```
gurobi_cl [--command]*
```

Available administrative commands are:

--status: Obtain a list of running and queued jobs.

--killjob=: Kill a job. The argument identifies the job to kill. You specify a job by giving the client hostname, followed by a comma, followed by the process ID (PID) of the job. You typically obtain this information from the output of `gurobi_cl --status`.

--joblimit=: Change the server job limit. The argument gives the new limit. Note that this command is useful for taking a Compute Server off-line: setting the job limit to zero allows currently running jobs to finish, but prevents new ones from starting.

--adminpassword: Change the administrator password.

Administrative commands can be run from any machine on the same network as the server. All except `--status` prompt you for the administrator password. For security reasons, if no initial administrator password is specified (via the `grb_rs.cnf` file), server administration is disabled.

One additional command-line argument that you may need in conjunction with these commands is `--server= (--servers= is also accepted). This argument specifies the machine where the requested command should be performed. If you omit this argument, the machine name will be pulled from the `COMPUTESERVER=` line of your client license file.

Note that the `--joblimit` and `--adminpassword` commands can only be applied to a single server at a time. If you specify multiple servers (either through the `--servers` switch or through your client license file), the command will only be applied to the first member of the list.

The following shows sample output from `gurobi_cl --status...`
Checking status of Gurobi Remote Services on server 'server1'...

Gurobi Remote Services (version 6.0.0) functioning normally
Available services: Distributed Worker, Compute Server
Job limit: 2, currently running: 2

Jobs currently running: 2 ...

<table>
<thead>
<tr>
<th>Client HostName</th>
<th>Client IP Address</th>
<th>UserName</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>client1</td>
<td>192.168.1.101</td>
<td>smith</td>
<td>7416</td>
</tr>
<tr>
<td>client2</td>
<td>192.168.1.102</td>
<td>jones</td>
<td>1536</td>
</tr>
</tbody>
</table>

Jobs currently queued: 1 ...

<table>
<thead>
<tr>
<th>Client HostName</th>
<th>Client IP Address</th>
<th>UserName</th>
<th>PID</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>client3</td>
<td>192.168.1.103</td>
<td>jim</td>
<td>2620</td>
<td>5</td>
</tr>
</tbody>
</table>

The report shows two jobs currently running (one from user smith on client machine client1, and one from user jones on client machine client2), and one job queued (from user jim on client machine client3).

Here are a few more example administrator commands:

> gurobi_cl --killjob=client1,7416
> gurobi_cl --adminpassword --server=gurobiserver1
> gurobi_cl --joblimit=0 --server=gurobiserver1
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.

When controlling the concurrent optimizer using these parameters, the strategies used for the different independent solves are chosen automatically. While we reserve the right to change our choices in the future, for LP models 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 one 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.

**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...
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:

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Current Node</th>
<th>Objective Bounds</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expl</td>
<td>Unexpl</td>
<td>Obj</td>
<td>Depth</td>
</tr>
<tr>
<td>0 0</td>
<td>- -</td>
<td>24.00000</td>
<td>13.00000</td>
</tr>
<tr>
<td>0 0</td>
<td>- -</td>
<td>16.50000</td>
<td>13.21154</td>
</tr>
<tr>
<td>0 0</td>
<td>- -</td>
<td>16.50000</td>
<td>13.25000</td>
</tr>
<tr>
<td>0 0</td>
<td>- -</td>
<td>16.50000</td>
<td>13.37500</td>
</tr>
<tr>
<td>0 0</td>
<td>- -</td>
<td>16.50000</td>
<td>13.37500</td>
</tr>
<tr>
<td>0 6</td>
<td>- -</td>
<td>16.50000</td>
<td>13.37500</td>
</tr>
<tr>
<td>310 149</td>
<td>- -</td>
<td>15.00000</td>
<td>13.66923</td>
</tr>
<tr>
<td>3873 1634</td>
<td>- -</td>
<td>15.00000</td>
<td>14.00000</td>
</tr>
<tr>
<td>9652 4298</td>
<td>- -</td>
<td>15.00000</td>
<td>14.12500</td>
</tr>
<tr>
<td>16535 6991</td>
<td>- -</td>
<td>15.00000</td>
<td>14.18056</td>
</tr>
<tr>
<td>23610 9427</td>
<td>- -</td>
<td>15.00000</td>
<td>14.22333</td>
</tr>
</tbody>
</table>

We also include node counts from one of the independent solves, as well as elapsed times, to give some indication of forward progress.

**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.
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, and both allow you to modify the same set of tuning parameters.

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), 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.

18.1 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:\gurobi550\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 7...

Method 2
MIPFocus 1

Solving with random seed #1 ... runtime 3.63s
Solving with random seed #2 ... runtime 4.12s+

Progress so far: baseline runtime 3.38s, best runtime 2.88s
Total elapsed tuning time 34s (66s remaining)
This output indicates that the tool has tried 7 parameter sets so far. For the seventh set, it changed the value of the **MIPFocus** parameter (the **Method** parameter was changed in our initial parameter settings, so this change will appear in every parameter set that the tool tries). The first trial solved the model in 3.63 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 worse 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 2.88s. 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 20 parameter sets in 97.89s

Baseline parameter set: runtime 3.38s

Improved parameter set 1 (runtime 1.62s):

Method 2
Heuristics 0
VarBranch 1
CutPasses 3
GomoryPasses 0

Improved parameter set 2 (runtime 2.03s):

Method 2
Heuristics 0
VarBranch 1
CutPasses 3

Improved parameter set 3 (runtime 2.38s):

Method 2
VarBranch 1

Wrote parameter files tune1.prm through tune3.prm
Wrote log files: tune1.log through tune3.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 **model3.prm** and **model1.log** through **model3.log**.

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 tradeoff 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...

```bash
> 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.

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.

### 18.2 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 Remote Services allow a machine to perform Gurobi computations on behalf of other machines. It is a Windows Service on Windows systems, and a daemon on Linux and Mac systems. The set of services provided will depend on your license. The most basic service is the Distributed Worker, which allows a machine to be used as a worker in a Distributed Algorithm. Another, more powerful service is Compute Server, which allows you to offload Gurobi computations from a set of client machines onto one or more servers. Later sections will discuss the use of these services. This section is devoted to the configuration and administration of Gurobi Remote Services.

19.1 Setting Up and Administering Gurobi Remote Services

Setting up Gurobi Remote Services is generally quite straightforward. One option is to simply follow the basic setup instructions in the Quick Start Guide. The default settings have been chosen to work well in most usage environment. However, even if you choose to use the defaults, you’ll probably want to be aware of the additional options and capabilities described here, including server parameters, firewall issues, and remote administration options.

Gurobi Remote Services Parameters

As noted in the Quick Start Guide, you start Gurobi Remote Services by running the `grb_rs` program on the server machine. This starts a Windows service on Windows systems, and a daemon on Linux or Mac OS systems. When Gurobi Remote Services starts, it picks up user parameter settings from an optional `grb_rs.cnf` file (the file must be in the directory that contains the `grb_rs` executable).

To modify the default settings, you should place a list of `parameter=value` lines in this configuration file. Lines that begin with the # symbol are treated as comments and are ignored. Here’s an example file:

```
# Configuration file
PASSWORD=abcd1234
ADMINPASSWORD=1234abcd
```

You can create this file using your favorite text editor (Notepad is a good choice on Windows).

Some Gurobi Remote Services parameters are generic, while others are specific to a particular service. The generic parameters are:

`PASSWORD`: The password that the client program must supply in order to submit a job. Note that all user data is passed between the client and server using 256-bit AES encryption, whether you supply a password or not. The server password simply prevents unauthorized clients from submitting jobs to the server. The default is no password.

`ADMINPASSWORD`: The password for performing administrative tasks. This is different from the password that client programs must provide. Administrative tasks are performed using the
gurobi_cl program. Example tasks include changing the job limit and killing jobs (details will follow). Note that the administrator password must be set if you wish to enable remote administration.

**THREADLIMIT**: A limit on the number of threads a single job can launch on the server. By default, a job can create as many threads as it likes (although by default the Gurobi algorithms won’t create more than one thread per core).

A few Gurobi Remote Services parameters are specific to Compute Server. They control the job queuing features...

**JOBLIMIT**: A limit on the number of client jobs that are allowed to run on the server at a time. Client requests beyond this limit are queued. The default limit is 2.

**HARDJOBLIMIT**: A hard limit on the number of simultaneous client jobs. Certain jobs (those with priority 100) are allowed to ignore the JOBLIMIT, but they aren’t allowed to ignore this limit. Client requests beyond this limit are queued. The default hard limit is 100.

**IGNOREPRIORITIES**: When set to 1, the server ignores user job priorities.

The configuration file is only read once, when Gurobi Remote Services first starts. Subsequent changes to the file won’t affect parameter values on a running server.

**Firewalls**

A machine running Gurobi Remote Services communicates with clients through a number of network ports on the server machine. By default, it uses ports 61000-65000. You generally don’t need to be aware of the details, since in most cases the server will either silently allow Gurobi Remote Services to use these ports or it will ask you to confirm that these ports can be used. However, some situations require you to manually open these ports.

One notable example is Amazon EC2, where most network ports are closed by default. You’ll need to create an EC2 Security Group (or modify your default group) when you start your instance in order to open these ports.

If for some reason our default port range is unavailable on your server, you can include a `PORT=` statement in both the client and the server license files to choose a different range. For example, the line:

```
PORT=43000
```

would use ports 43000-47000 instead.

If you run into trouble with firewall issues, we suggest you share this section with your network administrator.

**Administrative Commands**

Gurobi Remote Services provides a number of administration features. These allow you to check the status of a server, kill a running job, etc. All are accessed through the `gurobi_cl` command-line tool. Refer to this section (and this subsection in particular) for more information.
Copyright Notice for 3rd Party Library

Gurobi Compute Server uses an AES encryption library written Brian Gladman for encrypting messages between the client and the server. Here is the copyright notice for that library:

---------------------------------------------------------------------------
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights reserved.

The redistribution and use of this software (with or without changes) is allowed without the payment of fees or royalties provided that:

source code distributions include the above copyright notice, this list of conditions and the following disclaimer;

binary distributions include the above copyright notice, this list of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied warranties in respect of its operation, including, but not limited to, correctness and fitness for purpose.
---------------------------------------------------------------------------
Gurobi Optimizer implements a number of distributed algorithms that allow you to use multiple machines to solve a problem faster. Available distributed algorithms are:

- **A distributed MIP solver**, which allows you to divide the work of solving a single MIP model among multiple machines. A manager machine passes problem data to a set of worker machines in order to coordinate the overall solution process.

- **A distributed concurrent solver**, which allows you to use multiple machines to solve an LP or MIP model. Unlike the distributed MIP solver, the concurrent solver doesn’t divide the work associated with solving the problem among the machines. Instead, each machine uses a different strategy to solve the whole problem, with the hope that one strategy will be particularly effective and will finish much earlier than the others. For some problems, this concurrent approach can be more effective than attempting to divide up the work.

- **Distributed parameter tuning**, which automatically searches for parameter settings that improve performance on your optimization model. Tuning solves your model with a variety of parameter settings, measuring the performance obtained by each set, and then uses the results to identify the settings that produce the best overall performance. The distributed version of tuning performs these trials on multiple machines, which makes the overall tuning process run much faster.

These distributed algorithms are designed to be nearly transparent to the user. The user simply modifies a few parameters, and the work of distributing the computation among multiple machines is handled behind the scenes by the Gurobi library.

### 20.1 Configuring a Distributed Worker Pool

Before your program can perform a distributed optimization task, you’ll need to identify a set of machines to use as your distributed workers. Ideally these machines should give very similar performance. Identical performance is best, especially for distributed tuning, but small variations in performance won’t hurt your overall results too much.

**Specifying the Distributed Worker Pool**

Once you’ve identified your distributed worker machines, you’ll need to start Gurobi Remote Services on these machines. Instructions for setting up Gurobi Remote Services can be found in the Gurobi Quick Start Guide. As noted in the Quick Start Guide, run following command to make sure a machine is available to be used as a distributed worker:

```bash
> gurobi_cl --server=machine --status
```

(replace `machine` with the name or IP address of your machine). If you see Distributed Worker listed among the set of available services...
Gurobi Remote Services (version 6.0.0) functioning normally
Available services: Distributed Worker

then that machine is good to go.

We should reiterate a point that is raised in the Quick Start Guide: you do not need a Gurobi license to run Gurobi Remote Services on a machine. Some services are only available with a license (e.g., Compute Server). However, any machine that is running Gurobi Remote Services will provide the Distributed Worker service.

**The Distributed Manager Machine**

Once you have identified a set of distributed worker machines, you’ll need to choose a manager machine. This is the machine where your application actually runs. In addition to building the optimization model, your manager machine will coordinate the efforts of the distributed workers during the execution of the distributed algorithm.

Note that once the distributed algorithm completes, only the manager retains any information about the solution. The distributed workers go off to work on other things.

You’ll need to choose a manager machine that is licensed to run the distributed algorithms. You’ll see a DISTRIBUTED= line in your license file if distributed algorithms are enabled.

Note that, by default, the manager does not participate in the distributed optimization. It simply coordinates the efforts of the distributed workers. If you would like the manager to also act as one of the workers, you’ll need to start Gurobi Remote Services on the manager machine as well.
The workload associated with managing the distributed algorithm is quite light, so a machine can easily handle both the manager and worker role without degrading performance.

**Specifying the Distributed Worker Pool**

If you’d like to invoke a distributed algorithm from your application, you’ll need to provide the names of the distributed worker machines. You do this by setting the `WorkerPool` parameter (refer to the Gurobi Parameter section for information on how to set a parameter). The parameter should be set to a string that contains a comma-separated list of either machine names or IP addresses. For example, you might use the following in your `gurobi_cl` command line:

```plaintext
> gurobi_cl WorkerPool=server1,server2,server3 ...
```

If you have set up an access password on the distributed worker machines, you’ll need to provide it through the `WorkerPassword` parameter. All machines in the worker pool must have the same access password.

Note that providing a list of available workers is strictly a configuration step. Your program won’t actually use any of the distributed algorithms unless it specifically requests them. Instructions for doing so are next.

**Requesting A Distributed Algorithm**

Once you’ve set the `WorkerPool` parameter to the appropriate value, your final step is to set the `ConcurrentJobs`, `DistributedMIPJobs`, or `TuneJobs` parameter. These parameters indicate how many distinct distributed worker jobs you would like to start. For example, if you set `TuneJobs` to 2 in `grbtune`...

```plaintext
> grbtune WorkerPool=server1,server2 TuneJobs=2 misc07.mps
```

...you should see the following output in the log...

```
Started distributed worker on server1
Started distributed worker on server2
```

Distributed tuning: launched 2 distributed worker jobs

This output indicates that two jobs have been launched, one on machine `server1` and the other on machine `server2`. These two jobs will continue to run until your tuning run completes.

Similarly, if you launch distributed MIP...

```plaintext
> gurobi_cl WorkerPool=server1,server2 DistributedMIPJobs=2 misc07.mps
```

...you should see the following output in the log...

```
Started distributed worker on server1
Started distributed worker on server2
```

Distributed MIP job count: 2

Note that, in most cases, each machine runs one distributed worker job at a time. Distributed workers are allocated on a first-come, first-served basis, so if multiple users are sharing a set of distributed worker machines, you should be prepared for the possibility that some or all of them may be busy when the manager requests them. The manager will grab as many as it can, up to the requested count. If none are available, it will return an error.
Compute Server Considerations

If you have one or more Gurobi Compute Servers, you can use them for distributed optimization as well. Compute Servers offer a lot more flexibility than distributed workers, though, so they require a bit of additional explanation.

The first point you should be aware of is that one Compute Server can actually host multiple distributed worker jobs. Compute Servers allow you to set a limit on the number of jobs that can run simultaneously. Each of those jobs can be a distributed worker. For example, if you have a pair of Compute Servers, each with a job limit of 2, then issuing the command...

```bash
> gurobi_cl DistributedMIPJobs=3 WorkerPool=server1,server2 misc07.mps
```

...would produce the following output...

Started distributed worker on server1  
Started distributed worker on server2  
Started distributed worker on server1

Compute Server assigns a new job to the machine with the most available capacity, so assuming that the two servers are otherwise idle, the first distributed worker job would be assigned to server1, the second to server2, and the third to server1.

Another point to note is that, if you are working in a Compute Server environment, it is often better to use the Compute Server itself as the distributed manager, rather than the client machine. This is particularly true if the Compute Server and the workers are physically close to each other, but physically distant from the client machine. In a typical environment, the client machine will offload the Gurobi computations onto the Compute Server, and the Compute Server will then act as the manager for the distributed computation.

To give an example, running the following command on machine client1:

```bash
> gurobi_cl --server=server1 WorkerPool=server1,server2 DistributeMIPJobs=2 misc07.mps
```

...will lead to the following sequence of events...

- The model will be read from the disk on client1 and passed to Compute Server server1.
- Machine server1 will act as the manager of the distributed optimization.
- Machine server1 will start two distributed worker jobs, one that also runs on server1 and another that runs on server2.

Compute Server provides load balancing among multiple machines, so it is common for the user to provide a list of available servers when a Gurobi application starts. We'll automatically copy this list into the WorkerPool parameter. Of course, you can change the value of this parameter in your program, but the default behavior is to draw from the same set of machines for the distributed workers. Thus, the following command would be equivalent to the previous command:

```bash
> gurobi_cl --server=server1,server2 DistributedMIPJobs=2 misc07.mps
```

Please refer to the next section section for more information on using a Gurobi Compute Server.
20.2 Writing Your Own Distributed Algorithms

Gurobi provides a set of routines that allow you to write your own distributed algorithms. Doing so requires a Compute Server, though. This capability will be discussed in the Compute Server section.

20.3 Distributed Algorithm Considerations

So far in this section, we’ve focused almost entirely on configuration and setup issues for the distributed algorithms in this section. These algorithms have been designed to be nearly indistinguishable from the single machine versions. Our hope is that, if you know how to use the single machine version, you’ll find it straightforward to use the distributed version. The distributed algorithms respect all of the usual parameters. For distributed MIP, you can adjust strategies, adjust tolerances, set limits, etc. For concurrent MIP, you can allow Gurobi to choose the settings for each machine automatically or you can use concurrent environments to make your own choices. For distributed tuning, you can use the usual tuning parameters, including TuneTimeLimit, TuneTrials, and TuneOutput.

Performance Across Distributed Workers

There are a few things to be aware of when using distributed algorithms, though. One relates to relative machine performance. As we noted earlier, distributed algorithms work best if all of the workers give very similar performance. For example, if one machine in your worker pool were much slower than the others in a distributed tuning run, any parameter sets tested on the slower machine would appear to be less effective than if they were run on a faster machine. Similar considerations apply for distributed MIP and distributed concurrent. We strongly recommend that you use machines with very similar performance. Note that if your machines have similarly performing cores but different numbers of cores, we suggest that you use the Threads parameter to make sure that all machines use the same number of cores.

Callbacks

Another difference between the distributed algorithms and our single-machine algorithms is in the callbacks. The distributed MIP and distributed concurrent solvers do not provide the full range of callbacks that are available with our standard solvers. They will only provide the MIP, MIPNODE, and POLLING callbacks. See the Callback section for details on the different callback types.

Logging

The distributed algorithms provide slightly different logging information from the standard algorithms. Consult the Distributed MIP Logging section for details.
This section describes Gurobi Compute Server, an optional component of Gurobi Remote Services that allows you to choose one or more servers to run your Gurobi computations. You can then offload the work associated with solving optimization problems onto these servers from as many client machines as you like:

When considering a program that uses Gurobi Compute Server, you can think of the optimization as being split into two parts. A client program builds an optimization model using any of the standard Gurobi interfaces (C, C++, Java, .NET, Python, MATLAB, R). This happens in the left box of this figure:

All of our API’s sit on top of our C API. The C API is in charge of building the internal model data structures, invoking the Gurobi algorithms, retrieving solution information, etc. When running Gurobi on a single machine, the C API would build the necessary data structures in
memory. The Gurobi algorithms would take the data stored in these data structures as input, and produce solution data as output.

When running on a Compute Server, the C API instead passes model data to the server, where it is stored. When the Gurobi algorithms are invoked, the C API simply passes a message to the server, indicating that optimization should be performed on the stored model data. Solution data is computed and stored on the server. When the client program later queries the solution information, the client sends a message to the server in order to obtain the requested data. All communication between the client and server happens behind the scenes.

In other words, the overall process can be viewed as happening in three phases:

*Client computer uses any Gurobi API to build model; Gurobi library passes model data to server*

*Gurobi Compute Server solves the model*

*Gurobi library retrieves results on the client computer*

Of course, programs that use the Gurobi API’s in more complex ways would have additional steps.

Gurobi Compute Servers support queuing and load balancing. You can set a limit on the number of simultaneous jobs each Compute Server will run. When this limit has been reached, subsequent jobs will be queued. If you have multiple Compute Servers, the current job load is automatically balanced among the available servers.
By default, the Gurobi job queue is serviced in a First-In, First-Out (FIFO) fashion. However, jobs can be given different priorities (through a client license file, or through API calls). Jobs with higher priorities are then selected from the queue before jobs with lower priorities.

While the Gurobi Compute Server is meant to be transparent to both developers and users, there are a few aspects of Compute Server usage that you do need to be aware of. These include performance considerations, APIs for configuring client programs, and a few features that are not supported for Compute Server applications. Please proceed to Compute Server usage for details.

### 21.1 Setting Up and Administering a Gurobi Compute Server

To use Compute Server, you’ll need to start Gurobi Remote Services on one or more servers. Once you’ve got a Compute Server running, you can check to make sure that you will be able to submit jobs to it by issuing the following command from any machine that can reach the server on your network:

```bash
> gurobi_cl --server=servername --status
```

(replace `servername` with the name of your server). If you see the following lines in the resulting output, Compute Server is ready to go:

```
Gurobi Remote Services (version 6.0.0) functioning normally
Available services: Distributed Worker, Compute Server
```

Compute Server provides a number of user configurable parameters to control things like the user and administrator passwords, limits on the number of jobs that can run simultaneously, etc. Compute Server also has a number of administrative commands that allow you to kill jobs, obtain a list of running and queued jobs, and change parameters. Please refer to the Gurobi Remote Services section for details.

### 21.2 Compute Server Usage

The Gurobi Compute Server feature was designed to be almost entirely transparent to both the developers and the users of the programs that use it. However, there are a few topics that you may need to be aware of, including setting up a Compute Server client, setting job priorities, performance considerations, callbacks, and a few coding practices for Compute Server.

#### Client Configuration

Compute Server clients must know how to reach the desired servers. You have a few options for providing this information. Your first is through the client `gurobi.lic` file. That file should contain a line like the following:

```
COMPUTESERVER=server1.mydomain.com,server2.mydomain.com
```

You can create this license file yourself, using your favorite text editor (`Notepad` is a good choice on Windows). You simply need to provide a list of the names of the machines that are acting as Compute Servers. You can refer to the Compute Server machines using their names (e.g., server.mydomain.com) or their IP addresses (e.g., 192.168.1.100).

Your client license file may optionally specify a few additional pieces of information. The first is the Compute Server password:
PASSWORD=abcd

This should match the password that you chose when you started the Compute Server. The second is the job priority:

PRIORITY=10

As you might expect, higher priority jobs take precedence over lower priority jobs. Priorities will be discussed in more detail shortly. The third is the queuing timeout:

TIMEOUT=60

A job that has been sitting in queue for longer than the specified TIMEOUT value (in seconds) will return with a JOB_REJECTED error.

Your second option for specifying the desired Compute Servers is through API calls. The appropriate call depends on your programming language. From C, you would call GRBloadclientenv. From our object-oriented interfaces, the GRBEnv constructors each provide a signature that allows you to specify the compute server(s), the compute server password, the job priority, and a timeout for jobs submitted by that program.

Your final option for specifying the desiredCompute Servers is specific to the Gurobi command-line tool. The --server= or --servers= argument allows you to provide a comma-separated list of Compute Servers (and the optional --password argument allows you to specify the user password):

> gurobi_cl --servers=server1,server2 --password=password1 misc07.mps

Job Priorities

As noted earlier, Gurobi Compute Servers support job priorities. You can assign an integer priority between -100 and 100 to each job (the default is 0). When choosing among queued jobs, the Compute Server will run the highest priority job first. Note that servers will never preempt running jobs.

We have chosen to give priority 100 a special meaning. A priority 100 job will start immediately, even if this means that a server will exceed its job limit. You should be cautious with priority 100 jobs, since submitting too many at once could lead to very high server loads, which could lead to poor performance and even crashes in extreme cases.

Performance Considerations on a Wide-Area Network (WAN)

While using Gurobi Compute Server doesn't typically require you to make any modifications to your code, performance considerations can sometimes force you to do some tuning when your client and server are connected by a slow network (e.g., the internet). We'll briefly talk about the source of the issue, and the changes required to work around it.

In a Gurobi Compute Server, a call to a Gurobi routine typically results in network messages between the client and the server. While each individual message is not that expensive, sending hundreds or thousands of messages can be quite time-consuming. Compute Server makes heavy use of caching to reduce the number of such messages. However, not all methods are cached. As a result, we suggest that you avoid doing the following things:
• Retrieving the non-zero values for individual rows and columns of the constraint matrix (using, for example, GRBgetconstrs in C, GRBModel::getRow in C++, GRBModel.getRow in Java, GRBModel.GetRow in .NET, and Model.getRow in Python).

• Retrieving individual string-valued attributes.

Please note that you don’t need to be too concerned about this issue. Caching generally works well. In particular, when building a model, our lazy update approach avoids the issue entirely. You should feel free to build your model one constraint at a time, for example. Your changes are communicated to the server in one large message when you request a model update.

Of course, network overhead depends on both the number of messages that are sent and the sizes of these messages. We automatically perform data compression to reduce the time spent transferring very large messages. However, as you may expect, you will notice some lag when solving very large models over slow networks.

**Callbacks**

As you might imagine, since the actual optimization task runs on a remote system in a Compute Server environment, Gurobi callbacks give different behavior than they do when the task runs locally. In particular, callbacks are both less frequent and more restrictive. You will only receive MESSAGE, BARRIER, SIMPLEX, MIP, and MIPSOL callbacks; you will not receive PRESOLVE or MIPNODE callbacks. As a result, you will only have access to a subset of the callback information that you would be able to obtain when running locally. You can still request that the optimization be terminated from any of the callbacks you receive. Please refer to the Callback Code section for more information on the various callback codes.

**Developing for Compute Server**

With only a few exceptions, using Gurobi Compute Server requires no changes to your program. This section covers the exceptions. We’ll talk about program robustness issues that may arise specifically in a Compute Server environment, and we’ll give a full list of the Gurobi features that aren’t supported in Compute Server.

**Coding for Robustness**

Client-server computing introduces a few robustness situations that you wouldn’t face when all of your computation happens on a single machine. Specifically, by passing data between a client and a server, your program is dependent on both machines being available, and on an uninterrupted network connection between the two systems. The queuing and failover capabilities of Gurobi Compute Server can handle the vast majority of issues that may come up, but you can take a few additional steps in your program if you want to achieve the maximum possible robustness.

The one scenario you may need to guard against is the situation where you lose the connection to the server while the portion of your program that builds and solves an optimization model is running. Gurobi Compute Server will automatically route queued jobs to another server, but jobs that are running when the server goes down are interrupted (the client will receive a NETWORK error). If you want your program to be able to survive such failures, you will need to architect it in such a way that it will rebuild and resolve the optimization model in response to a NETWORK error. The exact steps for doing so are application dependent, but they generally involve encapsulating...
the code between the initial Gurobi environment creation and the last Gurobi call into a function that can be reinvoked in case of an error.

Features Not Supported in Compute Server

As noted earlier, there are a few Gurobi features that are not supported in Compute Server. We’ve mentioned some of them already, but we’ll give the full list here for completeness. You will need to avoid using these features if you want your application to work in a Compute Server environment.

The unsupported features are:

- **Lazy constraints:** While we do provide MIPSOL callbacks, we don’t allow you to add lazy constraints to cut off the associated MIP solutions.

- **User cuts:** The MIPNODE callback isn’t supported, so you won’t have the opportunity to add your own cuts. User cuts aren’t necessary for correctness, but applications that heavily rely on them may experience performance issues.

- **Multi-threading within a single Gurobi environment:** This isn’t actually supported in Gurobi programs in general, but the results in a Compute Server environment are sufficiently difficult to track down that we wanted to mention it again here. All models built from an environment share a single socket connection to the Compute Server. This one socket can’t handle multiple simultaneous messages. If you wish to call Gurobi from multiple threads in the same program, you should make sure that each thread works within its own Gurobi environment.

- **Advanced simplex basis routines:** The C routines that work with the simplex basis (GRBFSolve, GRBBSolve, GRBBinvColj, GRBBinvRowi, and GRBgetBasisHead) are not supported.

Acknowledgement of 3rd Party Icons

The icons used in this chapter come from the Open Security Architecture.