


# Cardinal Optimizer (COPT) User Guide

*Ver 8.0.7*

Cardinal Operations

D. Ge, Q. Huangfu, Z. Wang, J. Wu and Y. Ye

Sep 20, 2026

# Contents


**1 Introduction to Cardinal Optimizer** **1**
- 1.1 Overview . . . 1
- 1.2 Licenses . . . 3
- 1.3 How to Cite . . . 3
- 1.4 Contact Information . . . 4

**2 Installation Guide** **5**
- 2.1 Registration . . . 5
- 2.2 Software Installation . . . 5
  - 2.2.1 Windows . . . 5
  - 2.2.2 MacOS . . . 8
  - 2.2.3 Linux . . . 12
- 2.3 Setting Up License . . . 13
  - 2.3.1 Obtaining License . . . 13
  - 2.3.2 Verifying License . . . 14
  - 2.3.3 Installing License . . . 15
  - 2.3.4 Others . . . 18
- 2.4 Verify the installation and configuration . . . 18
- 2.5 Upgrade . . . 19

**3 COPT Command-Line** **21**
- 3.1 Overview . . . 21
- 3.2 Edit mode . . . 21
- 3.3 Script mode . . . 22
- 3.4 Shell commands . . . 23
  - 3.4.1 General shell commands . . . 23
  - 3.4.2 COPT shell commands . . . 24
- 3.5 Example Usage . . . 26

**4 COPT Floating Licensing service** **29**
- 4.1 Server Setup . . . 29
  - 4.1.1 Installation . . . 29
  - 4.1.2 Floating License . . . 30
  - 4.1.3 Configuration . . . 31
  - 4.1.4 Web License for Floating Server . . . 32
  - 4.1.5 Example Usage . . . 33
- 4.2 Client Setup . . . 33
  - 4.2.1 Configuration . . . 33
  - 4.2.2 High Availability . . . 34
  - 4.2.3 Example Usage . . . 34
- 4.3 Floating Token Server Managing Tool . . . 36
  - 4.3.1 Tool Usage . . . 36
  - 4.3.2 Example Usage . . . 37

4.4 Running as service . . . 38
4.4.1 Linux . . . 39
4.4.2 MacOS . . . 40

**5 COPT Compute Cluster Service 43**
5.1 Server Setup . . . 43
5.1.1 Installation . . . 43
5.1.2 Cluster License . . . 44
5.1.3 Configuration . . . 45
5.1.4 Web License for Compute Cluster . . . 49
5.1.5 Example Usage . . . 50
5.2 Server Memory Management . . . 50
5.2.1 Per-task memory cap . . . 51
5.2.2 Memory watermark defense . . . 51
5.2.3 Memory cap exemptions . . . 52
5.2.4 Compatibility with older COPT versions . . . 52
5.2.5 Deployment suggestions . . . 53
5.3 Client Setup . . . 53
5.3.1 Configuration . . . 53
5.3.2 High Availability . . . 54
5.3.3 Example Usage . . . 55
5.4 COPT Cluster Managing Tool . . . 56
5.4.1 Tool Usage . . . 56
5.4.2 Example Usage . . . 58
5.5 Running as service . . . 61
5.5.1 Linux . . . 62
5.5.2 MacOS . . . 63

**6 COPT Web Licensing Service 65**

**7 COPT Quick Start 67**
7.1 C Interface . . . 67
7.1.1 Example details . . . 67
7.1.2 Build and run . . . 74
7.2 C++ Interface . . . 75
7.2.1 Example details . . . 75
7.2.2 Build and Run . . . 79
7.3 C# Interface . . . 80
7.3.1 Example details . . . 80
7.3.2 Build and Run . . . 85
7.4 Java Interface . . . 86
7.4.1 Example details . . . 86
7.4.2 Build and Run . . . 91
7.5 Python Interface . . . 91
7.5.1 Installation guide . . . 91
7.5.2 Example details . . . 94
7.5.3 Best Practice . . . 97
7.6 AMPL Interface . . . 100
7.6.1 Installation Guide . . . 100
7.6.2 Solver Options and Exit Codes . . . 101
7.6.3 Example Usage . . . 103
7.7 Pyomo Interface . . . 106
7.7.1 Installation Guide . . . 106
7.7.2 Example Usage . . . 106
7.8 PuLP Interface . . . 112
7.8.1 Installation guide . . . 112
7.8.2 Introduction of features . . . 113
7.9 CVXPY Interface . . . 114
7.9.1 Installation guide . . . 115

7.9.2 Introduction of features . . . 115

**8 General Constants** **117**
8.1 Version information . . . 117
8.2 Optimization directions . . . 117
8.3 Infinity and undefined value . . . 118
8.4 Constraint senses . . . 118
8.5 Variable types . . . 119
8.6 SOS constraint types . . . 119
8.7 Indicator constraint types . . . 119
8.8 SOC constraint types . . . 120
8.9 Exponential Cone type . . . 120
8.10 Quadratic objective function . . . 121
8.11 Quadratic constraint . . . 121
8.12 Basis status . . . 121
8.13 Solution status . . . 121
8.14 Callback context . . . 122
8.15 Nonlinear Model Callback Evaluation Types . . . 123
8.16 Dense Storage Types for Nonlinear Matrices . . . 123
8.17 Client configuration . . . 123
8.18 API function return code . . . 124
8.19 Methods for accessing constants . . . 125

**9 Attributes** **127**
9.1 Problem related . . . 127
9.2 Solution related . . . 132
9.3 Methods for accessing attributes . . . 136

**10 Information** **137**
10.1 Problem information . . . 138
10.2 Solution and sensitivity analysis information . . . 138
10.3 Dual Farkas and primal ray . . . 139
10.4 Feasibility relaxation information . . . 140
10.5 Callback information . . . 140
10.6 Methods for accessing information . . . 141

**11 Parameters** **143**
11.1 Limits and tolerances . . . 143
11.2 Presolving and scaling . . . 147
11.3 Linear programming related . . . 148
11.4 Integer Programming related . . . 151
11.5 Semidefinite Programming related . . . 155
11.6 Nonlinear Programming related . . . 156
11.7 Multi-objective Optimization related . . . 157
11.8 Parallel computing related . . . 157
11.9 GPU computing related . . . 159
11.10 IIS computation related . . . 160
11.11 Feasibility relaxation related . . . 160
11.12 Parameter Tuning related . . . 161
11.13 Callback related . . . 163
11.14 Other parameters . . . 163
11.15 Methods for accessing and setting parameters . . . 164

**12 Modeling and Solving Optimization Problems** **165**
12.1 Linear Programming (LP) . . . 167
12.1.1 Mathematical Model . . . 167
12.1.2 Modeling . . . 167
12.1.3 Solving . . . 169
12.1.4 Related Attributes and Information . . . 169

12.2 Second-Order Cone Programming (SOCP) . . . 170
12.2.1 Mathematical Model . . . 170
12.2.2 Modeling . . . 171
12.3 ExpCone Programming . . . 174
12.3.1 Mathematical Formulation . . . 174
12.3.2 Exponential Cone Example . . . 174
12.4 Semidefinite Programming (SDP) . . . 175
12.4.1 Mathematical Model . . . 175
12.4.2 Modeling . . . 176
12.4.3 Solving . . . 179
12.4.4 Related Attributes . . . 180
12.5 Quadratic Programming (QP) . . . 180
12.5.1 Mathematical Formulation . . . 180
12.6 Quadratically Constrained Programming (QCP) . . . 180
12.6.1 Mathematical Model . . . 181
12.6.2 Modeling . . . 181
12.6.3 Related Attributes . . . 184
12.7 General Nonlinear Programming (NLP) . . . 184
12.7.1 Explicit Expression Nonlinear Modeling . . . 184
12.7.2 Callback-Based Nonlinear Modeling . . . 185
12.8 Mixed-Integer Programming (MIP) . . . 186
12.8.1 Modeling . . . 186
12.8.2 Solving . . . 187
12.8.3 Related Attributes . . . 187
12.9 Special Constraints . . . 187
12.9.1 SOS Constraints . . . 187
12.9.2 Indicator Constraints . . . 189
12.9.3 Attributes for Special Constraints . . . 191
12.9.4 IIS Status of Special Constraints . . . 192

**13 Handling Infeasible Models** **193**
13.1 IIS for Infeasible Models . . . 193
13.1.1 Computing IIS . . . 194
13.1.2 Getting IIS status of variables and constraints . . . 195
13.1.3 IIS-related parameters, attributes and information . . . 197
13.2 Feasibility Relaxation for infeasible models . . . 198
13.2.1 Computing Feasibility Relaxation . . . 198
13.2.2 Feasrelax-related parameters, attributes and information . . . 199

**14 MIP Starts** **201**
14.1 Utilities of MIP Starts . . . 201
14.1.1 Set and Load MIP Starts . . . 201
14.1.2 Read and Write MIP Starts . . . 202
14.2 Related Parameters . . . 202
14.3 Log of MIP starts . . . 203
14.3.1 MIP starts are accepted . . . 203
14.3.2 MIP starts are rejected . . . 203

**15 MIP Solution Pool** **205**

**16 COPT Tuner** **207**
16.1 Introduction . . . 207
16.2 Related parameters . . . 207
16.3 Provided capabilities . . . 208
16.3.1 Tuning method . . . 208
16.3.2 Tuning mode . . . 208
16.3.3 Tuning permutations . . . 208
16.3.4 Tuning measure . . . 209
16.3.5 Tuning targets . . . 209

16.3.6 Tuning output . . . 209
16.3.7 TuneTimeLimit . . . 209
16.3.8 User defined parts . . . 209
16.3.9 Loading or writing tuning parameters . . . 210
16.4 Example . . . 210

**17 Callbacks** **211**
17.1 Obtaining information during the solving process . . . 212
17.2 Controlling the MIP solving process . . . 212
17.2.1 Adding lazy constraints . . . 213
17.2.2 Adding cutting planes . . . 214
17.2.3 Adding feasible solutions . . . 214
17.3 Using the callback utilities in different APIs . . . 215

**18 Matrix Modeling** **217**
18.1 Two different matrix modeling modes . . . 217
18.2 Multi-dimensional Variables . . . 219
18.3 Multi-dimensional array operations and expressions . . . 219
18.3.1 Multi-dimensional Linear Expressions . . . 219
18.3.2 Multi-dimensional Quadratic Expression . . . 220
18.3.3 Other multi-dimensional array operations . . . 220
18.4 Matrix Constraints . . . 220
18.4.1 Matrix linear Constraints . . . 220
18.4.2 Quadratic Constraints . . . 221
18.5 Objective function composed of multi-dimensional variables . . . 222
18.6 Indexing and Slicing . . . 222
18.6.1 Integer Indexing . . . 222
18.6.2 Slice Indexing . . . 223
18.6.3 Tuple Indexing . . . 223

**19 Multi-objective Optimization** **225**
19.1 Modeling Multiple objectives . . . 225
19.2 Solving Multiple objectives . . . 226
19.2.1 Hierarchy Method . . . 226
19.2.2 Weighted-sum Method . . . 226
19.2.3 Objective degradation tolerance . . . 227
19.3 Retrieving the results of multiple objectives . . . 228

**20 Logging** **229**
20.1 Parameters and Functions for Logging . . . 229
20.2 Basic Information Section . . . 230
20.2.1 Solver Environment and Model Overview . . . 230
20.2.2 Model Size and Presolve . . . 231
20.2.3 Problem Numerical Characteristics . . . 231
20.3 Simplex Logging . . . 232
20.3.1 Simplex Iteration Process . . . 232
20.3.2 Solution Summary . . . 232
20.4 Barrier Logging . . . 233
20.4.1 Barrier Iteration Process . . . 233
20.4.2 Barrier Summary . . . 234
20.4.3 Solution Summary . . . 234
20.5 Branch-and-Cut Logging . . . 235
20.5.1 Branch-and-Cut Search Process . . . 235
20.5.2 Solution Summary . . . 236
20.6 First-order Method (PDLP) Logging for GPU Solver . . . 237
20.6.1 GPU Hardware Information . . . 237
20.6.2 First-order Method(PDLP) Iteration Process . . . 237

**21 File formats** **239**

21.1 File format list . . . 239
21.2 File I/O operations . . . 239
21.3 Model file introduction . . . 240

**22 FAQs** **243**
22.1 Installation and Licensing Configuration Related . . . 243
22.1.1 MacOS System . . . 244
22.1.2 Windows System . . . 244
22.2 Modeling and Solving Functions Related . . . 245
22.3 GPU Usage Related . . . 245
22.4 CPU and Memory Binding Related . . . 246

**23 C API Reference** **249**
23.1 Constants . . . 249
23.1.1 Optimization directions . . . 249
23.1.2 Infinity . . . 249
23.1.3 Undefined Value . . . 250
23.1.4 Constraint senses . . . 250
23.1.5 Variable types . . . 250
23.1.6 SOS-constraint types . . . 251
23.1.7 Indicator constraint . . . 251
23.1.8 SOC constraint type . . . 252
23.1.9 Exponential cone constraint type . . . 252
23.1.10 Quadratic objective function . . . 252
23.1.11 Quadratic constraint . . . 253
23.1.12 Basis status . . . 253
23.1.13 Optimization status . . . 253
23.1.14 LP solution status . . . 254
23.1.15 MIP solution status . . . 255
23.1.16 Callback context . . . 256
23.1.17 Nonlinear Expression Operators . . . 256
23.1.18 API function return code . . . 257
23.1.19 Client configuration . . . 258
23.1.20 Other constants . . . 258
23.2 Attributes . . . 259
23.2.1 Problem attribute . . . 259
23.2.2 Solution attribute . . . 263
23.3 Information . . . 266
23.3.1 Problem information . . . 266
23.3.2 Solution and sensitivity analysis information . . . 266
23.3.3 Dual Farkas and primal ray . . . 267
23.3.4 Feasibility relaxation information . . . 268
23.4 Callback information . . . 268
23.5 Parameters . . . 269
23.5.1 Limits and tolerances . . . 269
23.5.2 Presolving and scaling . . . 272
23.5.3 Linear Programming related . . . 272
23.5.4 Semidefinite Programming related . . . 275
23.5.5 Integer Programming related . . . 275
23.5.6 Nonlinear Programming related . . . 279
23.5.7 Parallel computing related . . . 280
23.5.8 IIS computation related . . . 281
23.5.9 Feasibility relaxation related . . . 282
23.5.10 Tuner related . . . 282
23.5.11 Callback related . . . 284
23.5.12 GPU computing related . . . 284
23.5.13 Multi-objective Optimization . . . 285
23.5.14 Other parameters . . . 285

23.6 API Functions . . . 286
23.6.1 Creating the environment and problem . . . 286
23.6.2 Building and modifying a problem . . . 290
23.6.3 Reading and writing the problem . . . 324
23.6.4 Solving the problem and accessing solutions . . . 328
23.6.5 Accessing information of problem . . . 337
23.6.6 Accessing and setting parameters . . . 369
23.6.7 Accessing attributes . . . 373
23.6.8 Logging utilities . . . 374
23.6.9 MIP start utilities . . . 374
23.6.10 Nonlinear start point utilities . . . 375
23.6.11 IIS utilities . . . 376
23.6.12 Feasibility relaxation utilities . . . 379
23.6.13 Parameter tuning utilities . . . 380
23.6.14 Callback utilities . . . 382
23.6.15 Other API functions . . . 391
23.6.16 Multi-objective Model API Functions . . . 392
23.6.17 CPU and Memory Binding Functions . . . 400

**24 Python API Reference** **403**
24.1 Constants . . . 403
24.1.1 General Constants . . . 403
24.1.2 Attributes . . . 403
24.1.3 Information . . . 404
24.1.4 Callback Information . . . 404
24.1.5 Parameters . . . 404
24.2 Python Modeling Classes . . . 404
24.2.1 EnvrConfig Class . . . 405
24.2.2 Envr Class . . . 405
24.2.3 Model Class . . . 408
24.2.4 Var Class . . . 497
24.2.5 VarArray Class . . . 501
24.2.6 PsdVar Class . . . 503
24.2.7 PsdVarArray Class . . . 507
24.2.8 SymMatrix Class . . . 508
24.2.9 SymMatrixArray Class . . . 509
24.2.10 Constraint Class . . . 511
24.2.11 ConstrArray Class . . . 514
24.2.12 ConstrBuilder Class . . . 516
24.2.13 ConstrBuilderArray Class . . . 517
24.2.14 QConstraint Class . . . 519
24.2.15 QConstrArray Class . . . 522
24.2.16 QConstrBuilder Class . . . 524
24.2.17 QConstrBuilderArray Class . . . 525
24.2.18 PsdConstraint Class . . . 527
24.2.19 PsdConstrArray Class . . . 529
24.2.20 PsdConstrBuilder Class . . . 530
24.2.21 PsdConstrBuilderArray Class . . . 532
24.2.22 LmiConstraint Class . . . 534
24.2.23 LmiConstrArray Class . . . 538
24.2.24 NlConstraint Class . . . 540
24.2.25 NlConstrArray Class . . . 542
24.2.26 NlConstrBuilder Class . . . 543
24.2.27 NlConstrBuilderArray Class . . . 545
24.2.28 SOS Class . . . 546
24.2.29 SOSArray Class . . . 546
24.2.30 SOSBuilder Class . . . 548
24.2.31 SOSBuilderArray Class . . . 551

24.2.32 Cone Class . . . 552
24.2.33 ConeArray Class . . . 553
24.2.34 ConeBuilder Class . . . 554
24.2.35 ConeBuilderArray Class . . . 556
24.2.36 ExpCone Class . . . 558
24.2.37 ExpConeArray Class . . . 559
24.2.38 ExpConeBuilder Class . . . 560
24.2.39 ExpConeBuilderArray Class . . . 562
24.2.40 AffineCone Class . . . 564
24.2.41 AffineConeArray Class . . . 565
24.2.42 AffineConeBuilder Class . . . 567
24.2.43 AffineConeBuilderArray Class . . . 570
24.2.44 GenConstr Class . . . 572
24.2.45 GenConstrArray Class . . . 573
24.2.46 GenConstrBuilder Class . . . 575
24.2.47 GenConstrBuilderArray Class . . . 577
24.2.48 Column Class . . . 579
24.2.49 ColumnArray Class . . . 583
24.2.50 MVar Class . . . 585
24.2.51 MConstr Class . . . 591
24.2.52 MConstrBuilder Class . . . 595
24.2.53 MQConstr Class . . . 595
24.2.54 MQConstrBuilder Class . . . 599
24.2.55 MPsdConstr Class . . . 600
24.2.56 MPsdConstrBuilder Class . . . 602
24.2.57 MLinExpr Class . . . 604
24.2.58 MQuadExpr Class . . . 610
24.2.59 NdArray Class . . . 616
24.2.60 ExprBuilder Class . . . 625
24.2.61 LinExpr Class . . . 628
24.2.62 QuadExpr Class . . . 634
24.2.63 PsdExpr Class . . . 640
24.2.64 MPsdExpr Class . . . 647
24.2.65 LmiExpr Class . . . 651
24.2.66 NlExpr Class . . . 656
24.2.67 nl Namespace . . . 662
24.2.68 CallbackBase Class . . . 671
24.2.69 NlpCallbackBase Class . . . 676
24.2.70 GenConstrX Class . . . 680
24.2.71 CoptError Class . . . 681
24.3 Helper Functions and Utilities . . . 681
24.3.1 Helper Functions . . . 681
24.3.2 tuplelist Class . . . 682
24.3.3 tupledict Class . . . 684
24.3.4 ProbBuffer Class . . . 685

**25 C++ API Reference** **689**
25.1 Constants . . . 689
25.2 Attributes . . . 689
25.3 Information . . . 689
25.4 Parameters . . . 690
25.5 C++ Modeling Classes . . . 690
25.5.1 Envr . . . 690
25.5.2 EnvrConfig . . . 692
25.5.3 Model . . . 693
25.5.4 Var . . . 787
25.5.5 VarArray . . . 789
25.5.6 Expr . . . 790

25.5.7 Constraint . . . 797
25.5.8 ConstrArray . . . 799
25.5.9 ConstrBuilder . . . 800
25.5.10 ConstrBuilderArray . . . 802
25.5.11 Column . . . 803
25.5.12 ColumnArray . . . 805
25.5.13 Sos . . . 806
25.5.14 SosArray . . . 807
25.5.15 SosBuilder . . . 808
25.5.16 SosBuilderArray . . . 810
25.5.17 GenConstr . . . 811
25.5.18 GenConstrArray . . . 812
25.5.19 GenConstrBuilder . . . 813
25.5.20 GenConstrBuilderArray . . . 815
25.5.21 Cone . . . 815
25.5.22 ConeArray . . . 816
25.5.23 ConeBuilder . . . 817
25.5.24 ConeBuilderArray . . . 818
25.5.25 ExpCone . . . 819
25.5.26 ExpConeArray . . . 819
25.5.27 ExpConeBuilder . . . 820
25.5.28 ExpConeBuilderArray . . . 821
25.5.29 AffineCone . . . 822
25.5.30 AffineConeArray . . . 823
25.5.31 AffineConeBuilder . . . 824
25.5.32 AffineConeBuilderArray . . . 826
25.5.33 QuadExpr . . . 827
25.5.34 QConstraint . . . 836
25.5.35 QConstrArray . . . 838
25.5.36 QConstrBuilder . . . 839
25.5.37 QConstrBuilderArray . . . 840
25.5.38 PsdVar . . . 841
25.5.39 PsdVarArray . . . 843
25.5.40 PsdExpr . . . 844
25.5.41 PsdConstraint . . . 852
25.5.42 PsdConstrArray . . . 853
25.5.43 PsdConstrBuilder . . . 854
25.5.44 PsdConstrBuilderArray . . . 856
25.5.45 LmiConstraint . . . 857
25.5.46 LmiConstrArray . . . 858
25.5.47 LmiExpr . . . 859
25.5.48 SymMatrix . . . 865
25.5.49 SymMatrixArray . . . 866
25.5.50 SymMatExpr . . . 867
25.5.51 MVar . . . 871
25.5.52 MConstr . . . 879
25.5.53 MConstrBuilder . . . 886
25.5.54 MExpression . . . 889
25.5.55 MLinExpr . . . 891
25.5.56 MPsdConstr Class . . . 901
25.5.57 MPsdConstrBuilder . . . 907
25.5.58 MPsdExpr Class . . . 910
25.5.59 MQConstr . . . 921
25.5.60 MQConstrBuilder . . . 928
25.5.61 MQExpression . . . 930
25.5.62 MQuadExpr . . . 934
25.5.63 NlExpr Class . . . 945
25.5.64 NlConstraint Class . . . 951

25.5.65 NlConstrArray Class . . . 953
25.5.66 NlConstrBuilder Class . . . 954
25.5.67 NlConstrBuilderArray Class . . . 955
25.5.68 NL Namespace . . . 956
25.5.69 NdArray . . . 963
25.5.70 Shape . . . 970
25.5.71 View . . . 973
25.5.72 CallbackBase . . . 976
25.5.73 NlpCallbackBase . . . 982
25.5.74 ProbBuffer . . . 984

**26 C# API Reference** **985**
26.1 Constants . . . 985
26.1.1 General Constants . . . 985
26.1.2 Attributes . . . 985
26.1.3 Information . . . 985
26.1.4 Callback Information . . . 986
26.1.5 Parameters . . . 986
26.2 C# Modeling Classes . . . 986
26.2.1 Envr . . . 986
26.2.2 EnvrConfig . . . 988
26.2.3 Model . . . 989
26.2.4 Var . . . 1081
26.2.5 VarArray . . . 1083
26.2.6 Expr . . . 1084
26.2.7 Constraint . . . 1089
26.2.8 ConstrArray . . . 1091
26.2.9 ConstrBuilder . . . 1092
26.2.10 ConstrBuilderArray . . . 1093
26.2.11 Column . . . 1094
26.2.12 ColumnArray . . . 1097
26.2.13 Sos . . . 1099
26.2.14 SosArray . . . 1099
26.2.15 SosBuilder . . . 1100
26.2.16 SosBuilderArray . . . 1102
26.2.17 GenConstr . . . 1103
26.2.18 GenConstrArray . . . 1104
26.2.19 GenConstrBuilder . . . 1105
26.2.20 GenConstrBuilderArray . . . 1107
26.2.21 Cone . . . 1108
26.2.22 ConeArray . . . 1108
26.2.23 ConeBuilder . . . 1109
26.2.24 ConeBuilderArray . . . 1110
26.2.25 ExpCone . . . 1111
26.2.26 ExpConeArray . . . 1112
26.2.27 ExpConeBuilder . . . 1113
26.2.28 ExpConeBuilderArray . . . 1114
26.2.29 AffineCone . . . 1115
26.2.30 AffineConeArray . . . 1116
26.2.31 AffineConeBuilder . . . 1117
26.2.32 AffineConeBuilderArray . . . 1119
26.2.33 QuadExpr . . . 1120
26.2.34 QConstraint . . . 1127
26.2.35 QConstrArray . . . 1129
26.2.36 QConstrBuilder . . . 1130
26.2.37 QConstrBuilderArray . . . 1131
26.2.38 PsdVar . . . 1132
26.2.39 PsdVarArray . . . 1135

26.2.40 PsdExpr . . . 1136
26.2.41 PsdConstraint . . . 1143
26.2.42 PsdConstrArray . . . 1145
26.2.43 PsdConstrBuilder . . . 1146
26.2.44 PsdConstrBuilderArray . . . 1147
26.2.45 LmiConstraint . . . 1148
26.2.46 LmiConstrArray . . . 1150
26.2.47 LmiExpr . . . 1151
26.2.48 SymMatrix . . . 1155
26.2.49 SymMatrixArray . . . 1156
26.2.50 SymMatExpr . . . 1157
26.2.51 MVar . . . 1161
26.2.52 MConstr . . . 1169
26.2.53 MConstrBuilder . . . 1175
26.2.54 MExpression . . . 1178
26.2.55 MLinExpr . . . 1180
26.2.56 MPsdConstr . . . 1190
26.2.57 MPsdConstrBuilder . . . 1196
26.2.58 MPsdExpr . . . 1199
26.2.59 MQConstr . . . 1211
26.2.60 MQConstrBuilder . . . 1217
26.2.61 MQExpression . . . 1220
26.2.62 MQuadExpr . . . 1224
26.2.63 NdArray . . . 1236
26.2.64 Shape . . . 1242
26.2.65 View . . . 1245
26.2.66 NlExpr Class . . . 1247
26.2.67 NlConstraint Class . . . 1251
26.2.68 NlConstrArray Class . . . 1253
26.2.69 NlConstrBuilder Class . . . 1254
26.2.70 NlConstrBuilderArray Class . . . 1255
26.2.71 NL Namespace . . . 1256
26.2.72 CallbackBase . . . 1263
26.2.73 ProbBuffer . . . 1270
26.2.74 CoptException . . . 1271

**27 Java API Reference** **1273**
27.1 Constants . . . 1273
27.1.1 General Constants . . . 1273
27.1.2 Attributes . . . 1273
27.1.3 Information . . . 1273
27.1.4 Callback Information . . . 1274
27.1.5 Parameters . . . 1274
27.2 Java Modeling Classes . . . 1274
27.2.1 Envr . . . 1274
27.2.2 EnvrConfig . . . 1276
27.2.3 Model . . . 1277
27.2.4 Var . . . 1362
27.2.5 VarArray . . . 1365
27.2.6 Expr . . . 1366
27.2.7 Constraint . . . 1371
27.2.8 ConstrArray . . . 1374
27.2.9 ConstrBuilder . . . 1375
27.2.10 ConstrBuilderArray . . . 1376
27.2.11 Column . . . 1377
27.2.12 ColumnArray . . . 1380
27.2.13 Sos . . . 1381
27.2.14 SosArray . . . 1382

27.2.15 SosBuilder . . . 1383
27.2.16 SosBuilderArray . . . 1385
27.2.17 GenConstr . . . 1386
27.2.18 GenConstrArray . . . 1387
27.2.19 GenConstrBuilder . . . 1388
27.2.20 GenConstrBuilderArray . . . 1390
27.2.21 Cone . . . 1391
27.2.22 ConeArray . . . 1391
27.2.23 ConeBuilder . . . 1392
27.2.24 ConeBuilderArray . . . 1394
27.2.25 ExpCone . . . 1395
27.2.26 ExpConeArray . . . 1395
27.2.27 ExpConeBuilder . . . 1396
27.2.28 ExpConeBuilderArray . . . 1397
27.2.29 AffineCone Class . . . 1398
27.2.30 AffineConeArray Class . . . 1399
27.2.31 AffineConeBuilder Class . . . 1400
27.2.32 AffineConeBuilderArray Class . . . 1402
27.2.33 QuadExpr . . . 1403
27.2.34 QConstraint . . . 1412
27.2.35 QConstrArray . . . 1414
27.2.36 QConstrBuilder . . . 1415
27.2.37 QConstrBuilderArray . . . 1416
27.2.38 PsdVar . . . 1417
27.2.39 PsdVarArray . . . 1418
27.2.40 PsdExpr . . . 1419
27.2.41 PsdConstraint . . . 1427
27.2.42 PsdConstrArray . . . 1428
27.2.43 PsdConstrBuilder . . . 1429
27.2.44 PsdConstrBuilderArray . . . 1431
27.2.45 LmiConstraint . . . 1432
27.2.46 LmiConstrArray . . . 1433
27.2.47 LmiExpr . . . 1434
27.2.48 SymMatrix . . . 1439
27.2.49 SymMatrixArray . . . 1440
27.2.50 SymMatExpr . . . 1441
27.2.51 NlExpr Class . . . 1445
27.2.52 NlConstraint Class . . . 1451
27.2.53 NlConstrArray Class . . . 1452
27.2.54 NlConstrBuilder Class . . . 1453
27.2.55 NlConstrBuilderArray Class . . . 1455
27.2.56 NL Namespace . . . 1456
27.2.57 CallbackBase . . . 1462
27.2.58 ProbBuffer . . . 1469
27.2.59 CoptException . . . 1470

# Chapter 1

# Introduction to Cardinal Optimizer

Cardinal Optimizer is a high-performance mathematical programming solver for efficiently solving large-scale optimization problems. This documentation provides a basic introduction to the Cardinal Optimizer, including:

- *How to install Cardinal Optimizer*
- *How to setup license files*
- *How to use Cardinal Optimizer in interactive shell*

We suggest that all users read the first two sections carefully before using the Cardinal Optimizer.

Once the installation and license setup are done, we recommend that users who want to do a quick experiment with the Cardinal Optimizer read the *COPT Interactive Shell* chapter for details. Users who already have a preferred programming language can select from available `Application Programming Interfaces (APIs)`, including:

- *C interface*
- *C++ interface*
- *C# interface*
- *Java interface*
- *Python interface*
- *AMPL interface*
- *Pyomo interface*
- *PuLP interface*
- *CVXPY interface*

## 1.1 Overview

Cardinal Optimizer supports solving Linear Programming (LP) problems, Second-Order-Cone Programming (SOCP) problems, Quadratic Programming (QP) problems, Quadratically Constrained Programming (QCP) problems, Exponential Cone Programming (ExpCone) problems, Semidefinite Programming problems (SDP), General Nonlinear Programming (NLP) problems, and Mixed Integer Programming (MIP) problems, which include Mixed Integer Linear Programming (MILP), Mixed Integer Second-Order-Cone Programming (MISOCP), Mixed Integer Convex Quadratic Programming (MIQP), Mixed Integer Convex Quadratically Constrained Programming (MIQCP).

We will support more problem types in the future. The supported problem types and available algorithms are summarized in Table 1.1.

Table 1.1: Supported problem types and available algorithms

| **Problem type** | Available algorithms |
|---|---|
| Linear Programming (LP) | Simplex, Barrier (CPU/GPU), First-order Method (PDLP)(CPU/GPU) |
| Second-Order-Cone Programming (SOCP) | Barrier (CPU/GPU) |
| Exponential Cone Programming (ExpCone) | Barrier (CPU/GPU) |
| Convex Quadratic Programming (QP) | Barrier (CPU/GPU) |
| Convex Quadratically Constrained Programming (QCP) | Barrier (CPU/GPU) |
| Semidefinite Programming (SDP) | Barrier (CPU/GPU), ADMM |
| Nonconvex Quadratic Programming (QP) | Spatial Branch-and-Bound (global optimum), Barrier (local optimum) |
| Nonconvex Quadratically Constrained Programming (QCP) | Spatial Branch-and-Bound (global optimum), Barrier (local optimum) |
| General Nonlinear Programming (NLP) | Barrier (local optimum) |
| Mixed Integer Linear Programming (MILP) | Branch-and-Cut |
| Mixed Integer Second-Order-Cone Programming (MISOCP) | Branch-and-Cut |
| Mixed Integer Convex/Nonconvex Quadratic Programming (MIQP) | Branch-and-Cut |
| Mixed Integer Convex/Nonconvex Quadratically Constrained Programming (MIQCP) | Branch-and-Cut |

Cardinal Optimizer supports all major 64-bit operating systems including Windows, Linux (including ARM64 platform) and MacOS (including ARM64 platform), and currently provides programming interfaces shown below:

- C interface
- C++ interface
- C# interface
- Java interface
- Python interface
- AMPL interface
- AIMMS interface
- Pyomo interface
- PuLP interface
- CVXPY interface
- GAMS interface
- Julia interface

We are going to develop more programming interfaces to suit various needs of users and situations.

## 1.2 Licenses

Now, we provide 4 types of licenses, which are Personal License, Server License, Floating License, and Cluster License. They are listed in the table below (Table 1.2) :

Table 1.2: License Type

| License Type | Detail |
|---|---|
| Personal License | It is tied to personal computers by username. Only approved users can run COPT on their devices. No limitations on CPU cores and threads. |
| Server License | It is tied to a single server computer by its hardware info (MAC and CPUID). An arbitrary number of users and programs can run COPT simultaneously. No limitations on CPU cores as well. |
| Floating License | It is tied to a server machine running COPT floating token service, by its hardware info (MAC and CPUID). Any COPT floating client connected to server can borrow and use the floating license, and thus run one process for optimization jobs simultaneously. The token number is the maximum number of clients who can use floating licenses simultaneously. |
| Cluster License | It is tied to a server machine running COPT compute cluster service, by its hardware info (MAC and CPUID). Any COPT compute cluster client connected to server can offload optimization computations. That is, clients are allowed to do modeling locally, execute optimization jobs remotely, and then obtain results interactively. Although the server can have multiple clients connected, each connection must run optimization jobs sequentially. No limitations on CPU cores. |

## 1.3 How to Cite

If you used COPT in your research work, please mention us in your publication. For example:

- We used COPT [1] in our project.
- To solve the integer problem, we used Cardinal Optimizer [1].

with the following entry in the Reference section:

```
[1] D. Ge, Q. Huangfu, Z. Wang, J. Wu and Y. Ye. Cardinal Optimizer (COPT) user guide.
↪ https://guide.coap.online/copt/en-doc, 2023.
```

The corresponding BiBTeX citation is:

```
@misc{copt,
  author={Dongdong Ge and Qi Huangfu and Zizhuo Wang and Jian Wu and Yinyu Ye},
  title={Cardinal {O}ptimizer {(COPT)} user guide},
  howpublished={https://guide.coap.online/copt/en-doc},
  year=2023
}
```

## 1.4 Contact Information

Cardinal Optimizer is developed by Cardinal Operations, users who want any further help can contact us using information provided in Table 1.3.

Table 1.3: Contact information

| Type | Information | Description |
|---|---|---|
| Website | https://www.shanshu.ai/ | |
| Phone | 400-680-5680 | |
| Email | coptsales@shanshu.ai | business support |
| Email | coptsupport@shanshu.ai | technical support |

# Chapter 2

# Installation Guide

This chapter introduces how to install **Cardinal Optimizer** on all supported operating systems, and how to obtain and setup license correctly. We recommend all users ready this chapter carefully before using Cardinal Optimizer.

## 2.1 Registration

Before using Cardinal Optimizer, users need to registrate online and then install the COPT package on your machine. If this is not done yet, please visit official COPT home page and fill the registration form following the guidelines.

The online registration is for personal license application. Specifically, users only need to provide username of machine, besides basic information.

Upon approval, you will receive a letter from coptsales@shanshu.ai. It gives both link to download COPT software package, a license key tied with registration information, and also two attached license files. You may refer to *Software Installation* below and *Setting Up License* for further steps.

If you encountered any problems, please contact coptsupport@shanshu.ai for help.

## 2.2 Software Installation

### 2.2.1 Windows

We provide two types of installation packages for Windows operating systems, The user needs to **choose one of the two options**. One is an executable installer (Download link name contains `installer`) for most of users and the other one is a zip-format archive specialized for expert users.

The executable installer provides a visual installation prompt window and automatically configures environment variables. Users only need to follow the instructions and click to complete the installation steps in sequence; The ZIP format installation package requires the user to decompress the installation package first and manually configure the environment variables.

- Executable installer: CardinalOptimizer-8.0.7-win64-installer.zip
- ZIP-format installation package: CardinalOptimizer-8.0.7-win64.zip

**We recommend that users download the executable installer.**

### Executable installer

If you download the executable installer for Windows from our website, e.g. CardinalOptimizer-8.0.7-win64-installer.exe for 64-bit version of COPT 8.0.7, just double-click it and follow the following guidance:

- Step 1: Click the installer and select the installation language. The default installation language is `English`, users can change it by select from the drop-down menu, see Fig. 2.1. Here we use the default setting.

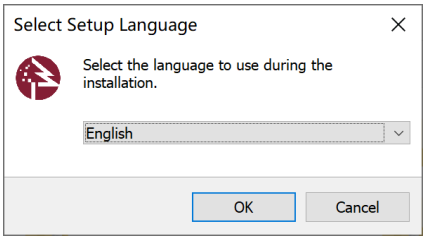



Fig. 2.1: Select installation language

- Step 2: If you agree with the `'End-User License Agreement (EULA)'`, just choose `'I accept the agreement'` and then click `'Next'`. The software won't install if you disagree with the `EULA`, see Fig. 2.2.

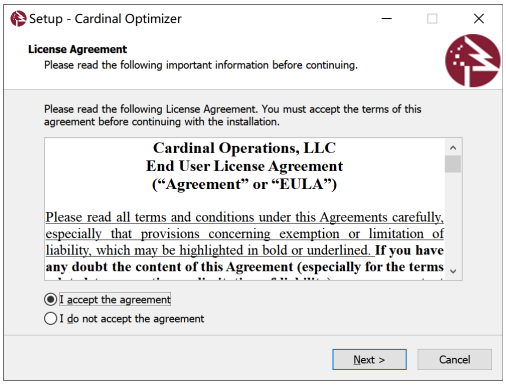



Fig. 2.2: License agreement page

- Step 3: By default, the installer will place all files into directory C:\Program Files\copt80, you may change it to any directories. If you have decided the install directory, just click `'Next'`, see Fig. 2.3.

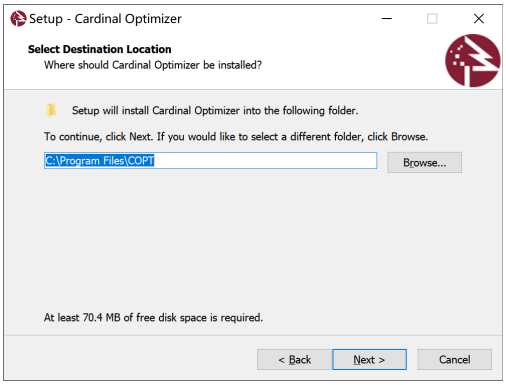



Fig. 2.3: Choose install directory

- Step 4: To select the start menu folder, you can simply use the default setting and click `'Next'`, see Fig. 2.4.

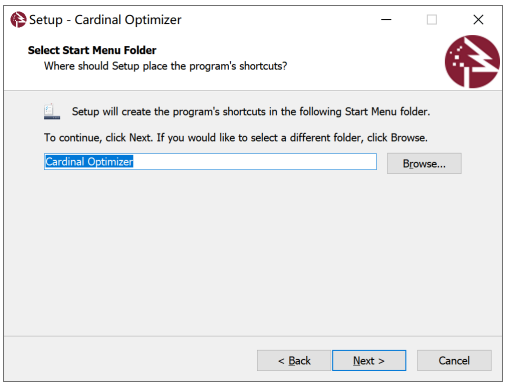



Fig. 2.4: Select start menu folder

- Step 5: By now, the software is ready to install, just click `'Install'`, see Fig. 2.5.

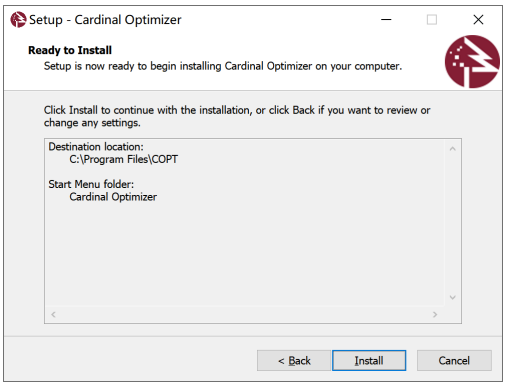



Fig. 2.5: Ready to install

- Step 6: When installation completed, the software requires restart of your machine since the installer has automatically made the required modifications to environment variables for you. Be sure to save your working files and close other running applications before restart, and then click `'Finish'`, see Fig. 2.6.




Fig. 2.6: Installation completed and restart your machine

#### Zip-format installer archive

If you downloaded the zip-format installer archive, just uncompress it to any directories with any unarchiver and set up environment variables as follows. Here we assumed the installation directory to be C:\Program Files\copt80:

- Step 1: Open `Command Prompt (cmd)` with **administrator privilege** and execute the following command to pop-up the environment variables setting panel.

```
rundll32 sysdm.cpl,EditEnvironmentVariables
```

- Step 2: Add directory C:\Program Files\copt80\bin to system environment variable `PATH`.
- Step 3: Create a new system environment variable named `COPT_HOME`, whose value is C:\Program Files\copt80.
- Step 4: Create a new system environment variable named `COPT_LICENSE_DIR`, whose value is C:\Program Files\copt80.

Up to now, you have already setup the required modifications to the environment variables. If you accept the `copt-eula_en.pdf` in installation directory, then please go to *Setting Up License* for license issues.

### 2.2.2 MacOS

We provide two types of installation packages for MacOS, one is a DMG-format installer for most of users and the other is a gzip-format archive for expert users. The user needs to **choose one of the two options**.

In addition, for MacOS systems, we provide the MacOS-Universal installation package, which is universal to both Apple Silicon and old Intel chips (the installation package suffix is `universal_mac`).

- DMG-format installer: CardinalOptimizer-8.0.7-universal_mac.dmg
- Gzip-format Archive: CardinalOptimizer-8.0.7-universal_mac.tar.gz

**We recommend that users download the DMG-format installer.**

#### DMG installer

If you download the DMG-format installer for MacOS-Universal from our website, e.g. CardinalOptimizer-8.0.7-universal_mac.dmg of COPT 8.0.7, please follow the following guidance to install the software:

- Step 1: Double-click the DMG-format installer, waiting for the OS to mount the DMG installer automatically.
- Step 2: Simply drag the copt80 folder into `'Applications'` folder, see Fig. 2.7:

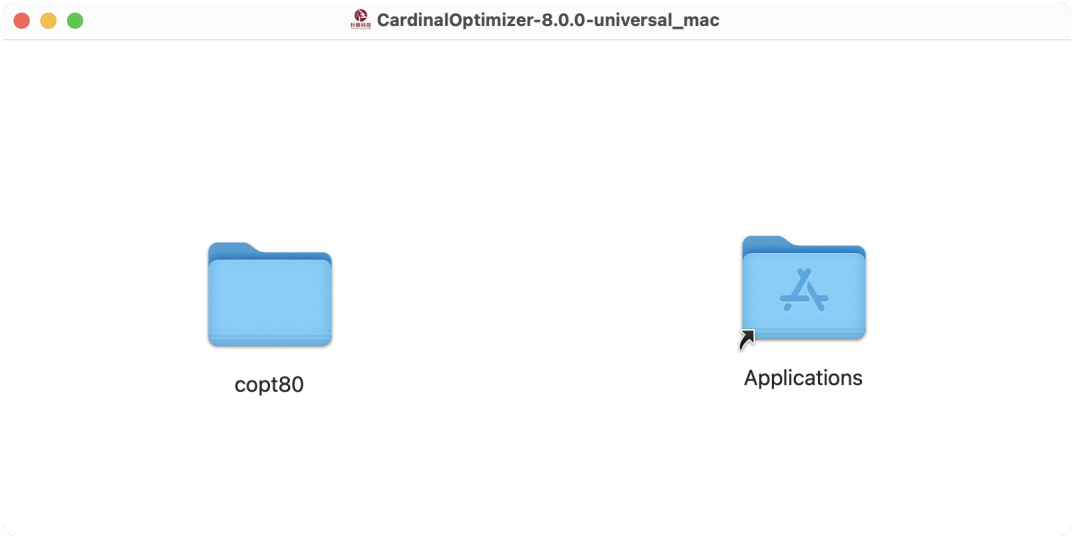



Fig. 2.7: Drag copt80 into `'Applications'`

After the above steps are completed, the necessary environment variables need to be configured. Please continue to read: *Environment variable configuration* section.

### GZIP installer

If you download the gzip-format installer archive, just uncompress it to any directories with commands:

```
tar -xzf CardinalOptimizer-8.0.7-universal_mac.tar.gz
```

You will get a folder named copt80 in current directory, you can move it to any other directories as you want. We recommend users move it to the `'Applications'` folder by executing commands below:

```
mv copt80 /Applications/
```

After the above steps are completed, the necessary environment variables need to be configured. Please continue to read: *Environment variable configuration* section.

### Environment variable configuration

The next step for both DMG-format and gzip-format installer is to setup the environment variables.

First, please execute the following command on the terminal to determine the current Shell version:

```
echo $SHELL
```

If the output is: `'/bin/bash'`, it means that the current terminal is `bash`; if the output is: `'/bin/zsh'`, It means that the current terminal is `zsh`.

Next, please open the terminal and confirm that you are in the user directory (if not, you need to execute the following command to switch to the user directory):

```
cd ~/
```

Please refer to the corresponding environment variable configuration guide according to the terminal.

- *BASH terminal*
- *ZSH terminal*

After the environment variable is configured, you can verify whether the configuration is successful, refer to *Verify the environment variable*.

### BASH terminal

There are three main steps to configure environment variables under the BASH terminal:

#### 1. Check 'bash_profile' file

Enter the following command to output all files in the user directory and check whether there is a `'.bash_profile'` hidden file:

```
ls -a
```

If the file does not exist, execute the following command to create an empty `'.bash_profile'` file at your own (if the file already exists, please ignore this step):

```
touch ~/.bash_profile
```

#### 2. Add environment variables

First, please execute the following command to open the `'.bash_profile'` file.

```
open ~/.bash_profile
```

Next, please add the following content to the `'.bash_profile'` using any editors that you prefer:

```
export COPT_HOME=/Applications/copt80

export COPT_LICENSE_DIR=/Applications/copt80

export PATH=$COPT_HOME/bin:$PATH

export DYLD_LIBRARY_PATH=$COPT_HOME/lib:$DYLD_LIBRARY_PATH
```

**Note:** There can be no spaces on both sides of the equal sign.

#### 3. Check the environment variables to take effect

Save the above modification and exit, the user can execute the following command in the terminal to view the modified `'.bash_profile'` file:

```
cat ~/.bash_profile
```

If the modification is successful, the content of the file output should include the newly added information above.

Then, the user needs to execute the following command on the terminal to make the above modification take effect.

```
source ~/.bash_profile
```

### ZSH terminal

Likewise, there are three main steps to configure environment variables under the ZSH terminal:

#### 1. Check 'zshrc' file

Enter the following command to output all files in the user directory and check whether there is a `'.zshrc'` hidden file:

```
ls -a
```

If the file does not exist, execute the following command to create an empty `'.zshrc'` file at your own (if the file already exists, please ignore this step):

```
touch ~/.zshrc
```

**2. Add environment variables**

First, please execute the following command to open the `'.zshrc'` file.

```
open ~/.zshrc
```

Next, please add the following content to the `'.zshrc'` using any editors that you prefer:

```
export COPT_HOME=/Applications/copt80

export COPT_LICENSE_DIR=/Applications/copt80

export PATH=$COPT_HOME/bin:$PATH

export DYLD_LIBRARY_PATH=$COPT_HOME/lib:$DYLD_LIBRARY_PATH
```

**3. Check the environment variables to take effect**

Save the above modification and exit, the user can execute the following command in the terminal to view the modified `'.zshrc'` file:

```
cat ~/.zshrc
```

If the modification is successful, the content of the file output should include the newly added information above.

Then, the user needs to execute the following command on the terminal to make the above modification take effect.

```
source ~/.zshrc
```

#### Verify environment variable configuration

Now open a new terminal to check if the previous modifications work by executing commands below respectively:

```
echo $COPT_HOME
echo $COPT_LICENSE_DIR

echo $PATH
echo $DYLD_LIBRARY_PATH
```

If the terminal outputs respectively:

```
/Applications/copt80
```

```
/Applications/copt80
```

```
/Applications/copt80/bin:$PATH
```

```
/Applications/copt80/lib:$DYLD_LIBRARY_PATH
```

It means that the COPT-related environment variables are configured successfully.

**Notes:** The environment variables `$PATH` and `$DYLD_LIBRARY_PATH` may display different contents on different computers. However, the set COPT-related environment variables should be displayed normally to indicate that the configuration is successful.

If the user checks that the COPT-related environment variables have been successfully added, please carefully read the user agreement document `'copt-eula_cn.pdf'` in the installation directory. If you accept the agreement, please go to *Setting Up License* for license issues.

#### MacOS Security Checkup

For MacOS 10.15 (Catalina), if users received error message below:

```
"libcopt.dylib" cannot be opened because the developer cannot be verified.
macOS cannot verify that this app is free from malware.
```

or

```
"libcopt_cpp.dylib": dlopen(libcopt_cpp.dylib,6): no suitable image found.
Did find: libcopt_cpp.dylib: code signature in (libcopt_cpp.dylib) not valid
for use in process using Library Validation: library load disallowed by
system policy.
```

Then execute the following commands:

```
xattr -d com.apple.quarantine CardinalOptimizer-8.0.7-universal_mac.dmg
xattr -d com.apple.quarantine CardinalOptimizer-8.0.7-universal_mac.tar.gz
```

or

```
xattr -dr com.apple.quarantine /Applications/copt80
```

to disable security check of MacOS.

### 2.2.3 Linux

For Linux platform, we support X86 and ARM64 architectures and provide packages in `GZIP` format. The installation package names differ from different architectures:

- Linux-X86: CardinalOptimizer-8.0.7-lnx64.tar.gz
- Linux-ARM64: CardinalOptimizer-8.0.7-aarch64_lnx.tar.gz

If you download the software, e.g. CardinalOptimizer-8.0.7-lnx64.tar.gz for 64-bit version of COPT 8.0.7, just type commands below in shell to extract it to any directories:

```
tar -xzf CardinalOptimizer-8.0.7-lnx64.tar.gz
```

You will get a folder named copt80 in current directory, you can move it to any other directories as you like. We recommend users move it to `'/opt'` directory by typing commands below in shell:

```
sudo mv copt80 /opt/
```

Note that the above command requires **root privilege** to execute.

The next step users need to do is to set the required environment variables, by adding the following commands to the `'.bashrc'` file in your `$HOME` directory using any editors that you prefer:

```
export COPT_HOME=/opt/copt80

export COPT_LICENSE_DIR=/opt/copt80

export PATH=$COPT_HOME/bin:$PATH

export LD_LIBRARY_PATH=$COPT_HOME/lib:$LD_LIBRARY_PATH
```

Remember to save your modifications to the `'.bashrc'` file, and open a new terminal to check if it works by executing commands below respectively:

```
echo $COPT_HOME
echo $COPT_LICENSE_DIR

echo $PATH
echo $LD_LIBRARY_PATH
```

If you accept the `copt-eula_en.pdf` in installation directory, then please go to *Setting Up License* for license issues.

## 2.3 Setting Up License

The Cardinal Optimizer requires a valid license to work properly. We offer different types of licenses most suitable for user's needs. All users should read this section carefully. If you encounter any problem about license, feel free to contact coptsupport@shanshu.ai.

The license specifically includes: `license.dat` and `license.key` two files. Starting from COPT 6.5, if you apply for personal license applications via our official website, we will send both files directly as attachments (no need to obtain it yourself).

Users can **directly download two license files to the local computer, skip the following steps of obtaining and verifying the license** , and can go straight to the *Install license* step.

The configuration of the license mainly includes the following three steps:

- *Obtaining License (skippable)*
- *Verifying License (skippable)*
- *Installing License*

### 2.3.1 Obtaining License

Once the registration is done, a license key is sent to users. It is a unique token binding with user's registration information. Afterwards, users may run the `copt_licgen` tool, shipped with Cardinal Optimizer, to obtain license files from COPT licensing server (Internet connection required) .

**Note**: If the user has already obtained the two files `license.dat` and `license.key`, there is no need to repeat the following steps to obtain them again.

The following notes show you how to play with the `copt_licgen` tool.

For Windows system, open a new cmd window. The current path is the user directory, and the path is as follows: `"C:\Users\copt80"`.

For MacOS and Linux systems, open a new terminal. The current path is the user directory, represented by the symbol `~`.

To obtain COPT license files, execute the following command using the option `'-key'` and the license key as argument. Below is an example, assuming the license key is `'19200817f147gd9f60abc791def047fb'`:

```
copt_licgen -key 19200817f147gd9f60abc791def047fb
```

If the license key is saved to `key.txt` file in format of `'KEY=xxx'`, which resides in the same place as `copt_licgen`, execute the following command using the option `'-file'` and `'key.txt'` as argument.

```
copt_licgen -file key.txt
```

We recommend to use the **first way** to get your license files.

If the authorization server verifies the license, then it generates `license.dat` and `license.key` and download it to the user's computer. The default download directory is the current working directory.

```
copt_licgen -key 19200817f147gd9f60abc791def047fb
  [Info] Cardinal Optimizer   COPT v8.0.7 20240304
  [Info] Use specific key 19200817f147gd9f60abc791def047fb
  [Info] * get new COPT license from licensing server *
  [Info] Write to license.dat
  [Info] Write to license.key
  [Info] Received new license files from server
  [Info] Done !!!
```

**Note:** Users do not need to have internet connection to run COPT. However, obtaining license itself requires internet connection. If you encounter any problem, please feel free to contact coptsupport@shanshu.ai.

### 2.3.2 Verifying License

If the license key binding to registration information is verified by COPT license server, two license files, `license.dat` and `license.key`, are downloaded to the current working directory. To double check two license files are valid in current version of COPT, execute the following command with the option `'-v'`. Note that this command requires both `license.dat` and `license.key` existing in the current working directory.

```
copt_licgen -v
```

If you see log information similar to the following, you have obtained and verified the license files successfully.

```
copt_licgen -v
  [Info] Cardinal Optimizer   COPT v8.0.7 20240304
  [Info] Run local validation
  [Info] Read license.dat
  [Info] Read license.key
  [Info] Expiry : Tue 2030-12-31 00:00:00 +0800
  [Info] Local validation result: Succeeded
  [Info] Done !!!
```

### 2.3.3 Installing License

Once you obtained the `license.dat` and `license.key` files from COPT license server and verified they work as expected, setting up them is just as simple as moving them to the same directory as the COPT dynamic library or *COPT Command-Line*. Note this installation is applied only to current COPT.

The following two ways of installation applied to all versions of COPT on your machine.

#### Via HOME directory method (recommended)

The simplest way to install COPT license is to create a folder of `copt` in your `HOME` directory, and move the authenticated license files `license.dat` and `license.key` to the new folder `copt`.

**Note:** The folder name `"copt"` is case sensitive and it must be lowercase.

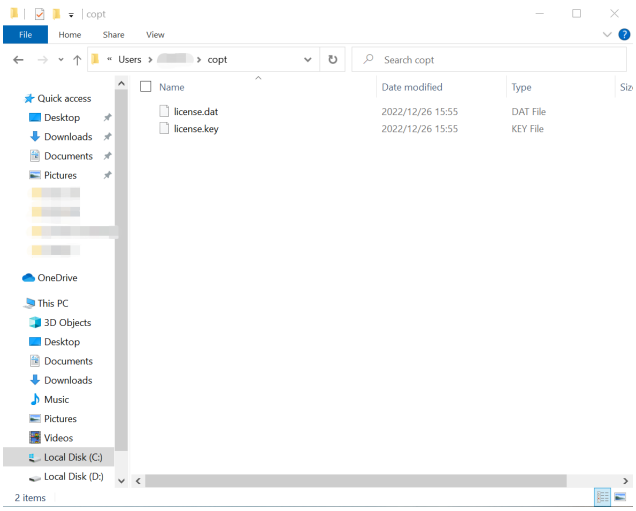
The `HOME` directory names are slightly different under different systems, please refer to the corresponding license installation steps according to your operating system:

- *Windows*
- *MacOS*
- *Linux*

**Windows**

The `HOME` directory looks like `"C:\Users\username\"` on Windows. User can manually move the two license files `license.dat` and `license.key` to the directory `"C:\Users\username\copt"`.

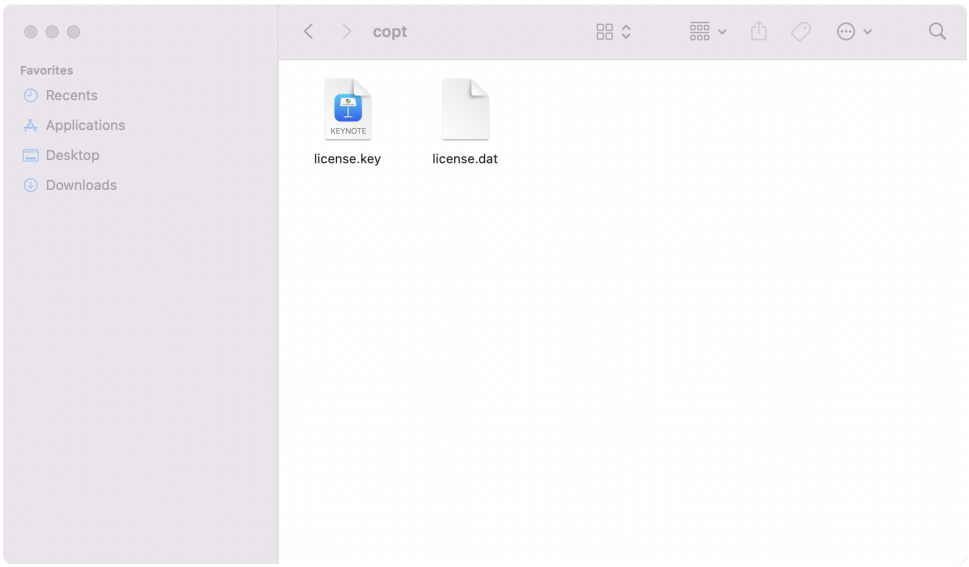
Finally, please check whether the license files `license.dat` and `license.key` are located in this directory, as shown in the figure below, indicating that the license has been installed correctly:




**MacOS**

The `HOME` directory looks like `"/home/your username/"` on MacOS. User can manually move the two license files to `"/home/your username/copt/"`.

Finally, please check whether the license files `license.dat` and `license.key` are located in this directory, as shown in the figure below, indicating that the license has been installed correctly:




### Linux

The `HOME` directory looks like `"/home/your username/"` on Linux.

User can execute the following command in the terminal to move the two license files `license.dat` and `license.key` to the directory `"/home/your username/copt"`.

```
mv license.*  ~/copt/
```

Next, please check that the license files `license.dat` and `license.key` are located in `"/home/username/copt"` directory, the command is:

```
ls ~/copt/
```

If the terminal output shows that there are two files `license.dat` and `license.key`, it means that the license has been installed correctly, otherwise the installation fails.

## Via environment variable method

Alternatively, for users who prefer having licenses in a customized folder, they can set environment variable `COPT_LICENSE_DIR` to the customized folder. You may refer to *Software Installation* for how to set environment variable on Windows, Linux and MacOS.

In addition, please double check that if license files `license.dat` and `license.key` locate in path specified by environmental variable `COPT_LICENSE_DIR`.

The viewing and operation of environment variables are slightly different under different systems, please refer to the following installation steps for your own operating system:

- *Windows*
- *MacOS*
- *Linux*

### Windows

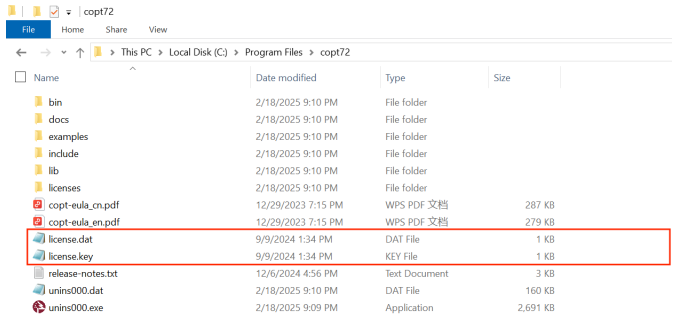
For Windows systems, execute the following command to view the path pointed to by the environment variable `COPT_LICENSE_DIR` :

```
echo %COPT_LICENSE_DIR%
```

**Note:** If there is no output on the terminal, it means that the COPT-related environment variables have not been configured, please check *Software installation*.

Then move the license files `license.dat` and `license.key` to the path `COPT_LICENSE_DIR` points to.

Here we assume that the environment variable `COPT_LICENSE_DIR` points to the default installation directory of COPT. As shown in the figure, it means that the license has been installed correctly:



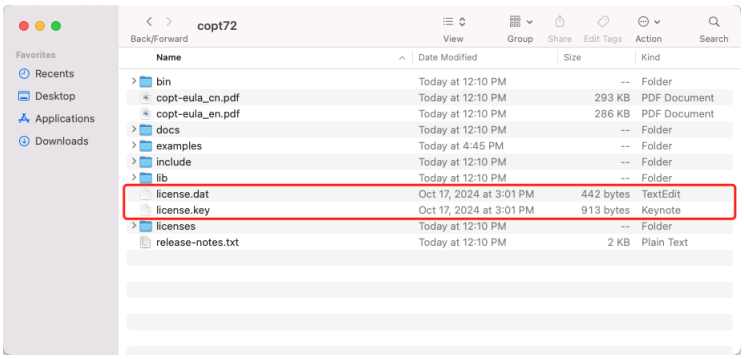

### MacOS

For MacOS system, you can use the following command to view the path pointed to by the environment variable `COPT_LICENSE_DIR`:

```
echo $COPT_LICENSE_DIR
```

Then move the license files `license.dat` and `license.key` to the path `COPT_LICENSE_DIR` points to.

Here we assume that the environment variable `COPT_LICENSE_DIR` points to the default installation directory of COPT. As shown in the figure, it means that the license has been installed correctly:




### Linux

For Linux systems, user can use the following command to view the path pointed to by the environment variable `COPT_LICENSE_DIR`:

```
echo $COPT_LICENSE_DIR
```

Then user can execute the following command to move the license files `license.dat` and `license.key` to the path `COPT_LICENSE_DIR` points to.

```
mv license.* $COPT_LICENSE_DIR/
```

**Note:** For Linux systems, if the path pointed to by the environment variable `COPT_LICENSE_DIR` is /opt/copt80, users need to perform the move operation with **root authority** , and the command is:

```
sudo mv license.* $COPT_LICENSE_DIR/
```

In addition, please double check that if license files `license.dat` and `license.key` locate in path specified by environmental variable `COPT_LICENSE_DIR` , and the command is:

```
ls $COPT_LICENSE_DIR/
```

If the terminal displays the files `license.dat` and `license.key`, it means that the license has been installed correctly, otherwise the license installation fails.

**Note:** If there are license files `license.dat` and `license.key` in the `copt` folder of the user directory and the directory pointed to by the environment variable, the former will be checked and used the former.

### 2.3.4 Others

Basically, each type of licenses includes two license files: `license.dat` and `license.key`, each of which has digital signature to protect its content. When invoking COPT command-line or loading COPT dynamic library to solve an optimization problem, the public RSA key stored in `'license.key'` is used to verify signature in `'license.dat'`.

Afterwards, license data in format of key-value pair is parsed to verify whether it is a legal license. Below is a sample of license data.

```
## COPT LICENSE FILE ##

USER = Trial User
MAC = 44:05:99:31:41:C2
CPUID = BFEBFBFF000706E5
EXPIRY = 2030-12-31
VERSION = 8.0.7
NOTE = Free For Trial
```

Note: Please make sure that the `VERSION` field in the license file is the same major version as the software installed on the machine. Otherwise, it may cause error: `Invalid signature in public key file` .

## 2.4 Verify the installation and configuration

After the user completes the above steps of software installation and license configuration, please enter the COPT command line tool to verify whether the installation is successful.

In MacOS and Linux systems, please open the terminal (in Windows system, please open the cmd window), and enter the `copt_cmd` command to use the COPT command line tool.

If the user sees the output as shown below (without any error message), the COPT command line tool can be entered normally, which means that the previous software and license configuration has been successfully completed.

```
Cardinal Optimizer v8.0.0. Build date Oct 17 2024
Copyright Cardinal Operations 2024. All Rights Reserved

COPT>
```

If you fail to enter the COPT command line tool normally, please refer to *FAQ* according to the error message and check whether the above steps of installation and license configuration are completed correctly.

After the installation is successfully completed, according to the interface of COPT as needed, please continue to refer to *Quick Start*, and check the installation and applying methods of each interface.

## 2.5 Upgrade

The upgrade steps of COPT mainly include two parts: 1. Software upgrade; 2. License file update.

**1. Uninstall the old version of COPT**

Firstly, if the user does not need to run multiple versions at the same time, the old version of the software can be uninstalled first.

For different operating systems, the methods of uninstalling COPT are slightly different:

In Windows system, please open the old COPT installation package (named like copt80 ), please double-click to run `unins000.exe` , and then the software can be uninstalled automatically. In MacOS and Linux systems, please delete the old version of the COPT installation package directly to complete the software uninstallation.

**2. Install the new version of COPT**

If it is an upgrade between major versions (for example: COPT 6.5 -> COPT 7.0), the user needs to re-apply for new version license. Next, the user needs to reinstall the new version of COPT. For the same steps, please refer to *Software Installation*.

**3. Update the license files**

In addition, after the software update is complete, the license file needs to be updated. We recommend that the user configure the license file via *user directory*. The user only needs to delete the license file of the old version under the copt folder and move the new license file into it to complete the license update.

After completing the above steps, users can refer to *Verify installation and configuration completion* to observe whether the version number in the banner information output by the COPT command line tool is a new version.

**Notes**

1. For the Python interface of COPT, users also need to **upgrade its corresponding interface `coptpy` at the same time**. If Python is an Anaconda distribution, it can be conducted via `pip install --upgrade coptpy` .
2. The above are the upgrade steps for major version iterations (for example: COPT 6.5 -> COPT 7.0). In addition, on the basis of each major version, COPT will also provide patch release (eg: COPT 8.0.1 -> COPT 8.0.2), for some feature fixes and updates. For upgrades between minor versions, there is no need to update the license files.

# Chapter 3

# COPT Command-Line

The **Cardinal Optimizer** ships with `copt_cmd` executable on all supported platforms, which let users solve optimization models in an interactive way. Before running COPT command-line, please make sure that you have valid license installed.

## 3.1 Overview

COPT command-line is a COPT API interpreter that executes commands read from the standard input or from a script file. COPT command-line interprets the following options when it is invoked:

- `-c`: If the `'-c'` option is present, it reads from an inline scripts, which is a quoted string and specified by the second argument.
- `-i`: If the `'-i'` option is present, it reads from an input script file, whose path is specified by the second argument.
- `-o`: If the `'-o'` option is present, it reads from standard input, while each **valid** command line is written to an output script file, whose path is specified by the second argument.

Regardless of arguments, the tool is interactive. Besides wrapping COPT API calls, it offers various features to help users move cursor around and edit lines. We try to provide as much user experience as standard command prompt (Windows console and Unix terminal).

## 3.2 Edit mode

This tool defines a number of commands to position the cursor, edit lines with combination keys on a standard keyboard. The following notes show you how to use the most important ones.

- **Basic commands**
  1. `<Insert>`: Toggle between inserting characters and replacing the existing ones.
  2. `<Esc>`: Discard inputs and move the cursor to the beginning of line. Press `<ESC>` twice on Linux/Mac platform to do the same thing.
- **Moving around**
  1. `<Home>/<End>`: Jump to the beginning/end of line.
  2. `<Left>/<Right>` Arrow: Move the cursor one character to the left/right.
  3. `<CTRL>+<Left>/<Right>` Arrow: Move the cursor one word to the left/right.
- **Cut and Paste**

  An internal paste buffer is available for the following cut operations.

1. <Delete>: Cut the character under the cursor.
2. <Backspace>: Cut the character before the cursor.
3. <CTRL>+<H>: Cut from the cursor to the beginning of line.
4. <CTRL>+<E>: Cut from the cursor to the end of line.
5. <CTRL>+<Y>: Paste text in paste buffer at the cursor position.

Each of cut operations defines cut direction: cut forward or cut backward. Obviously, <Delete> and <CTRL>+<E> cut forward; <Backspace> and <CTRL>+<H> cut backward. When two consecutive cut operations have the same cut direction, the cutting text is appended the paste buffer. Otherwise, the paste buffer is overwritten by the latest chopped text.

- **Command history**

  1. <Up>/<Down>: Move through the history of command lines in the older/newer direction. The tool remembers the history entry if the last executed line is in history.
  2. <CTRL>+<R> or <F8>: If you know what a previously executed line starts with, and you want to run it again, type prefix characters and then press <CTRL>+<R>, or <F8> on Windows platform, to iterate through the history of commands with matching prefix.

- **Tab completion**

  Use <Tab> to complete shell commands, COPT parameters/attributes, or files under specified path. To cycle though multiple matches, just repeat pressing <Tab>.

  1. If the cursor is over or right after the first word on the current command line, press <Tab> to complete available shell commands with matching prefix (from the cursor to the first character of word).
  2. Otherwise, press <Tab> to complete COPT parameters/attributes, or file names under path with matching prefix. Specifically, if the prefix matches with COPT parameters as well as file under current working directory, only COPT parameters will be listed. In this case, to iterate file names, add relative path `'./'` to start with.
  3. For convenience, tab completion ignores case and support asterisk (*) as wildcard to match file and directory pattern.
  4. <Shift>+<Tab>: Complete the next one in an opposite direction.

## 3.3 Script mode

There are two approaches to run scripts, a batch of commands, in COPT command-line. One is to save scripts as a text file. The other is called **inline scripts**, that is, a quoted string of commands separated by `';'`. Both of them can be loaded when COPT command-line is invoked, or loaded on fly in the edit mode (see shell command `'load'`). Below describes more details about loading scripts as arguments.

This tool allows users to load a script file to do a batch job automatically. As mentioned in *Overview*, a script file is read when its file path is specified as arguments of the option `'-i'`.

- When reading a script file, COPT command-line double checks whether the first non-blank line starts with special text: `'#COPT script file'`. This is to make sure users do not load an invalid script file by mistake. Indeed, only `'#COPT'` is verified. In addition, any line in scripts is commented out if its first non-blank character is `'#'`.
- After a script file is loaded, the tool keeps reading it as standard inputs, until reaching end of file or a special character `'?'`. Here, we use question mark `'?'` to pause scripts on purpose. To continue, users can type `'load'` in command line. Afterwards, the tool picks whatever left in scripts and start to run from there, until reaching end of file or another question mark `'?'`. Once current scripts finish, users can load any other script file on fly.

It also allows users to load special scripts, called **inline scripts**. The only difference from a script file is that commands are separated by `';'`, instead of `'\n'`. So inline scripts can be read by using arguments of the option `'-c'`, or loaded on fly by specifying a quoted string, and special character `'?'` works in the same way.

In addition, this tool provides a feature of recording **valid** command lines sequentially to a script file, if users specify an output script file as argument of the option `'-o'`. Here, **valid** command must use known shell commands and do not exceed number of allowed parameters.

In particular, if users load a script file or inline scripts on fly, all commands in scripts are written to the output script file. Note that command `'load'` itself is not written to output script file on purpose. Because we've expanded and written all commands in scripts. On the other hand, it may trigger infinity loop if the script file loaded is actually itself.

## 3.4 Shell commands

COPT command-line supports the following shell commands for users to manipulate optimization models. Moreover, shell commands are case-insensitive and support tab-completion.

### 3.4.1 General shell commands

The shell commands below are in support of interactions.

- `cd`: This shell command works similar to DOS command `'cd'`. That is, it changes **current working directory**, if its argument is valid relative or absolute path of a directory. Note that **current working directory** is the base directory for relative path and tab completion. It is initialized to current binary folder where `copt_cmd` exist. Users can change it by shell command `'cd <dirpath>'`. For example, if users change working directory to a folder having mps files, reading model becomes much easier because only filename is needed.
- `dir/ls`: This shell command works similar to DOS command `'dir'` or Bash command `'ls'`. That is, it lists all files and directories under given relative or absolute path. To see files under current working directory, type `'dir'` or `'ls'`; To see files under parent folder, type `'dir ../'`; To see files under home path, type `'dir ~/'`, etc. In addition, wildcard (`*`) is supported as well. That is, `'dir net'` lists all file names starting with `'net'` under current working directory; `'dir /home/user/*.gz'` lists all files of type of `'.gz'` under path of `'/home/user/'`.
- `exit/quit`: Leave COPT command-line.
- `help`: It provides information on all shell commands. Typing `'help'` followed by a shell command name gives you more details on shell commands. In particular, typing `'help'` without arguments lists all shell commands with short descriptions. Right after overview of shell commands, the text `'help'` with additional whitespace appear in the new prompt line. So users can directly type, or possibly `<Tab>` complete, actual shell command to read more details.
- `load`: Load a script file or inline scripts on fly and then execute a batch of commands. The syntax of `'load'` command should be followed by either relative/absolute path of a script file, or quoted string of inline scripts. One special scenario is when current script is paused, that is, hit question mark (`'?'`) during execution. In this case, type `'load'` will continue the paused scripts. If users forgot having scripts in progress and try to load another scripts, it works as command `'load'` and any additional argument is ignored. This behavior is back to normal after reaching the end of current scripts.
- `pwd`: This shell command works similar to Bash command `'pwd'`. That is, display current working directory to let users know where they are.
- `status`: COPT command-line has a state machine on status of problem solving (see Fig. 3.1). This is used to guide users through steps. Typing `'status'` shows you current interactive status. The status exposed to users are as follows:
  - `Initial`, initial status, either right after the tool is invoked, or shell command `reset` is called.

– `Read`, read an optimization model in format of mps successfully.

– `SetParam`, set value of any COPT parameter successfully.

– `Optimize`, shell command `opt` is called to solve current optimization model.

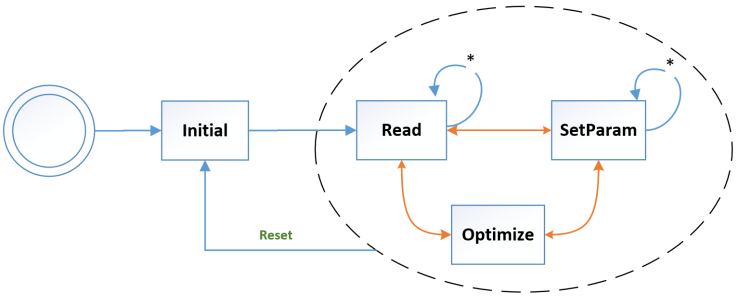



Fig. 3.1: Status of COPT command-line

### 3.4.2 COPT shell commands

The shell commands below are related to COPT API calls.

- `display/get`: Display current setting of any COPT parameter or attribute. Typing `'display'` followed by COPT parameter or attribute name shows its current value. Typing `'display'` only lists all COPT parameters and attributes with short descriptions. Right after overview of COPT parameters/attributes, the text `'display'` with additional whitespace appear in the new prompt line. So users can directly type, or possibly `<Tab>` complete, actual parameter/attribute name to see its current value.
- `opt/optimize`: Solve optimization model and display results on screen. This command requires no parameter and its interactive status is set to `'Optimize'`.
- `optlp/optimizelp`: Solve optimization model as LP model. This command requires no parameter and its interactive status is set to `'Optimize'`.
- `iis`: Compute IIS for the loaded model.
- `feasrelax`: Do feasible relaxation for infeasible problem. Note that an optimization problem must exist before calling 'feasrelax'. `'feasrelax'` or `'feasrelax all'` means to relax all bounds of variables and constraints with penalty 1, while `'feasrelax vars'` means to only relax bounds of variables with penalty 1, and `'feasrelax cons'` means to only relax bounds of constraints with penalty 1.
- `tune`: Tuning parameter automatically of the loaded model.
- `loadtuneparam`: Loads the specified tuning results into the currently loaded model. The calling command is `'loadtuneparam idx'`, where `'idx'` is the specified tuning result's number. Use the command `'get TuneResults'` to get the number of tuning results obtained after the current tuning.
- `read`: Read optimization model, MIP solution, LP basis, MIP initial solution and COPT parameters from file under given relative/absolute path. It supports files including optimization problem file of type `'.mps'` and compressed format `'.mps.gz'`, problem file of type `'.lp'` and compressed format `'.lp.gz'`, problem file of type `'.dat-s'` and compressed format `'.dat-s.gz'`, problem file of type `'.cbf'` and compressed format `'.cbf.gz'`, problem file of type `'.bin'` and compressed format `'.bin.gz'`, basis file of type `'.bas'`, solution file of type `'.sol'`, MIP initial solution file of type `'.mst'`, parameter file of type `'.par'` and tuning file of type `'.tune'`.
- `readmps`: Read optimization model in format of `'.mps'` or `'.mps.gz'` from file. Note this command does not require file type `'.mps'` or `'.mps.gz'`. That is, it is fine to have content in format of MPS, regardless of its file name. In addition, the interactive status is set to `'Read'`.

- `readlp`: Read optimization model in format of `'.lp'` or `'.lp.gz'` from file. Note this command does not require file type `'.lp'` or `'.lp.gz'`. That is, it is fine to have content in format of LP, regardless of its file name. In addition, the interactive status is set to `'Read'`.
- `readsdpa`: Read an optimization problem in format of `'.dat-s'` or `'.dat-s.gz'` from file. Note that this command is similar to `'read'` when the file type is `'.dat-s'` . However, `'readsdpa'` does not require the file has type `'.dat-s'`. It parses the file as SDPA format, no matter what file type it is.
- `readcbf`: Read an optimization problem in format of `'.cbf'` or `'.cbf.gz'` from a file. Note that this command is similar to `'read'` when the file type is `'.cbf'` . However, `'readcbf'` does not require the file has type `'.cbf'`. It parses the file as CBF format, no matter what file type it is.
- `readbin`: Read optimization model in format of `'.bin'` or `'.bin.gz'` from file. Note this command does not require file type `'.bin'` . That is, it is fine to have content in format of COPT binary format, regardless of its file name. In addition, the interactive status is set to `'Read'`.
- `readsol`: Read MIP solution from file. Note this command does not require file type of `'.sol'`. That is, it is fine to have content of solution, regardless of its file name.
- `readbasis`: Read optimal basis from file. Note this command does not require file type of `'.bas'`. That is, it is fine to have content of basis, regardless of its file name.
- `readmst`: Read MIP initial solution from file. Note this command does not require file type of `'.mst'`. That is, it is fine to have content of solution, regardless of its file name.
- `readparam`: Read COPT parameters from file and set corresponding values. Note this command does not require file type of `'.par'`. That is, it is fine to have content of COPT parameters, regardless of its file name.
- `readtune`: Read the parameter file under the given relative or absolute path. Note this command does not require file type of `'.tune'` . That is, it is fine that the file conforms to the COPT parameter tuning file format, regardless of its file name.
- `reset`: Reset current optimization model and all parameters/attributes to defaults. In addition, the interactive status is set to `'Initial'`.
- `set`: Set value of any COPT parameter. The syntax of this command should be `'set'`, followed by COPT parameter name and an integer/double number. Moreover, Typing `'set'` only lists all COPT parameters with short descriptions. Right after overview of COPT parameters, the text `'set'` with additional whitespace appears in the new prompt line. So users can directly type, or possibly `<Tab>` complete, actual parameter name to complete as partial command. If so, its current value, default value, min value, max value of given COPT parameter are displayed on screen. Now users know how to add correct value to complete the full command of `'set'`. One tip of saving typing here is to get last history entry by pressing `<Up>`. At last, the interactive status is set to `'SetParam'`.
- `set LogFile`:

  Set the log file for recording solver's logging output, e.g., `'set LogFile filename.log'`.

  This must be specified before the solve command (`opt`) is executed. COPT will write log to the specified file during the solving process.
- `set CpuAffinity`:

  Set the CPU affinity for the current solver process, e.g., `'set CpuAffinity 0f'`.
- `get CpuAffinity`:

  Get the CPU affinity configuration of the current solver process.
- `get NumaNodeCnt`:

  Get the number of NUMA nodes available on the system.
- `set CpuBind`:

  Bind the current solver process to the specified NUMA node, e.g., `'set CpuBind 0'`.

- `set MemBind`:

  Bind the memory allocations of the solver process to the specified NUMA node, e.g., `'set MemBind 0'`.

- `write`: Output MPS/LP/CBF format model, COPT binary format model, IIS model, FeasRelax model, LP/MIP solution, optimal basis, settings of modified COPT parameters to file under given relative/absolution path. This command detects file types. For an instance, `'write diet.sol'` outputs LP solution to file `'diet.sol'`. An error message will be shown to users if file type do not match. Supported file types are: `'.mps'`, `'.lp'`, `'.bin'`, `'cbf'`, `'.iis'`, `'.relax'`, `'.sol'`, `'.bas'`, `'.mst'` and `'.par'`.

- `writemps`: Output current optimization model to a file of type `'.mps'`. Note `'.mps'` is appended to the file name, if users do not add it.

- `writelp`: Output current optimization model to a file of type `'.lp'`. Note `'.lp'` is appended to the file name, if users do not add it.

- `writecbf`: Output problem to a file of type `'.cbf'`. Note `'.cbf'` is appended to the file name, if users do not add it.

- `writebin`: Output current optimization model to a file of type `'.bin'`. Note `'.bin'` is appended to the file name, if users do not add it.

- `writeiis`: Output computed IIS model to a file of type `'.iis'`. Note `'.iis'` is appended to the file name, if users do not add it.

- `writerelax`: Output feasibility relaxation problem to a file of type `'.relax'`. Note `'.relax'` is appended to the file name, if users do not add it.

- `writesol`: Output LP solution of problem to a file of type `'.sol'`. Note `'.sol'` is appended to the file name, if users do not add it.

- `writepoolsol`: For the current problem, save the solution with the specified number in the solution pool to the result file under the given relative or absolute path. If the file extension is not``'.sol'`` , the suffix `'.sol'` is automatically added. The calling command is `'writepoolsol idx pool_idx.sol'`, where `'idx'` is the specified solution number in solution pool. Use the command `'get PoolSols'` to get the current number of solution pool results.

- `writebasis`: Output optimal basis to a file of type `'.bas'`. Note `'.bas'` is appended to the file name, if users do not add it.

- `writemst`: Output current best MIP solution to a file of type `'.mst'`. Note `'.mst'` is appended to the file name, if users do not add it.

- `writeparam`: Output modified COPT parameters to a file of type `'.par'`. Note `'.par'` is appended to the file name, if users do not add it.

- `writetuneparam`: Save the specified tuning result to the parameter setting file under the given relative or absolute path. The calling command is `'writetuneparam idx tune_idx.par'`, where `'idx'` is the specified tuning result number. Use the command `'get TuneResults'` to get the number of tuning results obtained after the current tuning.

## 3.5 Example Usage

This section shows how to use COPT command-line to interactively solve a well-known problem, called "Diet Problem". Please refer to *AMPL Interface - Example Usage* for problem description in more detail.

With valid license, COPT command-line should run as follows, after entering `copt_cmd` in command prompt.

```
copt_cmd
```

If users are new to COPT command-line, always start with shell command `'help'`.

```
Cardinal Optimizer v8.0.7. Build date Mar 04 2024
Copyright Cardinal Operations 2025. All Rights Reserved

COPT>
```

Suppose diet model has mps format and exits in the current working directory. In this way, we just type its file name to read, without worrying about its path.

```
COPT> read diet.mps
Reading from '/home/username/copt/diet.mps'
Reading finished (0.00s)
```

Before solving it, users are free to tune any COPT parameter. Below is an example to set value of double parameter `TimeLimit` to 10s.

If users are not familiar with COPT parameters, just type `'set'` to list all public COPT parameters and attributes with short description. Furthermore, `'set'` with parameter name, for example `'set TimeLimit'`, displays its current value, default value, min value and max value of the given parameter.

```
COPT> set timelimit 10

Setting parameter 'TimeLimit' to 10
```

After tuning parameters, it is time to solve the model. The messages during problem solving are shown as follows.

```
COPT> opt
Model fingerprint: 129c032d

Hardware has 4 cores and 8 threads. Using instruction set X86_NATIVE (1)
Minimizing an LP problem

The original problem has:
    4 rows, 8 columns and 31 non-zero elements
The presolved problem has:
    4 rows, 8 columns and 31 non-zero elements

Starting the simplex solver using up to 8 threads

Method   Iteration           Objective  Primal.NInf   Dual.NInf        Time
Dual             0    0.0000000000e+00            4           0       0.01s
Dual             1    8.8201991646e+01            0           0       0.01s
Postsolving
Dual             1    8.8200000000e+01            0           0       0.01s

Solving finished
Status: Optimal  Objective: 8.8200000000e+01  Iterations: 1  Time: 0.01s
```

After solving the model, users might check results by using shell command `'get'` with a parameter name. Note that, similar to `'set'`, type `'get'` to list all public parameters and attributes. In particular, `'get all'` shows all parameters/attributes and their current value.

```
COPT> get TimeLimit
  DblParam: [Current]  10s
COPT> get LpObjVal
  DblAttr:  [Current] 88.2
COPT> get LpStatus
  IntAttr:  [Current] 1 optimal
```

Before leaving COPT command-line, users might output the model in format of mps, optimal basis,

modified parameters, or LP solution to files. Below is an example to write LP solution to current directory.

```
COPT> writesol diet
  Writing solutions to /home/username/copt/diet.sol
COPT> quit
  Leaving COPT...
```

Below is the script file `diet.in` by putting everything together, see Listing 3.1:

Listing 3.1: `diet.in`

```
#COPT script-in file

read diet.mps
set timelimit 10
opt
writesol diet
quit
```

which is loaded by using the option `'-i'` when starting `copt_cmd`:

```
copt_cmd -i diet.in
```

or executing shell command `load` on fly.

```
COPT> load diet.in
```

# Chapter 4

# COPT Floating Licensing service

The **Cardinal Optimizer** provides COPT Floating Token Server on all supported platforms, who serve license tokens to COPT client applications over local network.

Once you have floating license properly installed, server owns a set of license tokens up to number described in the license file. Any properly configured COPT client of the same version can request a token from server and release it whenever quit.

## 4.1 Server Setup

The application of COPT Floating Token server includes `copt_flserver` executable and a configuration file `fls.ini`. The very first thing to do when server starts is to verify floating license locally, whose location is specified in `fls.ini`. If local validation passes, server connects to remote COPT licensing server for further validation, including machine IP, which is supposed to match IP range that user provided during registration. This means the machine running COPT Floating Token Server should have internet access in specified area. For details, please see descriptions below or refer to *How to obtain and setup license*.

### 4.1.1 Installation

The **Cardinal Optimizer** provides a separate package for remote services, which include COPT floating token server. Users may apply for remote package from customer service. Afterwards, unzip the remote package and move to any folder on your computer. The software is portable and does not change anything in the system it runs on. Below are details of installation.

**Windows**

Please unzip the remote package and move to any folder. Though, it is common to move to folder under `C:\Program Files`.

**Linux**

To unzip the remote package, enter the following command in terminal:

```
tar -xzf CardinalOptimizer-Remote-8.0.7-lnx64.tar.gz
```

Then, the following command moves folder copt_remote80 in current directory to other path. For an example, admin user may move it to folder under `/opt` and standard user may move it to `$HOME`.

```
sudo mv copt_remote80 /opt
```

Note that it requires `root` privilege to execute this command.

**MacOS**

To unzip the remote package, enter the following command in terminal:

```
tar -xzf CardinalOptimizer-Remote-8.0.7-universal_mac.tar.gz
```

Then, the following command moves folder copt_remote80 in current directory to other path. For an example, admin user may move it to folder under `/Applications` and standard user may move it to `$HOME`.

```
mv copt_remote80 /Applications
```

### 4.1.2 Floating License

After installing COPT remote package, it requires floating license to run. It is preferred to save floating license files, `license.dat` and `license.key`, to `floating` folder in path of remote package.

The following explains how to obtain the license file via the `copt_licgen` tool and the license credential information `key` under different systems.

**Note**

If the user has already obtained the two license files `license.dat` and `license.key`, there is no need to obtain them again. You can skip the following steps to obtain the license file and refer to *Configuration* directly.

#### Windows

If the COPT remote package is installed under `"C:\Program Files"`, execute the following command to enter `floating` folder in path of remote package.

```
cd "C:\Program Files\copt_remote80\floating"
```

Note that the tool `copt_licgen` creating license files exists under `tools` folder in path of remote package. The following command creates floating license files in current directory, given a floating license key, such as `7483dff0863ffdae9fff697d3573e8bc` .

```
..\tools\copt_licgen -key 7483dff0863ffdae9fff697d3573e8bc
```

#### Linux and MacOS

If the COPT remote package is installed under `"/Applications"`, execute the following command to enter `floating` folder in path of remote package on MacOS system.

```
cd /Applications/copt_remote80/floating
```

The following command creates floating license files in current directory, given a floating license key, such as `7483dff0863ffdae9fff697d3573e8bc` .

```
../tools/copt_licgen -key 7483dff0863ffdae9fff697d3573e8bc
```

In addition, if users run the above command when current directory is different than `floating` folder in path of remote package, it is preferred to move them to `floating`. The following command does so.

```
mv license.* /Application/copt_remote80/floating
```

### 4.1.3 Configuration

Below is a typical configuration file, `fls.ini`, of COPT Floating Token Server.

```
[Main]
Port = 7979

# password is case-sensitive and default is empty
# it applies to both copt clients and managing tool
PassWd =

[SSL]
# needed if connecting using SSL
CaFile =
CertFile =
CertkeyFile =

[Licensing]
# if empty or default license name, it is from binary folder
# to get license files from cwd, add prefix "./"
# full path is supported as well
LicenseFile = license.dat
PubkeyFile = license.key

[WLS]
# WebServer have a default host and no need to edit in most scenarios
# Must specify WebLicenseId and WebAccesskey to trigger web licensing
WebServer =
WebLicenseId =
WebAccessKey =
WebTokenDuration = 300

[Filter]
# default policy 0 indicates accepting all connections, except for ones in blacklist
# otherwise, denying all connections except for ones in whitelist
DefaultPolicy = 0
UseBlackList = true
UseWhiteList = true
FilterListFile = flsfilters.ini

[Logs]
LogsFolder = ./logs
```

The `Main` section specifies port number, through which COPT clients connect to server and then obtain the license token. The `Licensing` section specifies location of floating license. As described in comments above, if empty string or default license file name is specified, floating license files are read from the binary folder where the server executable reside.

It is possible to run COPT Floating Token server, even if floating license files do not exist in the same folder as the server executive. One solution is to set `LicenseFile = ./license.dat` and `PubkeyFile = ./license.key`. By doing so, server read floating license from the current working directory. That is, user could execute server application at the path where floating license files exist.

The other solution is to set full path of license files in configuration. As mentioned before, Cardinal Optimizer allows users to set environment variable `COPT_LICENSE_DIR` for license files. For details, please refer to *How to install Cardinal Optimizer*. If user prefers the way of environment variable, the configuration file should have the full path to floating license.

In the `Filter` section, `DefaultPolicy` has default value 0, meaning all connections are accepted except for those in black lists; if it is set to non-zero value, then all connection are blocked except for those

in white lists. In addition, black list is enabled if `UseBlackList` is true and white list is enabled if `UseWhiteList` is true. The filter configuration file is specified by `FilterListFile`. Below is an example of the filter configuration file.

```
[BlackList]
# 127.0.*.* + user@machine*

[WhiteList]
# 127.0.1.2/16 - user@machine*

[ToolList]
# only tool client at server side can access by default
127.0.0.1/32
```

It has three sections and each section has its own rules. In section of `BlackList`, one may add rules to block others from connection. In section of `WhiteList`, one may add rules to grant others for connection, even if the default policy is to block all connections. Only users listed in section of `ToolList` are able to connect to floating token server by Floating Token Server Managing Tool (see below for details).

Specifically, rules in filter configuration have format of starting with IP address. To specify IP range, you may include wildcard (*) in IP address, or use CIDR notation, that is, an IPv4 address and its associated network prefix. In addition, a rule may include (+) or exclude (-) given user at given machine, such as `127.0.1.2/16 - user@machine`. Here, `user` refers to `username`, which can be queried by `whoami` on Linux/MacOS platform; `machine` refers to `computer name`, which can be queried by `hostname` on Linux/MacOS platform.

### 4.1.4 Web License for Floating Server

Besides local floating license above, users may use web license for floating server to run floating service. This requires that the machine running floating server must have internet access. However, hardware info are not required any more. That is, users are free to deploy floating server to any cloud machine or container, as long as they have internet access. Please refer to COPT web license page for details.

Below are brief steps:

- Follow steps to register an account and apply for trial of web license for floating server.
- Once approved, *Web License ID* is generated for users
- On page of *API Keys*, create *Web Access Key* using given *Web License ID*

Afterwards, users edit configuration file *fls.ini* and add values of both *Web License ID* and *Web Access Key* to related keywords in section of *WLS*. For instance,

```
[WLS]
# WebServer have a default host and no need to edit in most scenarios
# Must specify WebLicenseId and WebAccesskey to trigger web licensing
WebServer =
WebLicenseId =
WebAccessKey =
WebTokenDuration = 300
```

As of now, floating server talks to COPT web license page for licensing. Users are able to monitor its token usage and other information online.

### 4.1.5 Example Usage

Suppose that floating license exists in the same folder where the server executable reside. To start the COPT Floating Token Server, just execute the following command at any directory in Windows console, or Linux/Mac terminal.

```
copt_flserver
```

If you see log information as follows, the Floating Token Server has been successfully started. Server monitors any connection from COPT clients, manages approved clients as well as requests in queue. User can stop Floating Token Server anytime when entering `q` or `Q`.

```
> copt_flserver
  [ Info] Floating Token Server, COPT v8.0.7 20240304
  [ Info] server started at port 7979
```

If failed to verify local floating license, or something is wrong on remote COPT license server, you might see error logs as follows.

```
> copt_flserver
  [ Info] Floating Token Server, COPT v8.0.7 20240304
  [Error] Invalid signature in public key file
  [Error] Fail to verify local license
```

and

```
> copt_flserver
  [ Info] Floating Token Server, COPT v8.0.7 20240304
  [Error] Error to connect license server
  [Error] Fail to verify floating license by server
```

## 4.2 Client Setup

COPT Clients can be COPT command-line tool, or any application which solve problems using COPT api, such as COPT python interface. Floating licensing is a better approach in terms of flexibility and efficiency. Different from stand-alone license, any machine having properly configured COPT client can legally run Cardinal Optimizer, as long as peak number of connected clients does not exceed the token number.

### 4.2.1 Configuration

#### Via the configuration file

Before running COPT as floating client, please make sure that you have installed COPT locally. For details, please refer to *How to install Cardinal Optimizer*. Users can skip obtaining local licenses by adding a floating configuration file `client.ini`.

Below is a typical configuration file, `client.ini`, of COPT floating clients.

```
Host = 192.168.1.11
Port = 7979
QueueTime = 600
```

As configured above, COPT floating client tries to connect to `192.168.1.11` at port `7979` with wait time in queue up to 600 seconds. Here, `Host` is set to `localhost` if empty or not specified; `QueueTime` is set to 0 if empty or not specified. Specifically, empty QueueTime means client does not wait and should quit immediately, if COPT Floating Token Server have no tokens available. Port number must be great than zero and should be the same as that specified in server configuration file. Note that keywords in the client configuration file are case insensitive.

Without local license files, a COPT application still works if client configuration file, `client.ini`, exists in one of the following three locations in order, that is, current working directory, environment directory by `COPT_LICENSE_DIR` and binary directory where COPT executable is located.

By design, COPT application reads local license files instead of client configuration file, if they both exist in the same location. On the other hand, if local license files are under the environment directory, to activate approach of floating licensing, user can simply add a configuration file, `client.ini`, under the current working directory (different from the environment directory).

If a COPT application calls COPT API to solve problems, such as COPT python interface, license is checked as soon as COPT environment object is created. If there exists proper client configuration file, `client.ini`, a license token is granted to COPT client. This license token is released and sent back to token server, as soon as last COPT environment object in the same process destroys.

#### Via API Functions

In addition to the method using the `client.ini` file mentioned above, users can also configure the client in their code through API functions. Taking the COPT Python interface as an example, the corresponding class is *EnvrConfig Class*, and similar approaches apply to other programming languages. The configuration is as follows:

```
# Set client configuration parameters
envconfig.set(COPT.CLIENT_FLOATING, "192.168.1.11")
envconfig.set(COPT.CLIENT_PORT, "7979")
envconfig.set(COPT.CLIENT_WAITTIME, "600")
```

### 4.2.2 High Availability

In the case of multiple floating license servers, the client can achieve high availability by configuring multiple server IP addresses in the `Host` field of the `client.ini` file, as shown in the table below:

```
Host = 192.168.1.11; 192.168.1.22; 192.168.1.33
Port = 7979
WaitTime = 600
```

Here, the `Host` field contains the IP addresses of each floating license server. If the port number of a server is not `7979`, the port can be specified after the IP address.

The above configuration indicates that the client will first attempt to connect to the server `192.168.1.11:7979`. If that server is unavailable, the client will try connecting to `192.168.1.22:7979`, and so on, until a successful connection to one of the floating servers is made.

If a floating server fails during token issuance or management, the client will reassign to another available floating license server as specified in the configuration file. This setup ensures that the floating license service has disaster recovery capabilities, thereby improving the high availability of the COPT floating license service.

### 4.2.3 Example Usage

Suppose that we've set client configuration file `client.ini` properly and have no local license, below is an example of obtaining a floating token by COPT command-line tool `copt_cmd`. Execute the following command in Windows console, or Linux/Mac terminal.

```
copt_cmd
```

If you see log information as follows, the COPT client, `copt_cmd`, has obtained the floating token successfully. COPT command-line tool is ready to solve optimization problems.

```
> copt_cmd
  Cardinal Optimizer v8.0.7. Build date Mar 04 2024
  Copyright Cardinal Operations 2025. All Rights Reserved

  [ Info] initialize floating client: ./client.ini

  [ Info] connecting to server ...
  [ Info] connection established
COPT>
```

If you see log information as follows, the COPT client, `copt_cmd`, has connected to COPT Floating Token Server. But due to limited number of tokens, client waits in queue of size 1.

```
> copt_cmd
  Cardinal Optimizer v8.0.7. Build date Mar 04 2024
  Copyright Cardinal Operations 2025. All Rights Reserved

  [ Info] Initialize floating client: ./client.ini

  [ Info] connecting to server ...
  [Error] empty license and queue size 1
  [ Info] Wait for license in  2 / 39 secs
  [ Info] Wait for license in  4 / 39 secs
  [ Info] Wait for license in  6 / 39 secs
  [ Info] Wait for license in  8 / 39 secs
  [ Info] Wait for license in 10 / 39 secs
  [ Info] Wait for license in 20 / 39 secs
  [ Info] Wait for license in 30 / 39 secs
```

If you see log information as follows, the COPT client, `copt_cmd`, has connected to COPT Floating Token Server. But client refused to wait in queue, as Queuetime is 0.

```
> copt_cmd
  Cardinal Optimizer v8.0.7. Build date Mar 04 2024
  Copyright Cardinal Operations 2025. All Rights Reserved

  [ Info] Initialize floating client: ./client.ini

  [ Info] connecting to server ...
  [Error] Server error: "no more token available", code = 2
  [Error] Fail to open: ./license.dat

  [Error] Fail to initialize cmdline
```

If you see log information as follows, the COPT client, `copt_cmd`, fails to connect to COPT Floating Token Server. Finally, client quits after time limit.

```
> copt_cmd
  Cardinal Optimizer v8.0.7. Build date Mar 04 2024
  Copyright Cardinal Operations 2025. All Rights Reserved

  [ Info] initialize floating client: ./client.ini

  [ Info] connecting to server ...
  [ Info] wait for license in  2 / 10 secs
  [ Info] wait for license in  4 / 10 secs
  [ Info] wait for license in  6 / 10 secs
  [ Info] wait for license in  8 / 10 secs
  [ Info] wait for license in 10 / 10 secs
  [Error] timeout at waiting for license
  [Error] fail to open: ./license.dat
```

```
[Error] Fail to initialize cmdline
```

In addition, users can start the COPT command-line tool on the client side to connect to the specified floating server IP address as follows:

```
> copt_cmd -floating <ip>
```

## 4.3 Floating Token Server Managing Tool

COPT floating token service ships with a tool `copt_flstool`, for retrieving information and tune parameters of floating token server on fly.

### 4.3.1 Tool Usage

Execute the following command in Windows console, Linux or MacOS terminal:

```
> ./copt_flstool
```

Below displays help messages of this tool:

```
> ./copt_flstool
  COPT Floating Token Server Managing Tool

  copt_flstool [-s server ip] [-p port] [-x passwd] command <param>

  commands are:   addblackrule <127.0.0.1/20[-user@machine]>
                  addwhiterule <127.0.*.*[+user@machine]>
                  getfilters
                  getinfo
                  resetfilters
                  setpasswd <xxx>
                  toggleblackrule <n-th>
                  togglewhiterule <n-th>
                  writefilters
```

If the `-s` and `-p` option are present, tool connects to floating token server with given server IP and port. Otherwise, tool connections to localhost and default port 7979. If floating token server sets a password, tool must provide password string after the `-x` option.

This tool provides the following commands:

- `AddBlackRule`: Add a new rule in black filters. each rule has format starting with non-empty IP address, which may have wildcard to match IPs in the scope. In addition, it is optional to be followed by including (+) or excluding (-) user name at machine name.
- `AddWhiteRule`: Add a new rule in white filters. Note that a white rule has the same format as a black rule.
- `GetFilters`: Get all rules of black filters, white filters and tool filters, along with relative sequence numbers, which are parameters for command ToggleBlackRule and ToggleWhiteRule.
- `GetInfo`: Get general information of floating token server, including token usage, connected clients, and all COPT versions in support.
- `ResetFilters`: Reset filter lists in memory to those on filter config file.
- `SetPasswd`: Update password of target floating token server on fly.

- `ToggleBlackRule`: Toggle between enabling and disabling a black rule, given its sequence number by GetFilters.
- `ToggleWhiteRule`: Toggle between enabling and disabling a white rule, given its sequence number by GetFilters.
- `WriteFilters`: Write filter lists in memory to filter config file.

### 4.3.2 Example Usage

The following command lists general information on local machine.

```
> ./copt_flstool GetInfo

[ Info] COPT Floating Token Server Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to localhost:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] general info
  # of available tokens is 3 / 3, queue size is 0
  # of active clients is 0
```

To run managing tool on other machine, its IP should be added to a rule in `ToolList` section in filter configuration file `flsfilters.ini`. The following command from other machine lists information of server 192.168.1.11.

```
> ./copt_flstool -s 192.168.1.11 GetInfo

[ Info] COPT Floating Token Server Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] general info
  # of available tokens is 3 / 3, queue size is 0
  # of active clients is 0
```

The following command shows all filter lists of server 192.168.1.11, including those in BlackList section, WhiteList section and ToolList section.

```
> ./copt_flstool -s 192.168.1.11 GetFilters

[ Info] COPT Floating Token Server Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] filters info
[BlackList]

[WhiteList]

[ToolList]
  [1]  127.0.0.1
```

The following command added user of IP 192.168.3.13 to black list.

```
> ./copt_flstool -s 192.168.1.11 AddBlackRule 192.168.3.133

[ Info] COPT Floating Token Server Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] server added new black rule (succeeded)
```

The following command shows that a new rule in BlackList section is added.

```
> ./copt_flstool -s 192.168.1.11 GetFilters
```

```
[ Info] COPT Floating Token Server Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] filters info
[BlackList]
  [1] 192.168.3.133

[WhiteList]

[ToolList]
  [1]  127.0.0.1
```

The following command disable a rule in BlackList section.

```
> ./copt_flstool -s 192.168.1.11 ToggleBlackRule 1

[ Info] COPT Floating Token Server Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] server toggle black rule [1] (succeeded)
```

## 4.4 Running as service

To run COPT floating token server as a system service, follow steps described in `readme.txt` under `floating` folder, and set config file `copt_flserver.service` properly.

Below is `readme.txt`, which lists installing steps in both Linux and MacOS platforms.

```
[Linux] To run copt_flserver as a service with systemd

Add a systemd file
    cp copt_flserver.service to /lib/systemd/system/
    sudo systemctl daemon-reload

Enable new service
    sudo systemctl start copt_flserver.service
    or
    sudo systemctl enable copt_flserver.service

Restart service
    sudo systemctl restart copt_flserver.service

Stop service
    sudo systemctl stop copt_flserver.service
    or
    sudo systemctl disable copt_flserver.service

Verify service is running
    sudo systemctl status copt_flserver.service

[MacOS] To run copt_flserver as a service with launchctrl

Add a plist file
    cp copt_flserver.plist to /Library/LaunchAgents as current user
    or
    cp copt_flserver.plist to /Library/LaunchDaemons with the key 'UserName'
```

(continues on next page)

(continued from previous page)

```
Enable new service
    sudo launchctl load -w /Library/LaunchAgents/copt_flserver.plist
    or
    sudo launchctl load -w /Library/LaunchDaemons/copt_flserver.plist

Stop service
    sudo launchctl unload -w /Library/LaunchAgents/copt_flserver.plist
    or
    sudo launchctl unload -w /Library/LaunchDaemons/copt_flserver.plist

Verify service is running
    sudo launchctl list solver.copt.flserver
```

### 4.4.1 Linux

Below are the detailed steps for how to run COPT floating token server as a system service in Linux platform.

For instance, assume that COPT remote service is installed under `'/home/eleven'`. In your terminal, type the following command to enter the root directory of floating service.

`cd /home/eleven/copt_remote80/floating`

modify template of the service config file `copt_flserver.service` in text format:

```
[Unit]
Description=COPT Floating Token Server

[Service]
WorkingDirectory=/path/to/service
ExecStart=/path/to/service/copt_flserver
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

That is, update template path in keyword `WorkingDirectory` and `ExecStart` to actual path where the floating service exits.

```
[Unit]
Description=COPT Floating Token Server

[Service]
WorkingDirectory=/home/eleven/copt_remote80/floating
ExecStart=/home/eleven/copt_remote80/floating/copt_flserver
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

Afterwards, copy `copt_flserver.service` to system service folder `/lib/systemd/system/` (see below).

```
sudo cp copt_flserver.service /lib/systemd/system/
```

The following command may be needed if you add or update service config file. It is not needed if service unit has been loaded before.

```
sudo systemctl daemon-reload
```

The following command starts the new floating service.

```
sudo systemctl start copt_flserver.service
```

To verify the floating service is actually running, type the following command

```
sudo systemctl status copt_flserver.service
```

If you see logs similar to below, COPT floating server is running successfully as a system service.

```
copt_flserver.service - COPT Floating Token Server
Loaded: loaded (/lib/systemd/system/copt_flserver.service; enabled; vendor preset:␣
↪enabled)
Active: active (running) since Tue 2021-06-29 11:46:10 CST; 3s ago
Main PID: 3054 (copt_flserver)
    Tasks: 6 (limit: 4915)
CGroup: /system.slice/copt_flserver.service
        └─3054 /home/eleven/copt_remote80/floating/copt_flserver

eleven-ubuntu systemd[1]: Started COPT Floating Token Server.
eleven-ubuntu COPTCLS[3054]: LWS: 4.1.4-b2011a00, loglevel 1039
eleven-ubuntu COPTCLS[3054]: NET CLI SRV H1 H2 WS IPv6-absent
eleven-ubuntu COPTCLS[3054]: server started at port 7979
```

To stop the floating service, type the following command

```
sudo systemctl stop copt_flserver.service
```

### 4.4.2 MacOS

Below are the detailed steps for how to run COPT floating token server as a system service in MacOS platform.

For instance, assume that COPT remote service is installed under `'/Applications'`. In your terminal, type the following command to enter the root directory of floating service.

```
cd /Applications/copt_remote80/floating
```

modify template of the service config file `copt_flserver.plist` in xml format:

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>solver.copt.flserver</string>
        <key>Program</key>
        <string>/path/to/service/copt_flserver</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

That is, update template path in `Program` tag to actual path where the floating service exits.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
```

```
    <dict>
        <key>Label</key>
        <string>solver.copt.flserver</string>
        <key>Program</key>
        <string>/Applications/copt_remote80/floating/copt_flserver</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

Afterwards, copy `copt_flserver.plist` to system service folder `/Library/LaunchAgents` (see below).

```
sudo cp copt_flserver.plist /Library/LaunchAgents
```

The following command starts the new floating service.

```
sudo launchctl load -w /Library/LaunchAgents/copt_flserver.plist
```

To verify the floating service is actually running, type the following command

```
sudo launchctl list solver.copt.flserver
```

If you see logs similar to below, COPT floating server is running successfully as a system service.

```
{
    "LimitLoadToSessionType" = "System";
    "Label" = "solver.copt.flserver";
    "OnDemand" = false;
    "LastExitStatus" = 0;
    "PID" = 16406;
    "Program" = "/Applications/copt_remote80/floating/copt_flserver";
};
```

To stop the floating service, type the following command

```
sudo launchctl unload -w /Library/LaunchAgents/copt_flserver.plist
```

If the floating service should be run by a specific user, add `UserName` tag to config file. Below adds a user `eleven`, who has privilege to run the floating service.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>solver.copt.flserver</string>
        <key>Program</key>
        <string>/Applications/copt_remote80/floating/copt_flserver</string>
        <key>UserName</key>
        <string>eleven</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

Then copy new `copt_flserver.plist` to system service folder `/Library/LaunchDaemons` (see below).

```
sudo cp copt_flserver.plist /Library/LaunchDaemons
```

The following command starts the new floating service.

```
sudo launchctl load -w /Library/LaunchDaemons/copt_flserver.plist
```

To stop the floating service, type the following command

```
sudo launchctl unload -w /Library/LaunchDaemons/copt_flserver.plist
```

# Chapter 5

# COPT Compute Cluster Service

The **Cardinal Optimizer** provides COPT compute cluster service on all supported platforms, which allows you to offload optimization computations from COPT client applications over local network.

Once COPT compute cluster server runs at local network, any COPT client application with matching COPT version can connect to server and offload optimization computations. That is, COPT compute cluster clients are allowed to do modeling locally, execute optimization jobs remotely, and then obtain results interactively.

Note that the more computing power server has, the more optimization jobs can run simultaneously. Furthermore, COPT compute cluster service has functionality to cluster multiple servers together and therefore serve more COPT compute cluster clients over local network.

## 5.1 Server Setup

The COPT compute cluster service includes `copt_cluster` executable and a configuration file `cls.ini`. The very first thing to do when cluster server starts is to verify cluster license locally, whose path is specified in `cls.ini`. If local validation passes, cluster server might connect remotely to COPT licensing server for further validation, such as verifying machine IP, which is supposed to match IP range that user provided during registration. This means the server running COPT compute cluster service should have internet access in specified area, such as campus network. For details, please see descriptions below or refer to *How to obtain and setup license*.

### 5.1.1 Installation

The **Cardinal Optimizer** provides a separate package for remote services, which include COPT compute cluster. Users may apply for remote package from customer service. Afterwards, unzip the remote package and move to any folder on your computer. The software is portable and does not change anything in the system it runs on. Below are details of installation.

**Windows**

Please unzip the remote package and move to any folder. Though, it is common to move to folder under `C:\Program Files`.

**Linux**

To unzip the remote package, enter the following command in terminal:

```
tar -xzf CardinalOptimizer-Remote-8.0.7-lnx64.tar.gz
```

Then, the following command moves folder copt_remote80 in current directory to other path. For an example, admin user may move it to folder under `/opt` and standard user may move it to `$HOME`.

```
sudo mv copt_remote80 /opt
```

Note that it requires `root` privilege to execute this command.

**MacOS**

To unzip the remote package, enter the following command in terminal:

```
tar -xzf CardinalOptimizer-Remote-8.0.7-universal_mac.tar.gz
```

Then, the following command moves folder copt_remote80 in current directory to other path. For an example, admin user may move it to folder under `/Applications` and standard user may move it to `$HOME`.

```
mv copt_remote80 /Applications
```

If you see errors below or similar signature problem of COPT lib during installation,

```
"libcopt.dylib" cannot be opened because the developer cannot be verified.
macOS cannot verify that this app is free from malware.
```

run the following command as root user, to bypass check of loading dynamic lib on MacOS.

```
xattr -d com.apple.quarantine CardinalOptimizer-Remote-8.0.7-universal_mac.tar.gz
```

or

```
xattr -dr com.apple.quarantine /Applications/copt_remote80
```

### 5.1.2 Cluster License

After installing COPT remote package, it requires cluster license to run. It is preferred to save cluster license files, `license.dat` and `license.key`, to `cluster` folder in path of remote package.

The following explains how to obtain the license file via the `copt_licgen` tool and the license credential information `key` under different systems.

> **Note**
>
> If the user has already obtained the two license files `license.dat` and `license.key`, there is no need to obtain them again. You can skip the following steps to obtain the license file and refer to *Configuration* directly.

**Windows**

If the COPT remote package is installed under `"C:\Program Files"`, execute the following command to enter `cluster` folder in path of remote package.

```
cd "C:\Program Files\copt_remote80\cluster"
```

Note that the tool `copt_licgen` creating license files exists under `tools` folder in path of remote package. The following command creates cluster license files in current directory, given a cluster license key, such as `7483dff0863ffdae9fff697d3573e8bc` .

```
..\tools\copt_licgen -key 7483dff0863ffdae9fff697d3573e8bc
```

**Linux and MacOS**

If the COPT remote package is installed under `"/Applications"`, execute the following command to enter `cluster` folder in path of remote package on MacOS system.

```
cd /Applications/copt_remote80/cluster
```

The following command creates cluster license files in current directory, given a cluster license key, such as `7483dff0863ffdae9fff697d3573e8bc` .

```
../tools/copt_licgen -key 7483dff0863ffdae9fff697d3573e8bc
```

In addition, if users run the above command when current directory is different than `cluster` folder in path of remote package, it is preferred to move them to `cluster`. The following command does so.

```
mv license.* /Application/copt_remote80/cluster
```

### 5.1.3 Configuration

Below is a typical configuration file, `cls.ini`, of COPT compute cluster.

```
[Main]
Port = 7878
# number of total tokens, which copt jobs can run simultaneously up to
NumToken = 3
# password is case-sensitive and default is empty
# it applies to both copt clients and cluster nodes
PassWd =
# data folder of cluster relative to its binary folder,
# where various of copt libraries and temporary job files reside.
DataFolder = ./data

[Memory]
# Example memory management configuration
# memory watermark defense
MemGuard = true
MemSoft = 0.80
MemHard = 0.92
MemFatal = 0.97
MemEvictPolicy = largest
# automatic per-task memory cap
MemLimitAuto = true
MemUsableRatio = 0.90
MemInflation = 1.35
MemOvercommit = 1.50
# 0 means detecting physical memory from the system
MemPhysicalMB = 0

[SSL]
# needed if connecting using SSL
CaFile =
CertFile =
CertkeyFile =

[Licensing]
# if empty or default license name, it is from binary folder
# to get license files from cwd, add prefix "./"
# full path is supported as well
LicenseFile = license.dat
PubkeyFile = license.key

[WLS]
# WebServer have a default host and no need to edit in most scenarios
# Must specify WebLicenseId and WebAccesskey to trigger web licensing
# If specified, ignore settings in Licensing section
WebServer =
WebLicenseId =
```

(continues on next page)

(continued from previous page)

```
WebAccessKey =
WebTokenDuration = 300

[FLS]
# Lease token from floating server
# If specified, ignore settings in Licensing section and WLS section
TokenServer =
TPort = 7979

[Cluster]
# host name and port of parent node in cluster
# If specified or changed on fly, connect to parent node
Parent =
PPort = 7878

[Filter]
# default policy 0 indicates accepting all connections, except for ones in blacklist
# otherwise, denying all connections except for ones in whitelist
DefaultPolicy = 0
UseBlackList = true
UseWhiteList = true
FilterListFile = clsfilters.ini

[Logs]
LogsFolder = ./logs
```

The `Main` section specifies port number, through which COPT compute cluster clients connect to server; token number, the number of optimization jobs that server can run simultaneously up to; password string, if specified, cluster clients should send the same password when requesting for service.

**Main:**

In the `[Main]` section of the configuration file, users can set the following connection information:

- `Port` : Specifies the connection port for the cluster server, through which COPT compute cluster clients connect to server;
- `NumToken` : Defines the number of optimization jobs that the COPT cluster server can run simultaneously up to. This number can be estimated based on the server's hardware resource capacity and is user-configurable. The default value is 3, with no concurrency limit.
- `PassWd` : Sets the password for client connections to the cluster server. The default is empty, meaning no password is required. If specified, clients should send the same password when requesting for cluster service. For a more secure connection, users can enable encrypted communication using RSA certificates in the `[SSL]` section.
- `DataFolder` : Specifies the path where the COPT solver is installed on the server.

The COPT compute cluster may install multiple versions of COPT to subfolder of `DataFolder`. Only clients with matching version (major and minor) will get approved and then offload optimization jobs at server side. Note that the COPT compute cluster pre-installs a COPT solver of the same version as server itself, which illustrate how to install other versions of COPT.

For instance, the COPT compute cluster has default COPT v8.0.7 installed and users plan to install COPT of other version v6.0.7. Users may create a folder ./data/copt/6.0.7/ and copy a COPT C lib of the same version to it. Specifically, on Linux platform, copy C dynamic library `libcopt.so` from the binary folder `$COPT_HOME/lib/` of COPT v6.0.7 to subfolder ./data/copt/6.0.7/ of the COPT compute cluster.

Furthermore, users are allowed to install newer version of COPT than cluster server version, such as COPT v9.0.1. To do so, follow the same step of copying a C lib of COPT v9.0.1 to ./data/copt/9.0.1/.

In addition, users need a personal license of v9.0.1 to load C lib of COPT v9.0.1 at server side. That is, copy valid personal license files to folder ./data/copt/9.0.1/ as well. However, this simple procedure may break if the newer COPT solver has significant updates. In this case, it is necessary to upgrade the COPT compute cluster to newer version, that is, v9.0.1.

Below is an example of directory structure of cluster server on Linux platform. It includes pre-installed COPT v8.0.7, COPT of previous version v6.0.7, and COPT of newer version v9.0.1.

```
~/copt_remote80/cluster
│  cls.ini
│  copt_cluster
│  license.dat -> cluster license v8.0.7
│  license.key
│
└─data
    └─copt
        └─6.0.7
        │   libcopt.so
        └─8.0.7
        │   libcopt.so
        └─9.0.1
            libcopt.so
            license.dat -> license v9.0.1
            license.key
```

**Memory:**

In the `[Memory]` section of the configuration file, users can configure the per-task memory cap and the memory watermark defense of the COPT compute cluster server.

> **Note**
>
> The values shown in the `[Memory]` section above are examples only, and do not represent the default values of the program.

**Per-task memory cap:**

- `MemLimitAuto`: Whether to enable the automatic per-task memory cap. The default is `true`. This parameter is the master switch of the automatic per-task memory cap: when set to `true`, the server computes and applies a per-task memory cap at the start of every solve; when set to `false`, the server neither computes nor applies that cap, and solving tasks are subject only to the `MemLimit` set by the client itself and to the memory watermark defense. In addition, when the number of tokens is 0, the server does not apply a per-task memory cap;
- `MemUsableRatio`: The proportion of physical memory available to solving tasks;
- `MemInflation`: The conversion factor between the memory usage reported by the solver and the actual resident memory usage;
- `MemOvercommit`: How far the memory available to a single task may exceed its average memory share;
- `MemPhysicalMB`: The physical memory size of the server, in MB. When set to `0`, the server detects it automatically; in a memory-limited container, the cgroup memory limit is used.

**Memory watermark defense:**

- `MemGuard`: Whether to enable the memory watermark defense;
- `MemSoft`: The soft watermark. On reaching it, the server stops accepting new tasks;
- `MemHard`: The hard watermark. On reaching it, the server selects one solving task and interrupts it;

- `MemFatal`: The fatal watermark. On reaching it, the server interrupts all solving tasks;
- `MemEvictPolicy`: The policy for selecting the task to interrupt when the hard watermark is reached. The following values are supported:
  - `largest`: Select the task with the largest current memory usage. This is the default policy;
  - `longest`: Select the task that has been solving the longest;
  - `youngest`: Select the task whose solve started most recently;
  - `random`: Select a solving task at random.

`MemSoft`, `MemHard` and `MemFatal` all denote the proportion of used physical memory to total physical memory, ranging from `0.0` to `1.0`.

For the computation of the per-task memory cap, the watermark triggering mechanism, memory cap exemptions, compatibility with older COPT versions and deployment suggestions, please refer to *Server Memory Management*.

**Licensing:**

The `Licensing` section specifies location of cluster license files. As described in comments above, if empty string or default license file name ( `license.dat` or `license.key` ) is specified, cluster license files are read from the binary folder where the cluster executive reside.

It is possible to run COPT compute cluster service, even if cluster license files do not exist in the same folder as the cluster executive. One option is to set `LicenseFile = ./license.dat` and `PubkeyFile = ./license.key`. By doing so, the COPT compute cluster reads cluster license files from the current working directory. That is, user could execute command at the path where cluster license files exist to run service.

The other option is to set full path of license files in configuration. As mentioned before, Cardinal Optimizer allows user to set environment variable `COPT_LICENSE_DIR` for license files. For details, please refer to *How to install Cardinal Optimizer*. If user prefers the way of environment variable, `cls.ini` should have the full path to cluster license files.

**Cluster:**

In the `[Cluster]` section of the configuration file, users can set the parent node (IP and port) for the current cluster server connection. By default, this is empty, indicating no connection to other nodes.

- `Parent`: The IP address of the parent node within the local network.
- `PPort`: The port number of the parent node.

If there is only one cluster server or if the current server is the root node, this section does not need to be configured.

For multiple cluster servers, specifying a parent node allows users to form a tree-structured network topology for the COPT computing cluster. The current server can join the parent node's cluster through this configuration.

COPT supports cluster load balancing in multi-task computing scenarios, enabling computing tasks to be reasonably distributed across servers and improving the utilization of computing resources.

**Filter:**

In the `[Filter]` section of the configuration file, users can configure the filtering policy for the cluster server.

- `DefaultPolicy`: Default is set to 0, meaning all clients are allowed to connect to the cluster server, except those on the blacklist. If it is set to non-zero value, then all connection are blocked except for those in white lists.
- `UseBlackList`: If set to `True`, the blacklist will be enabled.
- `UseWhiteList`: If set to `True`, the whitelist will be enabled.

- `FilterListFile`: Specifies the name of the filtering configuration file for the cluster server, default is `clsfilters.ini`. Below is an example of the filter configuration file:

```
[BlackList]
# 127.0.*.* + user@machine*

[WhiteList]
# 127.0.1.2/16 - user@machine*

[ToolList]
# only tool client at server side can access by default
127.0.0.1/32

[MemWhiteList]
# clients exempt from the automatic per-task memory cap
# 192.168.1.*
# 10.0.0.0/16+batch@prod-*
```

It has four sections and each section has its own rules. In section of `BlackList`, one may add rules to block others from connection. In section of `WhiteList`, one may add rules to grant others for connection, even if the default policy is to block all connections. Only users listed in section of `ToolList` are able to connect to cluster server by Cluster Managing Tool (see below for details). Section `MemWhiteList` holds the per-task memory cap exemption rules. A client matching any rule in it is not subject to the memory cap computed automatically by the server, but is still subject to the `MemLimit` it sets itself and to the memory watermark defense. This section never affects whether a client may connect to the cluster server, and it is always active regardless of the connection filtering policy.

Specifically, rules in filter configuration have format of starting with IP address. To specify IP range, users may include wildcard (*) in IP address, or use CIDR notation, that is, an IPv4 address and its associated network prefix. In addition, a rule may include (+) or exclude (-) given user at given machine, such as `127.0.1.2/16 - user@machine`. Here, `user` refers to `username`, which can be queried by `whoami` on Linux/MacOS platform; `machine` refers to `computer name`, which can be queried by `hostname` on Linux/MacOS platform.

Note that after modifying the configuration file `clsfilters.ini` , users can use the `ResetFilters` command to reset the current rules to those in the filter configuration document. Users can also use the `WriteFilters` command to output the current rules to the filter configuration document.

**Logs:**

In the `[Logs]` section of the configuration file, users can set the log file path for the cluster server, which is by default stored in the `./logs` path under the cluster server installation directory.

### 5.1.4 Web License for Compute Cluster

Besides local cluster license above, users may use web license for compute cluster to run compute cluster service. This requires that machines running compute cluster server must have internet access. However, hardware info are not required any more. That is, users are free to deploy compute cluster servers to any cloud machine or container, as long as they have internet access. Please refer to COPT web license page for details.

Below are brief steps:

- Follow steps to register an account and apply for trial of web license for compute cluster.
- Once approved, `Web License ID` is generated for users
- On page of `API Keys` , create `Web Access Key` using given `Web License ID`

Afterwards, users edit configuration file `cls.ini` and add values of both `Web License ID` and `Web Access Key` to related keywords in section of `WLS` . For instance,

```
[WLS]
# WebServer have a default host and no need to edit in most scenarios
# Must specify WebLicenseId and WebAccesskey to trigger web licensing
WebServer =
WebLicenseId =
WebAccessKey =
WebTokenDuration = 300
```

As of now, compute cluster server talks to COPT web license page for licensing. Users are able to monitor its token usage and other information online.

### 5.1.5 Example Usage

Suppose that cluster license files exist in the same folder where the cluster executable reside. To start the COPT compute cluster, just execute the following command at any directory in Windows console, or Linux/Mac terminal.

```
./copt_cluster
```

If you see log information as follows, the COPT compute cluster has been successfully started. Server monitors any connection from COPT compute cluster clients, manages client requests in queue as well as approved clients. User may stop cluster server anytime when entering `q` or `Q`.

```
> ./copt_cluster
  [ Info] start COPT Compute Cluster, COPT v8.0.7 20240304
  [ Info] [NODE] node has been initialized
  [ Info] server started at port 7878
```

If failed to verify local cluster license, or something is wrong on remote COPT licensing server, you might see error logs as follows.

```
> ./copt_cluster
  [ Info] start COPT Compute Cluster, COPT v8.0.7 20240304
  [Error] Invalid signature in public key file
  [Error] Fail to verify local license
```

and

```
> ./copt_cluster
  [ Info] start COPT Compute Cluster, COPT v8.0.7 20240304
  [Error] Error to connect license server
  [Error] Fail to verify cluster license by server
```

## 5.2 Server Memory Management

The COPT compute cluster server manages memory through two mechanisms: the per-task memory cap and the memory watermark defense. The per-task memory cap controls the memory usage of an individual solving task; the memory watermark defense monitors the overall physical memory usage of the server and takes action when a configured watermark is reached.

### 5.2.1 Per-task memory cap

Once `MemLimitAuto` is enabled, the server computes the per-task memory cap at the start of every solve, based on the physical memory of the server, the number of sessions currently holding a token, and the related settings. The cap stays unchanged for the duration of that solve, and is recomputed at the start of the next one.

The server computes the per-task memory cap according to the following formulas:

```
Budget = PhysicalMemory * MemUsableRatio / MemInflation
Share  = Budget / SessionsHoldingToken
Cap    = min(MemOvercommit * Share, Budget / 2)
```

Where:

- `PhysicalMemory` denotes the physical memory of the server;
- `SessionsHoldingToken` denotes the number of sessions currently holding a token;
- `Budget` denotes the memory budget available to solving tasks;
- `Share` denotes the average memory share per session holding a token;
- `Cap` denotes the per-task memory cap computed automatically by the server.

When only one session holds a token, a newly started solving task may receive at most half of the memory budget as its cap. As the number of sessions holding a token increases, newly started solving tasks receive smaller caps.

Clients still set `MemLimit` through the normal COPT parameter interface; no dedicated cluster client interface is involved. If the client has also set `MemLimit`, the limit actually applied during a solve is the smaller of the client-set value and the value computed automatically by the server. A client-set `MemLimit = 0` denotes a strict zero memory limit.

After the solve finishes, the server restores the `MemLimit` originally set by the client. At the start of the next solve, the server recomputes the per-task memory cap.

`MemInflation` converts between the memory usage reported by the solver itself and the actual resident memory usage observed by the operating system. For example, setting it to `1.35` means that for every 1 byte of memory the solver accounts for, about 1.35 bytes of actual resident memory are expected. This value only illustrates the meaning of the parameter, and does not represent the default value of the program.

### 5.2.2 Memory watermark defense

Once `MemGuard` is enabled, the server checks the system physical memory usage roughly every 500 milliseconds, and takes the following actions according to the current memory usage:

- On reaching the soft watermark `MemSoft`, the server stops accepting new tasks, without affecting the tasks already running. When memory usage falls below `MemSoft - 0.03`, the server resumes accepting new tasks;
- On reaching the hard watermark `MemHard`, the server selects one solving task according to `MemEvictPolicy` and interrupts it. After interrupting a task, the server waits at least 5 seconds and waits for that task to actually stop. If memory usage is still falling, the server does not immediately interrupt the next task;
- On reaching the fatal watermark `MemFatal`, the server interrupts all solving tasks, and keeps checking until memory usage falls below that watermark.

The server interrupts a task by calling `COPT_Interrupt`, and the solve stops at its next interruptible point. If the solver has already obtained a current best feasible solution, the client can still retrieve that solution together with the corresponding interrupted status. The session the task belongs to stays connected, and may start another solve afterwards.

Only tasks that are actually solving are eligible for selection. An idle session is not selected even though it still holds its model, because calling `COPT_Interrupt` on an idle session does not release the memory of its model.

### 5.2.3 Memory cap exemptions

Users can set per-task memory cap exemption rules in the `[MemWhiteList]` section of the filter configuration file `clsfilters.ini`. A client matching any rule is not subject to the per-task memory cap computed automatically by the server, but is still subject to the `MemLimit` it sets itself and to the memory watermark defense.

`[MemWhiteList]` uses the same `IP + user@machine` rule format as the blacklist and the whitelist. This section never affects whether a client may connect to the cluster server, and it is always active regardless of the connection filtering policy. Restricting exemption rules to specific IP addresses or subnets wherever possible is recommended.

For example, the following rules may be added to `clsfilters.ini`:

```
[MemWhiteList]
192.168.1.*
10.0.0.0/16+batch@prod-*
*.*.*.*+alice@*
```

In the example above:

- `192.168.1.*`: exempts clients from that IP address range;
- `10.0.0.0/16+batch@prod-*`: exempts clients from that subnet whose user name is `batch` and whose machine name matches `prod-*`;
- `*.*.*.*+alice@*`: exempts clients whose user name is `alice`, with no restriction on IP address or machine name.

Multiple rules are independent of each other. A client is exempt from the per-task memory cap as soon as it matches any one of them. For a rule that contains an IP address, a user name and a machine name at the same time, the client has to match every part of that rule.

### 5.2.4 Compatibility with older COPT versions

The cluster server checks whether the relevant memory management features are available, according to the COPT version each task actually loads:

- The soft and fatal watermarks do not depend on a specific COPT version;
- The `longest`, `youngest` and `random` policies do not depend on a specific COPT version;
- The `largest` policy requires the COPT version loaded by the task to support `COPT_GetMemUsage`. If it is not supported, no task can be selected by that policy when the hard watermark is reached, but the fatal watermark remains effective;
- The automatic per-task memory cap requires the COPT version loaded by the task to support the `MemLimit` parameter. If it is not supported, that task is not subject to the memory cap computed automatically by the server;
- If the COPT version loaded by the task does not support retrieving the current memory usage, the peak memory usage or the memory limit, `GetJobs` displays `-` in the corresponding column.

When the cluster server has to load an older COPT version that does not support `COPT_GetMemUsage`, `MemEvictPolicy` can be set to `longest`, `youngest` or `random`.

### 5.2.5 Deployment suggestions

The following configuration can be used to further reduce the risk of exhausting the memory of the server. The values in it are for reference only, and do not represent the default values of the program:

```
[Memory]
MemGuard = true
MemSoft = 0.70
MemHard = 0.80
MemFatal = 0.90
MemLimitAuto = true
MemUsableRatio = 0.75
MemInflation = 1.50
MemOvercommit = 1.00
```

Adjusting `MemUsableRatio`, `MemInflation` and `MemOvercommit` controls the per-task memory cap computed automatically, and reserves room for the other memory overheads of the server. The specific values should be determined according to the resources of the server and the memory usage of the actual tasks.

With `MemOvercommit = 1.00`, the memory caps of all tasks together do not exceed the proportion of physical memory given by `MemUsableRatio / MemInflation`, even at full concurrency; 50% of physical memory in the example above.

If the maximum memory usage of the cluster service needs to be further restricted by the operating system, cgroup can be used, or a memory limit below the total physical memory can be set for the service through the `MemoryMax=` option of systemd.

## 5.3 Client Setup

The COPT compute cluster client can be COPT command-line, or any application which solves problems by COPT API, such as COPT cpp/java/csharp/python interface. The COPT compute cluster service is a better approach in terms of flexibility and efficiency. Any COPT compute cluster client can legally run Cardinal Optimizer without local license.

### 5.3.1 Configuration

#### Via the configuration file

Before running COPT as cluster client, please make sure that you have installed COPT locally. For details, please refer to *How to install Cardinal Optimizer*. Users can skip obtaining local licenses by adding a cluster configuration file `client.ini` .

Below is a typical configuration file, `client.ini`, of COPT compute cluster client.

```
Cluster = 192.168.1.11
Port = 7878
WaitTime = 600
Passwd =
```

As configured above, COPT compute cluster client tries to connect to `192.168.1.11` at port `7878` with waiting time in queue up to 600 seconds. Here, the default value of `Cluster` is localhost. `WaitTime` (or `QueueTime` ) is set to 0 if empty or not specified. Specifically, empty `WaitTime` means client does not wait and should quit immediately, if the COPT compute cluster have no more token available. `Port` default number is 7878. It must be great than zero if specified and should be the same as that specified in cluster configuration file `cls.ini`. Note that keywords in the configuration file are case insensitive.

In addition, users can set the password for connecting the remote server through `Passwd`. The `Priority` can be used to set the priority of cluster optimization jobs. Possible values range from 0 to 99, with

higher values indicating higher priority.If jobs are queued, the priority setting will ensure that the next task is processed first, but it will not affect tasks that are already running.

To run as a COPT compute cluster client, an application must have configuration file, `client.ini`, in one of the following three locations, that is, current working directory, environment directory by `COPT_LICENSE_DIR` and binary directory where COPT executable resides.

By design, COPT application reads local license files instead of `client.ini`, if they both exist in the same location. However, if local license files are under the environment directory, to connect to cluster server, user could simply add a configuration file, `client.ini`, under the current working directory (different from the environment directory).

If a COPT application calls COPT API to solve problems, such as COPT python interface, license is checked as soon as COPT environment object is created. If there only exists proper configuration file, `client.ini`, the application works as a COPT compute cluster client and obtains token to offload optimization jobs. As soon as COPT environment object is destroyed, the COPT compute cluster server is notified to release token and thus to approve more requests waiting in queue.

#### Via API Functions

In addition to the method using the `client.ini` file mentioned above, users can also configure the client in their code through API functions. Taking the COPT Python interface as an example, the corresponding class is *EnvrConfig Class*, and similar approaches apply to other programming languages. The configuration is as follows:

```
# Set client configuration parameters
envconfig.set(COPT.CLIENT_CLUSTER, "192.168.1.11")
envconfig.set(COPT.CLIENT_PORT, "7878")
envconfig.set(COPT.CLIENT_WAITTIME, "600")
```

### 5.3.2 High Availability

When there are multiple cluster servers, the client can achieve high availability by configuring multiple Cluster server IP addresses in the `Cluster` field of `client.ini`, as shown in the following table:

```
Cluster = 192.168.1.11; 192.168.1.22; 192.168.1.33
Port = 7878
WaitTime = 600
Passwd =
```

According to the above configuration, the first IP in the `Cluster` field is the root node IP of the cluster. If the port of other child nodes of the cluster is not `7878`, it could be added after the IP address. The above configuration file indicates that the client will first try to connect to the root node `192.168.1.11:7878` . If the root node is not available, it will try to connect to the child node of `192.168.1.22:7878` , and so on, until the connection with the cluster server is successful.

If a cluster node fails when performing a computing task, the client will reallocate the available cluster nodes according to the backup nodes in the configuration file. This can enable the cluster service to have a certain disaster recovery capability, thereby improving the high availability of the COPT computing cluster service.

### 5.3.3 Example Usage

Suppose that we've set configuration file `client.ini` properly and have no local license, below is an example of connecting to cluster server by COPT command-line tool `copt_cmd`. Execute the following command in Windows console, or Linux/Mac terminal.

```
copt_cmd
```

If you see log information as follows, the COPT compute cluster client, `copt_cmd`, has connected to cluster server successfully. COPT command-line tool is ready to do modeling locally and then offload optimization jobs to server.

```
> copt_cmd
  Cardinal Optimizer v8.0.7. Build date Mar 04 2024
  Copyright Cardinal Operations 2025. All Rights Reserved

  [ Info] initialize cluster client with ./client.ini

  [ Info] wait for server in 0 / 39 secs
  [ Info] connecting to cluster server 192.168.1.11:7878
COPT>
```

If you see log information as follows, the COPT compute cluster client, `copt_cmd`, has connected to cluster server. However, due to limited number of tokens, it waits in queue of size 5, until timeout.

```
> copt_cmd
  Cardinal Optimizer v8.0.7. Build date Mar 04 2024
  Copyright Cardinal Operations 2025. All Rights Reserved

  [ Info] initialize cluster client with ./client.ini

  [ Info] wait for server in 0 / 39 secs
  [ Info] connecting to cluster server 192.168.1.11:7878

  [ Warn] wait in queue of size 5
  [ Info] wait for license in  2 / 39 secs
  [ Info] wait for license in  4 / 39 secs
  [ Info] wait for license in  6 / 39 secs
  [ Info] wait for license in  8 / 39 secs
  [ Info] wait for license in 10 / 39 secs
  [ Info] wait for license in 20 / 39 secs
  [ Info] wait for license in 30 / 39 secs
  [Error] timeout at waiting for server approval
  [Error] Fail to initialize copt command-line tool
```

If you see log information as follows, the COPT compute cluster client, `copt_cmd`, has connected to cluster server. But it refused to wait in queue, as Queuetime was set to 0. Therefore, client quits with error immediately.

```
> copt_cmd
  Cardinal Optimizer v8.0.7. Build date Mar 04 2024
  Copyright Cardinal Operations 2025. All Rights Reserved

  [ Info] initialize cluster client with ./client.ini

  [ Info] wait for server in 0 / 9 secs
  [ Info] connecting to cluster server 192.168.1.11:7878
  [ Warn] server error: "no more token available", code = 129
  [Error] Fail to initialize copt command-line tool
```

If you see log information as follows, the COPT compute cluster client, `copt_cmd`, fails to connect to

cluster server. Finally, client quits after timeout.

```
> copt_cmd
  Cardinal Optimizer v8.0.7. Build date Mar 04 2024
  Copyright Cardinal Operations 2025. All Rights Reserved

  [ Info] initialize cluster client with ./client.ini

  [ Info] wait for server in 0 / 39 secs
  [ Info] connecting to cluster server 192.168.1.11:7878
  [ Info] wait for license in  2 / 39 secs
  [ Info] wait for license in  4 / 39 secs
  [ Info] wait for license in  6 / 39 secs
  [ Info] wait for license in  8 / 39 secs
  [ Info] wait for license in 10 / 39 secs
  [ Info] wait for license in 20 / 39 secs
  [ Info] wait for license in 30 / 39 secs
  [Error] timeout at waiting for server approval
  [Error] Fail to initialize copt command-line tool
```

In addition, users can start the COPT command-line tool on the client side to connect to the specified cluster server IP address as follows:

```
> copt_cmd -cluster <ip>
```

## 5.4 COPT Cluster Managing Tool

COPT cluster service ships with a tool `copt_clstool`, for retrieving information and tune parameters of cluster servers on fly.

### 5.4.1 Tool Usage

Execute the following command in Windows console, Linux or MacOS terminal:

```
> ./copt_clstool
```

Below displays help messages of this tool:

```
> ./copt_clstool
  COPT Cluster Managing Tool

  copt_clstool [-s server ip] [-p port] [-x passwd] command <param>

  commands are:   addblackrule <127.0.0.1/20[-user@machine]>
                  addwhiterule <127.0.*.*[+user@machine]>
                  addmemexempt <127.0.*.*[+user@machine]>
                  delmemexempt <127.0.*.*[+user@machine]>
                  getfilters
                  getinfo
                  getnodes
                  getjobs
                  interrupt
                  reload
                  resetfilters
                  setparent <xxx:7878>
                  setpasswd <xxx>
```

(continues on next page)

(continued from previous page)

```
                settoken  <num>
                setmemguard <on|off>
                setmemlimitauto <on|off>
                setmemevictpolicy <largest|longest|youngest|random>
                toggleblackrule <n-th>
                togglewhiterule <n-th>
                writefilters
```

If the `-s` and `-p` option are present, tool connects to cluster server with given server IP and port. Otherwise, tool connections to localhost and default port 7878. If cluster server sets a password, tool must provide password string after the `-x` option.

This tool provides the following commands:

- `AddBlackRule`: Add a new rule in black filters. each rule has format starting with non-empty IP address, which may have wildcard to match IPs in the scope. In addition, it is optional to be followed by including (+) or excluding (-) user name at machine name.
- `AddWhiteRule`: Add a new rule in white filters. Note that a white rule has the same format as a black rule.
- `AddMemExempt`: Add a per-task memory cap exemption rule. A client matching the rule is not subject to the per-task memory cap computed automatically by the server, but is still subject to the `MemLimit` it sets itself and to the memory watermark defense. This command modifies the rules in memory and takes effect immediately.
- `DelMemExempt`: Delete the given per-task memory cap exemption rule. This command modifies the rules in memory and takes effect immediately.
- `GetFilters`: Get the current black filters, white filters, tool filters and per-task memory cap exemption rules. Each rule in the black and white filters carries a sequence number, which is the parameter for command ToggleBlackRule and ToggleWhiteRule respectively.
- `GetInfo`: Get general information of cluster server, including token usage, connected clients, all COPT versions in support, along with the state of the memory watermark defense, the memory cap a next solving task is expected to receive, and the clients currently exempt from the memory cap.
- `GetNodes`: Get information of nodes in cluster, including parent address and status, all children nodes.
- `Reload`: Reload available token information of all child nodes, in case it is not consistent for various reasons.
- `GetJobs`: Retrieves the information of the tasks on the current server, including the task ID (TID), runtime (in seconds), client ID, current memory usage, peak memory usage and memory limit. The memory related information is given in MB. If the COPT version loaded by the task does not support retrieving the corresponding memory information, - is displayed in that column.
- `Interrupt`: Terminates the specified task (TID) on the current server (cluster server). After executing this command, the task running on the client will be stopped, and the optimization status will return `stopped (user interrupt)`.
- `ResetFilters`: Reset filter lists in memory to those on filter config file.
- `SetParent`: Change parent node address on fly and then connecting to new parent. In this way, it avoids `draining` operation when stopping a node for maintenance purpose.
- `SetPasswd`: Update password of target cluster server on fly.
- `SetToken`: Change token number of target cluster server on fly.
- `SetMemGuard`: Enable or disable the memory watermark defense. The available values are `on` and `off`.

- `SetMemLimitAuto`: Enable or disable the automatic per-task memory cap. The available values are `on` and `off`. The modified setting applies to the solving tasks started afterwards.
- `SetMemEvictPolicy`: Set the policy for selecting the task to interrupt when the hard watermark is reached. The available values are `largest`, `longest`, `youngest` and `random`.
- `ToggleBlackRule`: Toggle between enabling and disabling a black rule, given its sequence number by GetFilters.
- `ToggleWhiteRule`: Toggle between enabling and disabling a white rule, given its sequence number by GetFilters.
- `WriteFilters`: Write all filter rules in memory to the filter config file, including the black filters, white filters, tool filters and per-task memory cap exemption rules.

Changes made to the memory management settings through `SetMemGuard`, `SetMemLimitAuto` and `SetMemEvictPolicy` only apply while the current server is running, and are not written back to `cls.ini`. After the server restarts, the settings in `cls.ini` are used again.

### 5.4.2 Example Usage

The following command lists general information on local machine.

```
> ./copt_clstool GetInfo

[ Info] COPT Cluster Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to localhost:7878
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] general info
  # of available tokens is 3 / 3, queue size is 0
  # of active clients is 0
  # of installed COPT versions is 1
    COPT v8.0.7  Physical Memory: 71.8% used, Defense Status: [OK] (soft 80% / hard 92
↪% / fatal 97%)
  Per-task Memory Cap: 5461 MB (by demand with N=3)
  Memory exempt clients: 127.0.0.1, 127.0.0.*
```

The following command lists the tasks on the server together with their memory information:

```
./copt_clstool -s <ip> -p <port> getjobs
```

The output contains the following fields:

```
ThreadID  Duration(s)  MemUsage(MB)  MemPeak(MB)  MemLimit(MB)  ClientID
```

Where:

- `ThreadID`: the task ID;
- `Duration(s)`: the runtime of the task, in seconds;
- `MemUsage(MB)`: the current memory usage, in MB;
- `MemPeak(MB)`: the peak memory usage, in MB;
- `MemLimit(MB)`: the memory limit currently set on the problem, in MB. During a solve this is the limit actually enforced; otherwise it is the value set by the client itself;
- `ClientID`: the client ID.

`MemUsage` and `MemPeak` both come from the memory accounting of the solver itself, and may therefore differ from the actual resident memory usage observed by the operating system. For the conversion between the two, refer to `MemInflation`.

If the COPT version loaded by the task does not support retrieving one of these pieces of memory information, - is displayed in the corresponding column.

The following commands show how to enable the memory watermark defense and the automatic per-task memory cap while the server is running, and set the policy for selecting the task to interrupt when the hard watermark is reached to `largest`:

```
./copt_clstool -s <ip> -p <port> setmemguard on
./copt_clstool -s <ip> -p <port> setmemlimitauto on
./copt_clstool -s <ip> -p <port> setmemevictpolicy largest
```

To disable the corresponding features, execute respectively:

```
./copt_clstool -s <ip> -p <port> setmemguard off
./copt_clstool -s <ip> -p <port> setmemlimitauto off
```

`setmemevictpolicy` supports the four policies `largest`, `longest`, `youngest` and `random`.

Changes made by these commands are not written back to `cls.ini`. After the server restarts, the settings in `cls.ini` are used again.

To run managing tool on other machine, its IP should be added to a rule in `ToolList` section in filter configuration file `clsfilters.ini`. The following command from other machine lists cluster information of server 192.168.1.11.

```
> ./copt_clstool -s 192.168.1.11 GetNodes

[ Info] COPT Cluster Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7878
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] node info
  [Parent] (null):7878 (Lost)
  [Child] Node_192.168.1.12:7878_N0001, v2.0=3
  Total num of child nodes is 1
```

The following command changes token number of server 192.168.1.11 from 3 to 0.

```
> ./copt_clstool -s 192.168.1.11 SetToken 0

[ Info] COPT Cluster Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7878
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] total token was 3 and now set to 0
```

The following command shows all filter lists of server 192.168.1.11, including those in BlackList section, WhiteList section, ToolList section and MemWhiteList section.

```
> ./copt_clstool -s 192.168.1.11 GetFilters

[ Info] COPT Cluster Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] filters info
[BlackList]

[WhiteList]

[ToolList]
  [1]  127.0.0.1

[MemWhiteList]
```

The following command added user of IP 192.168.3.13 to black list.

```
> ./copt_clstool -s 192.168.1.11 AddBlackRule 192.168.3.133

[ Info] COPT Cluster Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] server added new black rule (succeeded)
```

The following command shows that a new rule in BlackList section is added.

```
> ./copt_clstool -s 192.168.1.11 GetFilters

[ Info] COPT Cluster Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] filters info
[BlackList]
  [1] 192.168.3.133

[WhiteList]

[ToolList]
  [1]  127.0.0.1

[MemWhiteList]
```

The following command disable a rule in BlackList section.

```
> ./copt_clstool -s 192.168.1.11 ToggleBlackRule 1

[ Info] COPT Cluster Managing Tool, COPT v8.0.7 20240304
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] server toggle black rule [1] (succeeded)
```

The following command adds a per-task memory cap exemption rule for the given client. The server IP, the port and the rule in the command should be replaced according to the actual deployment.

```
./copt_clstool -s <ip> -p <port> addmemexempt '127.0.*.*+user@machine*'
```

After adding the rule, the current filter rules can be listed with:

```
./copt_clstool -s <ip> -p <port> getfilters
```

To write the filter rules currently in memory back to the filter configuration file, execute:

```
./copt_clstool -s <ip> -p <port> writefilters
```

The following command deletes the per-task memory cap exemption rule added above:

```
./copt_clstool -s <ip> -p <port> delmemexempt '127.0.*.*+user@machine*'
```

Changes made to the rules in memory by `addmemexempt` and `delmemexempt` take effect immediately; `writefilters` writes all filter rules, including the memory cap exemption rules, back to the configuration file.

In the commands above, `<ip>` and `<port>` denote the IP address and the port of the cluster server. They should be replaced with the actual values, and the angle brackets should not be kept.

## 5.5 Running as service

To run COPT compute cluster server as a system service, follow steps described in `readme.txt` under `cluster` folder, and set config file `copt_cluster.service` properly.

Below is `readme.txt`, which lists installing steps in both Linux and MacOS platforms.

```
[Linux] To run copt_cluster as a service with systemd

Add a systemd file
    cp copt_cluster.service to /lib/systemd/system/
    sudo systemctl daemon-reload

Enable new service
    sudo systemctl start copt_cluster.service
    or
    sudo systemctl enable copt_cluster.service

Restart service
    sudo systemctl restart copt_cluster.service

Stop service
    sudo systemctl stop copt_cluster.service
    or
    sudo systemctl disable copt_cluster.service

Verify service is running
    sudo systemctl status copt_cluster.service

[MacOS] To run copt_cluster as a service with launchctrl

Add a plist file
    cp copt_cluster.plist to /Library/LaunchAgents as current user
    or
    cp copt_cluster.plist to /Library/LaunchDaemons with the key 'UserName'

Enable new service
    sudo launchctl load -w /Library/LaunchAgents/copt_cluster.plist
    or
    sudo launchctl load -w /Library/LaunchDaemons/copt_cluster.plist

Stop service
    sudo launchctl unload -w /Library/LaunchAgents/copt_cluster.plist
    or
    sudo launchctl unload -w /Library/LaunchDaemons/copt_cluster.plist

Verify service is running
    sudo launchctl list solver.copt.cluster
```

### 5.5.1 Linux

Below are the detailed steps for how to run COPT compute cluster server as a system service in Linux platform.

For instance, assume that COPT remote service is installed under `'/home/eleven'`. In your terminal, type the following command to enter the root directory of cluster service.

```
cd /home/eleven/copt_remote80/cluster
```

modify template of the service config file `copt_cluster.service` in text format:

```
[Unit]
Description=COPT Compute Cluster Server

[Service]
WorkingDirectory=/path/to/service
ExecStart=/path/to/service/copt_cluster
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

That is, update template path in keyword `WorkingDirectory` and `ExecStart` to actual path where the cluster service exits.

```
[Unit]
Description=COPT Compute Cluster Server

[Service]
WorkingDirectory=/home/eleven/copt_remote80/cluster
ExecStart=/home/eleven/copt_remote80/cluster/copt_cluster
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

Afterwards, copy `copt_cluster.service` to system service folder `/lib/systemd/system/` (see below).

```
sudo cp copt_cluster.service /lib/systemd/system/
```

The following command may be needed if you add or update service config file. It is not needed if service unit has been loaded before.

```
sudo systemctl daemon-reload
```

The following command starts the new cluster service.

```
sudo systemctl start copt_cluster.service
```

To verify the cluster service is actually running, type the following command

```
sudo systemctl status copt_cluster.service
```

If you see logs similar to below, COPT compute cluster server is running successfully as a system service.

```
copt_cluster.service - COPT Cluster Server
Loaded: loaded (/lib/systemd/system/copt_cluster.service; enabled; vendor preset:␣
↪enabled)
Active: active (running) since Sat 2021-08-28 11:46:10 CST; 3s ago
```

```
Main PID: 3054 (copt_cluster)
    Tasks: 6 (limit: 4915)
CGroup: /system.slice/copt_cluster.service
          └─3054 /home/eleven/copt_remote80/cluster/copt_cluster

eleven-ubuntu systemd[1]: Started COPT Cluster Server.
eleven-ubuntu COPTCLS[3054]: LWS: 4.1.4-b2011a00, loglevel 1039
eleven-ubuntu COPTCLS[3054]: NET CLI SRV H1 H2 WS IPv6-absent
eleven-ubuntu COPTCLS[3054]: server started at port 7878
eleven-ubuntu COPTCLS[3054]: LWS: 4.1.4-b2011a00, loglevel 1039
eleven-ubuntu COPTCLS[3054]: NET CLI SRV H1 H2 WS IPv6-absent
eleven-ubuntu COPTCLS[3054]: [NODE] node has been initialized
```

To stop the cluster service, type the following command

```
sudo systemctl stop copt_cluster.service
```

### 5.5.2 MacOS

Below are the detailed steps for how to run COPT Compute Cluster server as a system service in MacOS platform.

For instance, assume that COPT remote service is installed under `"/Applications"`. In your terminal, type the following command to enter the root directory of cluster service.

```
cd /Applications/copt_remote80/cluster
```

modify template of the service config file `copt_cluster.plist` in xml format:

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>solver.copt.cluster</string>
        <key>Program</key>
        <string>/path/to/service/copt_cluster</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

That is, update template path in `Program` tag to actual path where the cluster service exits.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>solver.copt.cluster</string>
        <key>Program</key>
        <string>/Applications/copt_remote80/cluster/copt_cluster</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

Afterwards, copy `copt_cluster.plist` to system service folder `/Library/LaunchAgents` (see below).

```
sudo cp copt_cluster.plist /Library/LaunchAgents
```

The following command starts the new cluster service.

```
sudo launchctl load -w /Library/LaunchAgents/copt_cluster.plist
```

To verify the cluster service is actually running, type the following command

```
sudo launchctl list solver.copt.cluster
```

If you see logs similar to below, COPT compute cluster server is running successfully as a system service.

```
{
    "LimitLoadToSessionType" = "System";
    "Label" = "solver.copt.cluster";
    "OnDemand" = false;
    "LastExitStatus" = 0;
    "PID" = 16406;
    "Program" = "/Applications/copt_remote80/cluster/copt_cluster";
};
```

To stop the cluster service, type the following command

```
sudo launchctl unload -w /Library/LaunchAgents/copt_cluster.plist
```

If the cluster service should be run by a specific user, add `UserName` tag to config file. Below adds a user `eleven`, who has privilege to run the cluster service.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>solver.copt.cluster</string>
        <key>Program</key>
        <string>/Applications/copt_remote80/cluster/copt_cluster</string>
        <key>UserName</key>
        <string>eleven</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

Then copy new `copt_cluster.plist` to system service folder `/Library/LaunchDaemons` (see below).

```
sudo cp copt_cluster.plist /Library/LaunchDaemons
```

The following command starts the new cluster service.

```
sudo launchctl load -w /Library/LaunchDaemons/copt_cluster.plist
```

To stop the cluster service, type the following command

```
sudo launchctl unload -w /Library/LaunchDaemons/copt_cluster.plist
```

# Chapter 6

# COPT Web Licensing Service

COPT's Web Licensing Service provides users with remote licensing services. Regardless of whether the client is located in the cloud or in a container, as long as it can access the Internet through the HTTPS protocol, it can obtain the Token from COPT's Web License server to run COPT without binding any hardware information.

Therefore, compared with traditional authorization methods, Web License is not limited to fixed hardware environments, flexibly supporting multiple users, and is suitable for container deployment (such as Docker) and cloud deployment (Internet environment is required). Its characteristics are summarized as follows:

1. It does not bind to any hardware information, flexibly supporting scenarios such as cloud and container deployment (hardware does not need to be fixed);
2. No version restrictions; cross-version usage is supported;
3. At least one machine needs to be connected to the Internet (remote communication with the web server);
4. Provide a COPT web license page for users to obtain and manage licenses by themselves, which is user-friendly and convenient.

At the same time, corresponding to the traditional authorization method, Web License also includes three subcategories: Web License, Web License for Floating Server, and Web License for Compute Cluster.

**Web License**

It supports the deployment of servers running COPT in the cloud development environment (without binding any machine hardware information), and supports multiple modeling and solving tasks on the server at the same time.

**Web License for Floating Server**

It supports deploying a floating token server in a cloud development environment (the server needs to connect to the Internet and obtain remote authorization via Web License), and then using the floating token server to authorize other machines (clients) within the local network to run COPT.

**Web License for Compute Cluster**

It supports setting up one or more computing cluster servers in the cloud development environment. Modeling can be performed on the local machine (client) within the local network while optimization can be conducted on the remote cluster server machine (server), so that the powerful computing resources of the server can be efficiently utilized.

Users need to register on the COPT web license page first. After logging in, they can **apply directly on the page** to obtain the above three types of Web licenses, create new API Keys, obtain license files, and manage token occupancy and usage.

For more information on Web License or installation and usage tutorials, please refer to the Web license help documentation. If you have any further questions about web licensing, please contact us as follows:

Table 6.1: Contact information

| Email | Description |
|---|---|
| coptsales@shanshu.ai | Business Support |
| coptsupport@shanshu.ai | Technical Support |

# Chapter 7

# COPT Quick Start

## 7.1 C Interface

This chapter illustrate the use of C interface of Cardinal Optimizer through a simple C example. The problem to solve is shown in Eq. 7.1:

$$
\begin{aligned}
&\text{Maximize:} \\
&\qquad 1.2x + 1.8y + 2.1z \\
&\text{Subject to:} \\
&\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\
&\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\
&\text{Bounds:} \\
&\qquad 0.1 \leq x \leq 0.6 \\
&\qquad 0.2 \leq y \leq 1.5 \\
&\qquad 0.3 \leq z \leq 2.8
\end{aligned}
\tag{7.1}
$$

### 7.1.1 Example details

The source code for solving the above problem using C API of Cardinal Optimizer is shown in Listing 7.1:

Listing 7.1: `lp_ex1.c`

```c
/*
 * This file is part of the Cardinal Optimizer, all rights reserved.
 */

/*
 * The problem to solve:
 *
 *  Maximize:
 *    1.2 x + 1.8 y + 2.1 z
 *
 *  Subject to:
 *    1.5 x + 1.2 y + 1.8 z <= 2.6
 *    0.8 x + 0.6 y + 0.9 z >= 1.2
 *
 *  where:
 *    0.1 <= x <= 0.6
```

(continues on next page)

(continued from previous page)

```
 *    0.2 <= y <= 1.5
 *    0.3 <= z <= 2.8
 */

#include "copt.h"

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  int errcode = 0;

  copt_env* env = NULL;
  copt_prob* prob = NULL;

  // Create COPT environment
  errcode = COPT_CreateEnv(&env);
  if (errcode)
    goto COPT_EXIT;

  // Create COPT problem
  errcode = COPT_CreateProb(env, &prob);
  if (errcode)
    goto COPT_EXIT;

  /*
   * Add variables
   *
   *   obj: 1.2 C0 + 1.8 C1 + 2.1 C2
   *
   *   var:
   *     0.1 <= C0 <= 0.6
   *     0.2 <= C1 <= 1.5
   *     0.3 <= C2 <= 2.8
   *
   */
  int ncol = 3;
  double colcost[] = {1.2, 1.8, 2.1};
  double collb[] = {0.1, 0.2, 0.3};
  double colub[] = {0.6, 1.5, 1.8};

  errcode = COPT_AddCols(prob, ncol, colcost, NULL, NULL, NULL, NULL, NULL, collb,␣
↪colub, NULL);
  if (errcode)
    goto COPT_EXIT;

  /*
   * Add constraints
   *
   *   r0: 1.5 C0 + 1.2 C1 + 1.8 C2 <= 2.6
   *   r1: 0.8 C0 + 0.6 C1 + 0.9 C2 >= 1.2
   */
  int nrow = 2;
  int rowbeg[] = {0, 3};
  int rowcnt[] = {3, 3};
```

(continues on next page)

(continued from previous page)

```
  int rowind[] = {0, 1, 2, 0, 1, 2};
  double rowelem[] = {1.5, 1.2, 1.8, 0.8, 0.6, 0.9};
  char rowsen[] = {COPT_LESS_EQUAL, COPT_GREATER_EQUAL};
  double rowrhs[] = {2.6, 1.2};

  errcode = COPT_AddRows(prob, nrow, rowbeg, rowcnt, rowind, rowelem, rowsen, rowrhs,␣
↪NULL, NULL);
  if (errcode)
    goto COPT_EXIT;

  // Set parameters and attributes
  errcode = COPT_SetDblParam(prob, COPT_DBLPARAM_TIMELIMIT, 10);
  if (errcode)
    goto COPT_EXIT;
  errcode = COPT_SetObjSense(prob, COPT_MAXIMIZE);
  if (errcode)
    goto COPT_EXIT;

  // Solve problem
  errcode = COPT_SolveLp(prob);
  if (errcode)
    goto COPT_EXIT;

  // Analyze solution
  int lpstat = COPT_STATUS_UNSTARTED;
  double lpobjval;
  double* lpsol = NULL;
  int* colstat = NULL;

  errcode = COPT_GetIntAttr(prob, COPT_INTATTR_STATUS, &lpstat);
  if (errcode)
    goto COPT_EXIT;

  if (lpstat == COPT_STATUS_OPTIMAL)
  {
    lpsol = (double*)malloc(ncol * sizeof(double));
    colstat = (int*)malloc(ncol * sizeof(int));

    errcode = COPT_GetLpSolution(prob, lpsol, NULL, NULL, NULL);
    if (errcode)
      goto COPT_EXIT;

    errcode = COPT_GetBasis(prob, colstat, NULL);
    if (errcode)
      goto COPT_EXIT;

    errcode = COPT_GetDblAttr(prob, COPT_DBLATTR_OBJVAL, &lpobjval);
    if (errcode)
      goto COPT_EXIT;

    printf("\nObjective value: %.6f\n", lpobjval);

    printf("Variable solution: \n");
    for (int i = 0; i < ncol; ++i)
      printf("  x[%d] = %.6f\n", i, lpsol[i]);

```

(continues on next page)

(continued from previous page)

```
    printf("Variable basis status: \n");
    for (int i = 0; i < ncol; ++i)
      printf("  x[%d]: %d\n", i, colstat[i]);

    free(lpsol);
    free(colstat);
  }

  // Write problem, solution and modified parameters to files
  errcode = COPT_WriteMps(prob, "lp_ex1.mps");
  if (errcode)
    goto COPT_EXIT;
  errcode = COPT_WriteBasis(prob, "lp_ex1.bas");
  if (errcode)
    goto COPT_EXIT;
  errcode = COPT_WriteSol(prob, "lp_ex1.sol");
  if (errcode)
    goto COPT_EXIT;
  errcode = COPT_WriteParam(prob, "lp_ex1.par");
  if (errcode)
    goto COPT_EXIT;

  // Error handling
COPT_EXIT:
  if (errcode)
  {
    char errmsg[COPT_BUFFSIZE];

    COPT_GetRetcodeMsg(errcode, errmsg, COPT_BUFFSIZE);
    printf("ERROR %d: %s\n", errcode, errmsg);

    return 0;
  }

  // Delete problem and environment
  COPT_DeleteProb(&prob);

  COPT_DeleteEnv(&env);

  return 0;
}
```

We will explain how to use the C API step by step based on code above, please refer to *C API Reference* for detailed usage of C API.

### Creating the environment

To solve any problem with Cardinal Optimizer, users are required to create optimization environment first, and check if it was created successfully by checking the return value:

```
  // Create COPT environment
  errcode = COPT_CreateEnv(&env);
  if (errcode)
    goto COPT_EXIT;
```

If non-zero value was returned, it will jump to error reporting code block for detailed information and

exit.

### Creating the problem

Once the optimization environment was successfully created, users will need to create problem then, the problem is the main structure that consists of variables, constraints etc. Users need to check the return value too.

```
// Create COPT problem
errcode = COPT_CreateProb(env, &prob);
if (errcode)
  goto COPT_EXIT;
```

If non-zero value was returned, it will jump to error reporting code block for detailed information and exit.

### Adding variables

For linear problem, C API allows users to specify costs of variables in objective, and lower and upper bound simultaneously. For the problem above, we use code below to create variables:

```
/*
 * Add variables
 *
 *   obj: 1.2 C0 + 1.8 C1 + 2.1 C2
 *
 *   var:
 *     0.1 <= C0 <= 0.6
 *     0.2 <= C1 <= 1.5
 *     0.3 <= C2 <= 2.8
 *
 */
int ncol = 3;
double colcost[] = {1.2, 1.8, 2.1};
double collb[] = {0.1, 0.2, 0.3};
double colub[] = {0.6, 1.5, 1.8};

errcode = COPT_AddCols(prob, ncol, colcost, NULL, NULL, NULL, NULL, NULL, collb,␣
↪colub, NULL);
if (errcode)
  goto COPT_EXIT;
```

The argument `ncol` specify that the number of variables to create is 3, while the argument `colcost`, `collb` and `colub` specify the costs in objective, lower and upper bound respectively. Regarding other arguments of `COPT_AddCols` for specifying variables types and names, we just pass `NULL` to them, which means all variables are continuous and names are automatically generated by the Cardinal Optimizer. For the remaining arguments, we passed `NULL` too for further action.

Similarly, if non-zero value was returned, it will jump to error reporting code block for detailed information and exit.

### Adding constraints

The next step to do after adding variables successfully is to add constraints to problem. For the problem above, the implementation is shown below:

```
/*
 * Add constraints
 *
 *   r0: 1.5 C0 + 1.2 C1 + 1.8 C2 <= 2.6
 *   r1: 0.8 C0 + 0.6 C1 + 0.9 C2 >= 1.2
 */
int nrow = 2;
int rowbeg[] = {0, 3};
int rowcnt[] = {3, 3};
int rowind[] = {0, 1, 2, 0, 1, 2};
double rowelem[] = {1.5, 1.2, 1.8, 0.8, 0.6, 0.9};
char rowsen[] = {COPT_LESS_EQUAL, COPT_GREATER_EQUAL};
double rowrhs[] = {2.6, 1.2};

errcode = COPT_AddRows(prob, nrow, rowbeg, rowcnt, rowind, rowelem, rowsen, rowrhs,␣
↪NULL, NULL);
if (errcode)
  goto COPT_EXIT;
```

The argument `nrow` specifies that the number of constraints to create is 2, while argument `rowbeg`, `rowcnt`, `rowind` and `rowelem` define the coefficient matrix in CSR format. The argument `rowsen` represents the sense of constraints, while argument `rowrhs` specifies the right hand side of constraints. For remaining arguments in `COPT_AddRows`, we simply pass `NULL` to them.

If the return value is non-zero, then it jump to error reporting code block for detailed information and exit.

### Setting parameters and attributes

Users are allowed to set parameters and attributes of problem before solving. For example, to set the time limit to 10 seconds, and to set the optimization direction to maximization, the code is shown below:

```
// Set parameters and attributes
errcode = COPT_SetDblParam(prob, COPT_DBLPARAM_TIMELIMIT, 10);
if (errcode)
  goto COPT_EXIT;
errcode = COPT_SetObjSense(prob, COPT_MAXIMIZE);
if (errcode)
  goto COPT_EXIT;
```

If non-zero value was returned, then it will jump to error reporting code block for detailed information and exit.

### Solve the problem

The next step to do is to solve the problem using code below:

```
// Solve problem
errcode = COPT_SolveLp(prob);
if (errcode)
  goto COPT_EXIT;
```

Non-zero return value indicates unsuccessful solve and jump to error reporting code block for detailed information and exit.

### Analyze the solution

Once the solving process was finished, check the solution status first. If it claimed to have found the optimal solution, then use code below to obtain objective value, variables' solution and basis status:

```
// Analyze solution
int lpstat = COPT_STATUS_UNSTARTED;
double lpobjval;
double* lpsol = NULL;
int* colstat = NULL;

errcode = COPT_GetIntAttr(prob, COPT_INTATTR_STATUS, &lpstat);
if (errcode)
  goto COPT_EXIT;

if (lpstat == COPT_STATUS_OPTIMAL)
{
  lpsol = (double*)malloc(ncol * sizeof(double));
  colstat = (int*)malloc(ncol * sizeof(int));

  errcode = COPT_GetLpSolution(prob, lpsol, NULL, NULL, NULL);
  if (errcode)
    goto COPT_EXIT;

  errcode = COPT_GetBasis(prob, colstat, NULL);
  if (errcode)
    goto COPT_EXIT;

  errcode = COPT_GetDblAttr(prob, COPT_DBLATTR_OBJVAL, &lpobjval);
  if (errcode)
    goto COPT_EXIT;

  printf("\nObjective value: %.6f\n", lpobjval);

  printf("Variable solution: \n");
  for (int i = 0; i < ncol; ++i)
    printf("  x[%d] = %.6f\n", i, lpsol[i]);

  printf("Variable basis status: \n");
  for (int i = 0; i < ncol; ++i)
    printf("  x[%d]: %d\n", i, colstat[i]);

  free(lpsol);
  free(colstat);
}
```

#### Write problem and solution

Users are allowed not only to save the problem to solve to standard MPS file, but also the solution, basis status and modified parameters to files:

```
  // Write problem, solution and modified parameters to files
  errcode = COPT_WriteMps(prob, "lp_ex1.mps");
  if (errcode)
    goto COPT_EXIT;
  errcode = COPT_WriteBasis(prob, "lp_ex1.bas");
  if (errcode)
    goto COPT_EXIT;
  errcode = COPT_WriteSol(prob, "lp_ex1.sol");
  if (errcode)
    goto COPT_EXIT;
  errcode = COPT_WriteParam(prob, "lp_ex1.par");
  if (errcode)
    goto COPT_EXIT;
```

#### Error handling

The error handling block report error code and message by checking if the return value was non-zero:

```
  // Error handling
COPT_EXIT:
  if (errcode)
  {
    char errmsg[COPT_BUFFSIZE];

    COPT_GetRetcodeMsg(errcode, errmsg, COPT_BUFFSIZE);
    printf("ERROR %d: %s\n", errcode, errmsg);

    return 0;
  }
```

#### Delete environment and problem

Before exiting, delete problem and environment respectively:

```
  // Delete problem and environment
  COPT_DeleteProb(&prob);

  COPT_DeleteEnv(&env);
```

### 7.1.2 Build and run

To ease the work for running the example for users on different operating systems, we provide Visual Studio project and Makefile for Windows, Linux and MacOS respectively, details are shown below.

#### Windows

For users on Windows platform, we provide Visual Studio project, all users are required to install Visual Studio 2017 beforehand. Assume that the installation directory is: `'<instdir>'`, users that install the Cardinal Optimizer with executable installer can change directory to `'<instdir>\examples\c\vsprojects'` and open the Visual Studio project `lp_ex1.vcxproj` to build the solution. Users that install the Cardinal Optimizer using ZIP-format archive should make sure that all required environment variables are set correctly, see *Install Guide for Cardinal Optimizer* for details.

In addition, for Windows systems, COPT also supports the MinGW-w64 toolchain. Please refer to the corresponding instructions in the Makefile under the path `'<instdir>\examples\c'` for usage.

#### Linux and MacOS

For users on Linux or MacOS, we provide Makefile to build the example. Please install GCC toolchain for Linux and Clang toolchain for MacOS, together with the `make` utility beforehand. What's more, users should make sure also that all required environment variables are set correctly, see *Install Guide for Cardinal Optimizer* for details. Let's assume that the installation directory of Cardinal Optimizer is `'<instdir>'`, then users need to change directory to `'<instdir>\examples\c'` and execute command `make` in terminal.

## 7.2 C++ Interface

This chapter walks through a simple C++ example to illustrate the use of the COPT C++ interface. In short words, the example creates an environment, builds a model, add variables and constraints, optimizes it, and then outputs the optimal objective value.

The example solves the following linear problem:

$$\begin{aligned} &\text{Maximize:} \\ &\qquad 1.2x + 1.8y + 2.1z \\ &\text{Subject to:} \\ &\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\ &\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\ &\text{Bounds:} \\ &\qquad 0.1 \leq x \leq 0.6 \\ &\qquad 0.2 \leq y \leq 1.5 \\ &\qquad 0.3 \leq z \leq 2.8 \end{aligned} \tag{7.2}$$

Note that this is the same problem that was modeled and optimized in chapter of *C Interface*.

### 7.2.1 Example details

Below is the source code solving the above problem using COPT C++ interface.

Listing 7.2: `lp_ex1.cpp`

```
1  /*
2   * This file is part of the Cardinal Optimizer, all rights reserved.
3   */
4  #include "coptcpp_pch.h"
5
6  using namespace std;
7
```

(continues on next page)

(continued from previous page)

```
/*
 * This example solves the following LP model:
 *
 *  Maximize:
 *    1.2 x + 1.8 y + 2.1 z
 *
 *  Subject to:
 *    R0: 1.5 x + 1.2 y + 1.8 z <= 2.6
 *    R1: 0.8 x + 0.6 y + 0.9 z >= 1.2
 *
 *  Where:
 *    0.1 <= x <= 0.6
 *    0.2 <= y <= 1.5
 *    0.3 <= z <= 2.8
 */
int main(int argc, char* argv[])
{
  try
  {
    Envr env;
    Model model = env.CreateModel("lp_ex1");

    // Add variables
    Var x = model.AddVar(0.1, 0.6, 0.0, COPT_CONTINUOUS, "x");
    Var y = model.AddVar(0.2, 1.5, 0.0, COPT_CONTINUOUS, "y");
    Var z = model.AddVar(0.3, 2.8, 0.0, COPT_CONTINUOUS, "z");

    // Set objective
    model.SetObjective(1.2 * x + 1.8 * y + 2.1 * z, COPT_MAXIMIZE);

    // Add linear constraints using linear expression
    model.AddConstr(1.5 * x + 1.2 * y + 1.8 * z <= 2.6, "R0");

    Expr expr(x, 0.8);
    expr.AddTerm(y, 0.6);
    expr += 0.9 * z;
    model.AddConstr(expr >= 1.2, "R1");

    // Set parameters
    model.SetDblParam(COPT_DBLPARAM_TIMELIMIT, 10);

    // Solve problem
    model.Solve();

    // Output solution
    if (model.GetIntAttr(COPT_INTATTR_HASSOL) != 0)
    {
      cout << "\nFound optimal solution:" << endl;
      VarArray vars = model.GetVars();
      for (int i = 0; i < vars.Size(); i++)
      {
        Var var = vars.GetVar(i);
        cout << "  " << var.GetName() << " = " << var.Get(COPT_DBLINFO_VALUE) << endl;
      }
      cout << "Obj = " << model.GetDblAttr(COPT_DBLATTR_OBJVAL) << endl;
    }
```

(continues on next page)

(continued from previous page)

```
  }
  catch (CoptException e)
  {
    cout << "Error Code = " << e.GetCode() << endl;
    cout << e.what() << endl;
  }
  catch (...)
  {
    cout << "Unknown exception occurs!";
  }
}
```

Let's now walk through the example, line by line, to understand how it achieves the desired result of optimizing the model. Note that the example must include header coptcpp_pch.h.

### Creating environment and model

Essentially, any C++ application using Cardinal Optimizer should start with a COPT environment, where user could add one or more models. Note that each model encapsulates a problem and corresponding data.

Furthermore, to create multiple problems, one can load them one by one in the same model, besides the naive option of creating multiple models in the environment.

```
    Envr env;
    Model model = env.CreateModel("lp_ex1");
```

The above call instantiates a COPT environment and a model with name "lp_ex1".

### Adding variables

The next step in our example is to add variables to the model. Variables are added through AddVar() or AddVars() method on the model object. A variable is always associated with a particular model.

```
    // Add variables
    Var x = model.AddVar(0.1, 0.6, 0.0, COPT_CONTINUOUS, "x");
    Var y = model.AddVar(0.2, 1.5, 0.0, COPT_CONTINUOUS, "y");
    Var z = model.AddVar(0.3, 2.8, 0.0, COPT_CONTINUOUS, "z");
```

The first and second arguments to the AddVar() call are the variable lower and upper bounds, respectively. The third argument is the linear objective coefficient (zero here - we'll set the objective later). The fourth argument is the variable type. Our variables are all continuous in this example. The final argument is the name of the variable.

The AddVar() method has been overloaded to accept several different argument lists. Please refer to *C++ API Reference* for further details.

The objective is built here using overloaded operators. The C++ API overloads the arithmetic operators to allow you to build linear expressions by COPT variables. The second argument indicates that the sense is maximization.

### Adding constraints

The next step in the example is to add the linear constraints. As with variables, constraints are always associated with a specific model. They are created using AddConstr() or AddConstrs() methods on the model object.

```
// Add linear constraints using linear expression
model.AddConstr(1.5 * x + 1.2 * y + 1.8 * z <= 2.6, "R0");

Expr expr(x, 0.8);
expr.AddTerm(y, 0.6);
expr += 0.9 * z;
model.AddConstr(expr >= 1.2, "R1");
```

The first constraint is to use overloaded arithmetic operators to build the linear expression. The comparison operators are also overloaded to make it easier to build constraints.

The second constraint is created by building a linear expression incrementally. That is, an expression can be built by constructor of a variable and its coefficient, by AddTerm() method, and by overloaded operators.

### Setting parameters and attributes

The next step in the example is to set parameters and attributes of the problem before optimization.

```
// Set parameters
model.SetDblParam(COPT_DBLPARAM_TIMELIMIT, 10);
```

The SetDblParam() call here with COPT_DBLPARAM_TIMELIMIT argument sets solver to optimize up to 10 seconds.

### Solving problem

Now that the model has been built, the next step is to optimize it:

```
// Solve problem
model.Solve();
```

This routine performs the optimization and populates several internal model attributes (including the status of the optimization, the solution, etc.).

### Outputting solution

After solving the problem, one can query the values of the attributes for various of purposes.

```
// Output solution
if (model.GetIntAttr(COPT_INTATTR_HASSOL) != 0)
{
  cout << "\nFound optimal solution:" << endl;
  VarArray vars = model.GetVars();
  for (int i = 0; i < vars.Size(); i++)
  {
    Var var = vars.GetVar(i);
    cout << "  " << var.GetName() << " = " << var.Get(COPT_DBLINFO_VALUE) << endl;
  }
  cout << "Obj = " << model.GetDblAttr(COPT_DBLATTR_OBJVAL) << endl;
}
```

Specifically, one can query the COPT_INTATTR_HASLPSOL attribute on the model to know whether we have optimal LP solution; query the COPT_DBLINFO_VALUE attribute of a variable to obtain its solution value; query the COPT_DBLATTR_LPOBJVAL attribute on the model to obtain the objective value for the current solution.

The names and types of all model, variable, and constraint attributes can be found in *Attributes* of C API reference.

#### Error handling

Errors in the COPT C++ interface are handled through the C++ exception mechanism. In the example, all COPT statements are enclosed inside a try block, and any associated errors would be caught by the catch block.

```
catch (CoptException e)
{
  cout << "Error Code = " << e.GetCode() << endl;
  cout << e.what() << endl;
}
catch (...)
{
  cout << "Unknown exception occurs!";
}
```

### 7.2.2 Build and Run

To build and run the example, users may refer to files under `$COPT_HOME/examples/cpp`. Specifically, We provide visual studio project on Windows, as well as makefile project on Linux and Mac platforms.

#### Windows

**1. Visual Studio project**

For Windows platform, Visual Studio project is located at `$COPT_HOME/examples/cpp/vsprojects`. Double-clicking the project file `lp_ex1.vcxproj` will bring Visual Studio. Note that it requires Visual studio 2017 or 2019 installed on Windows 10 to build.

The Visual Studio project has dependency on COPT cpp shared library `copt_cpp.dll`, which is referred in project file, along with its import library `copt_cpp.lib`. The required headers are `copt.h`, `coptcpp.h` and `coptcpp.idl.h`, which declare COPT constants, interfaces and methods exported from `copt_cpp.dll`. In addition, the example provides class header files under `$COPT_HOME/include/coptcpp_inc`, which wraps COPT cpp interfaces and redefines overloaded operators.

In simple terms, users only need to include the header file `coptcpp_pch.h` as shown in the example, configure additional dependencies as `copt_cpp.lib`, and set the directory of additional link libraries as `$(COPT_HOME)/lib`, and make sure that the dynamic library `copt_cpp.dll` has been installed in the appropriate path, and can be loaded at runtime.

To run the example, users should have COPT installed. Specifically, it requires COPT cpp library, `copt_cpp.dll`, and valid license files to run. Please refer to *Install Guide for Cardinal Optimizer* for further details.

In addition, on Windows systems, the Clang and Intel compilers, like the aforementioned MSVC compiler, all support compiling the C++ examples of COPT.

**2. MinGW-w64 Toolchain**

For Windows systems, COPT also supports the MinGW-w64 toolchain. For the two variants of the MinGW-w64, we provide the dynamic library `copt_mmingwcpp.dll` supporting MSVCRT and the dynamic library `copt_umingwcpp.dll` supporting UCRT. Please refer to the corresponding instructions in the Makefile under the path `$COPT_HOME/examples/cpp` for usage.

We recommend using `copt_mmingwcpp.dll` supporting MSVCRT by default, and trying the UCRT if any issues arise.

#### Linux and MacOS

We provide Makefile to build the example for Linux and MacOS platforms. Please make sure tools, `gcc` and `make`, are already installed on the platforms. To build the example, change directory to `$COPT_HOME/examples/cpp` and execute command `'make'` on unix terminal.

The project has dependency on COPT cpp shared library, that is, `libcopt_cpp.so` on Linux platform and `libcopt_cpp.dylib` on MacOS. Similar to Windows VS project, user should refer to `coptcpp_pch.h` under `$COPT_HOME/include/coptcpp_inc'` to include all necessary headers, as shown in the example.

To run the example, users should have COPT installed. Specifically, it requires COPT cpp library, `libcopt_cpp.so` on Linux, `libcopt_cpp.dylib` on MacOS, and valid license files to run. Alternatively, user might set `LD_LIBRARY_PATH` and `COPT_LICENSE_DIR` properly to work around. Please refer to *Install Guide for Cardinal Optimizer* for further details.

## 7.3 C# Interface

This chapter walks through a simple C# example to illustrate the use of the COPT C# interface. In short words, the example creates an environment, builds a model, add variables and constraints, optimizes it, and then outputs the optimal objective value.

The example solves the following linear problem:

$$
\begin{aligned}
&\text{Maximize:} \\
&\qquad 1.2x + 1.8y + 2.1z \\
&\text{Subject to:} \\
&\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\
&\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\
&\text{Bounds:} \\
&\qquad 0.1 \leq x \leq 0.6 \\
&\qquad 0.2 \leq y \leq 1.5 \\
&\qquad 0.3 \leq z \leq 2.8
\end{aligned}
\tag{7.3}
$$

Note that this is the same problem that was modeled and optimized in chapter of *C Interface*.

### 7.3.1 Example details

Below is the source code solving the above problem using COPT C# interface.

Listing 7.3: `lp_ex1.cs`

```
1  /*
2   * This file is part of the Cardinal Optimizer, all rights reserved.
3   */
4  using Copt;
5  using System;
6
```

(continues on next page)

(continued from previous page)

```
/*
 * This C# example solves the following LP model:
 *
 *
 * Maximize:
 *  1.2 x + 1.8 y + 2.1 z
 *
 * Subject to:
 *  1.5 x + 1.2 y + 1.8 z <= 2.6
 *  0.8 x + 0.6 y + 0.9 z >= 1.2

 * where:
 *  0.1 <= x <= 0.6
 *  0.2 <= y <= 1.5
 *  0.3 <= z <= 2.8
 */
public class lp_ex1
{
  public static void Main()
  {
    try
    {
      Envr env = new Envr();
      Model model = env.CreateModel("lp_ex1");

      /*
       * Add variables x, y, z
       *
       * obj: 1.2 x + 1.8 y + 2.1 z
       *
       * var:
       *  0.1 <= x <= 0.6
       *  0.2 <= y <= 1.5
       *  0.3 <= z <= 2.8
       */
      Var x = model.AddVar(0.1, 0.6, 0.0, Copt.Consts.CONTINUOUS, "x");
      Var y = model.AddVar(0.2, 1.5, 0.0, Copt.Consts.CONTINUOUS, "y");
      Var z = model.AddVar(0.3, 2.8, 0.0, Copt.Consts.CONTINUOUS, "z");

      model.SetObjective(1.2 * x + 1.8 * y + 2.1 * z, Copt.Consts.MAXIMIZE);

      /*
       * Add two constraints using linear expression
       *
       * r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
       * r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
       */
      model.AddConstr(1.5 * x + 1.2 * y + 1.8 * z <= 2.6, "r0");

      Expr expr = new Expr(x, 0.8);
      expr.AddTerm(y, 0.6);
      expr += 0.9 * z;
      model.AddConstr(expr >= 1.2, "r1");

      // Set parameters
      model.SetDblParam(Copt.DblParam.TimeLimit, 10);
```

(continues on next page)

(continued from previous page)

```
63
64      // Solve problem
65      model.Solve();
66
67      // Output solution
68      if (model.GetIntAttr(Copt.IntAttr.Status) == Copt.Status.OPTIMAL)
69      {
70        Console.WriteLine("\nFound optimal solution:");
71        VarArray vars = model.GetVars();
72        for (int i = 0; i < vars.Size(); i++)
73        {
74          Var xi = vars.GetVar(i);
75          Console.WriteLine("  {0} = {1}", xi.GetName(), xi.Get(Copt.DblInfo.Value));
76        }
77        Console.WriteLine("Obj = {0}", model.GetDblAttr(Copt.DblAttr.ObjVal));
78      }
79
80      Console.WriteLine("\nDone");
81    }
82    catch (CoptException e)
83    {
84      Console.WriteLine("Error Code = {0}", e.GetCode());
85      Console.WriteLine(e.Message);
86    }
87  }
88 }
```

Let's now walk through the example, line by line, to understand how it achieves the desired result of optimizing the model.

### Creating environment and model

Essentially, any C# application using Cardinal Optimizer should start with a COPT environment, where user could add one or more models. Note that each model encapsulates a problem and corresponding data.

Furthermore, to create multiple problems, one can load them one by one in the same model, besides the naive option of creating multiple models in the environment.

```
      Envr env = new Envr();
      Model model = env.CreateModel("lp_ex1");
```

The above call instantiates a COPT environment and a model with name "lp_ex1".

### Adding variables

The next step in our example is to add variables to the model. Variables are added through AddVar() or AddVars() method on the model object. A variable is always associated with a particular model.

```
      /*
       * Add variables x, y, z
       *
       * obj: 1.2 x + 1.8 y + 2.1 z
       *
       * var:
```

(continues on next page)

(continued from previous page)

```
     *  0.1 <= x <= 0.6
     *  0.2 <= y <= 1.5
     *  0.3 <= z <= 2.8
     */
    Var x = model.AddVar(0.1, 0.6, 0.0, Copt.Consts.CONTINUOUS, "x");
    Var y = model.AddVar(0.2, 1.5, 0.0, Copt.Consts.CONTINUOUS, "y");
    Var z = model.AddVar(0.3, 2.8, 0.0, Copt.Consts.CONTINUOUS, "z");

    model.SetObjective(1.2 * x + 1.8 * y + 2.1 * z, Copt.Consts.MAXIMIZE);
```

The first and second arguments to the AddVar() call are the variable lower and upper bounds, respectively. The third argument is the linear objective coefficient (zero here - we'll set the objective later). The fourth argument is the variable type. Our variables are all continuous in this example. The final argument is the name of the variable.

The AddVar() method has been overloaded to accept several different argument lists. Please refer to *Model* of C# API reference for further details.

The objective is built here using overloaded operators. The C# API overloads the arithmetic operators to allow you to build linear expressions by COPT variables. The second argument indicates that the sense is maximization.

### Adding constraints

The next step in the example is to add the linear constraints. As with variables, constraints are always associated with a specific model. They are created using AddConstr() or AddConstrs() methods on the model object.

```
    /*
     * Add two constraints using linear expression
     *
     * r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
     * r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
     */
    model.AddConstr(1.5 * x + 1.2 * y + 1.8 * z <= 2.6, "r0");

    Expr expr = new Expr(x, 0.8);
    expr.AddTerm(y, 0.6);
    expr += 0.9 * z;
    model.AddConstr(expr >= 1.2, "r1");
```

The first constraint is to use overloaded arithmetic operators to build the linear expression. The comparison operators are also overloaded to make it easier to build constraints.

The second constraint is created by building a linear expression incrementally. That is, an expression can be built by constructor of a variable and its coefficient, by AddTerm(), and by overloaded operators.

### Setting parameters and attributes

The next step in the example is to set parameters and attributes of the problem before optimization.

```
    // Set parameters
    model.SetDblParam(Copt.DblParam.TimeLimit, 10);
```

The SetDblParam() call here with Copt.DblParam.TimeLimit argument sets solver to optimize up to 10 seconds.

### Solving problem

Now that the model has been built, the next step is to optimize it:

```
    // Solve problem
    model.Solve();
```

This routine performs the optimization and populates several internal model attributes (including the status of the optimization, the solution, etc.).

### Outputting solution

After solving the problem, one can query the values of the attributes for various of purposes.

```
    // Output solution
    if (model.GetIntAttr(Copt.IntAttr.Status) == Copt.Status.OPTIMAL)
    {
      Console.WriteLine("\nFound optimal solution:");
      VarArray vars = model.GetVars();
      for (int i = 0; i < vars.Size(); i++)
      {
        Var xi = vars.GetVar(i);
        Console.WriteLine("  {0} = {1}", xi.GetName(), xi.Get(Copt.DblInfo.Value));
      }
      Console.WriteLine("Obj = {0}", model.GetDblAttr(Copt.DblAttr.ObjVal));
    }

    Console.WriteLine("\nDone");
```

Specifically, one can query the Copt.IntAttr.LpStatus attribute of the model to determine whether we have found optimal LP solution; query the Copt.DblInfo.Value attribute of a variable to obtain its solution value; query the Copt.DblAttr.LpObjVal attribute on the model to obtain the objective value for the current solution.

The names and types of all model, variable, and constraint attributes can be found in *Constants* of C# API reference.

#### Error handling

Errors in the COPT C# interface are handled through the C# exception mechanism. In the example, all COPT statements are enclosed inside a try block, and any associated errors would be caught by the catch block.

```
    catch (CoptException e)
    {
      Console.WriteLine("Error Code = {0}", e.GetCode());
      Console.WriteLine(e.Message);
    }
```

#### Releasing environments and models

`Envr` and `Model` implement `IDisposable`. Wrap them in `using` statements so their native resources are released deterministically, on the same thread that used them, with the model disposed before the environment:

```
using (Envr env = new Envr())
using (Model model = env.CreateModel("lp_ex1"))
{
  // build and solve the model
  model.Solve();
}
```

`Dispose()` may also be called explicitly, e.g. in a `finally` block, and is idempotent. Do not rely on the garbage collector to release environments or models: finalization runs at an unpredictable time, which matters for long-running applications that solve many problems, e.g. on a thread pool.

Note that `Envr.Close()` only closes the connection to a remote server and releases the license token; it does not release the environment or the models created from it, which `Dispose()` does. For a floating or cluster license both calls release the token, but `Close()` releases it sooner.

### 7.3.2 Build and Run

To build and run csharp example, users may refer to project under `$COPT_HOME/examples/csharp`. Specifically, We provide a csharp project file in cross-platform framework of dotnet core 2.0. This example shows a single project working on Windows, as well as Linux and Mac platforms.

First of all, download and install dotnet core 2.0 on your platform. To get started, follow instructions in the dotnet core docs.

#### Dotnet core 2.0 project

The dotnet core 2.0 project file `example.csproj` example locates in folder `$COPT_HOME/examples/csharp/dotnetprojects`. Copy example file `lp_ex1.cs` to this folder and change directory to there by Windows command line prompt, then run with command `'dotnet run --framework netcoreapp2.0'`. For users of dotnet core 3.0, just run with `'dotnet run --framework netcoreapp3.0'` instead will work too.

This csharp project has dependency on COPT dotnet 2.0 shared library `copt_dotnet20.dll`, which is referred in the project file and defines all managed classes of COPT solver. In addition, `copt_dotnet20.dll` loads two shared libraries, that is, `coptcswrap.dll` and `copt_cpp.dll` on Windows, `libcoptcswrap.so` and `libcopt_cpp.so` on Linux, `libcoptcswrap.dylib` and `libcopt_cpp.dylib` on Mac respectively. Note that `coptcswrap` library is a bridge between managed COPT library and native library `copt_cpp.dll`, which declares and implements COPT constants, interfaces and methods. So users should make sure they are installed properly on runtime search paths.

In summary, to run csharp example, users should have COPT installed properly. Specifically, it requires three related COPT shared libraries existing on runtime search paths, and valid license files to run. Please refer to *Install Guide for Cardinal Optimizer* for further details.

## 7.4 Java Interface

This chapter walks through a simple Java example to illustrate the use of the COPT Java interface. In short words, the example creates an environment, builds a model, add variables and constraints, optimizes it, and then outputs the optimal objective value.

The example solves the following linear problem:

$$\begin{aligned} &\text{Maximize:} \\ &\qquad 1.2x + 1.8y + 2.1z \\ &\text{Subject to:} \\ &\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\ &\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\ &\text{Bounds:} \\ &\qquad 0.1 \leq x \leq 0.6 \\ &\qquad 0.2 \leq y \leq 1.5 \\ &\qquad 0.3 \leq z \leq 2.8 \end{aligned} \tag{7.4}$$

Note that this is the same problem that was modeled and optimized in chapter of *C Interface*.

### 7.4.1 Example details

Below is the source code solving the above problem using COPT Java interface.

Listing 7.4: `Lp_ex1.java`

```
/*
 * This file is part of the Cardinal Optimizer, all rights reserved.
 */
import copt.*;

/*
 * This Java example solves the following LP model:
 *
 * Maximize:
 *  1.2 x + 1.8 y + 2.1 z
 *
 * Subject to:
 *  1.5 x + 1.2 y + 1.8 z <= 2.6
 *  0.8 x + 0.6 y + 0.9 z >= 1.2
 *
 * where:
 *  0.1 <= x <= 0.6
 *  0.2 <= y <= 1.5
 *  0.3 <= z <= 2.8
 */
public class Lp_ex1 {
  public static void main(final String argv[]) {
    try {
      Envr env = new Envr();
```

(continues on next page)

(continued from previous page)

```
      Model model = env.createModel("lp_ex1");

      /*
       * Add variables x, y, z
       *
       * obj: 1.2 x + 1.8 y + 2.1 z
       *
       * var:
       *  0.1 <= x <= 0.6
       *  0.2 <= y <= 1.5
       *  0.3 <= z <= 2.8
       */
      Var x = model.addVar(0.1, 0.6, 1.2, copt.Consts.CONTINUOUS, "x");
      Var y = model.addVar(0.2, 1.5, 1.8, copt.Consts.CONTINUOUS, "y");
      Var z = model.addVar(0.3, 2.8, 2.1, copt.Consts.CONTINUOUS, "z");

      /*
       * Add two constraints using linear expression
       *
       * r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
       * r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
       */
      Expr e0 = new Expr(x, 1.5);
      e0.addTerm(y, 1.2);
      e0.addTerm(z, 1.8);
      model.addConstr(e0, copt.Consts.LESS_EQUAL, 2.6, "r0");

      Expr e1 = new Expr(x, 0.8);
      e1.addTerm(y, 0.6);
      e1.addTerm(z, 0.9);
      model.addConstr(e1, copt.Consts.GREATER_EQUAL, 1.2, "r1");

      // Set parameters and attributes
      model.setDblParam(copt.DblParam.TimeLimit, 10);
      model.setObjSense(copt.Consts.MAXIMIZE);

      // Solve problem
      model.solve();

      // Output solution
      if (model.getIntAttr(copt.IntAttr.HasSol) != 0) {
        System.out.println("\nFound optimal solution:");
        VarArray vars = model.getVars();
        for (int i = 0; i < vars.size(); i++) {
          Var xi = vars.getVar(i);
          System.out.println("  " + xi.getName() + " = " + xi.get(copt.DblInfo.
↪Value));
        }
        System.out.println("Obj = " + model.getDblAttr(copt.DblAttr.ObjVal));
      }

      System.out.println("\nDone");
    } catch (CoptException e) {
      System.out.println("Error Code = " + e.getCode());
      System.out.println(e.getMessage());
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Let's now walk through the example, line by line, to understand how it achieves the desired result of optimizing the model.

### Import COPT class

To use the Java interface of COPT, users need to import the Java interface class of COPT first.

```
import copt.*;
```

### Creating environment and model

Essentially, any Java application using Cardinal Optimizer should start with a COPT environment, where user could add one or more models. Note that each model encapsulates a problem and corresponding data.

Furthermore, to create multiple problems, one can load them one by one in the same model, besides the naive option of creating multiple models in the environment.

```
    Envr env = new Envr();
    Model model = env.createModel("lp_ex1");
```

The above call instantiates a COPT environment and a model with name "lp_ex1".

### Adding variables

The next step in our example is to add variables to the model. Variables are added through addVar() or addVars() method on the model object. A variable is always associated with a particular model.

```
    /*
     * Add variables x, y, z
     *
     * obj: 1.2 x + 1.8 y + 2.1 z
     *
     * var:
     *  0.1 <= x <= 0.6
     *  0.2 <= y <= 1.5
     *  0.3 <= z <= 2.8
     */
    Var x = model.addVar(0.1, 0.6, 1.2, copt.Consts.CONTINUOUS, "x");
    Var y = model.addVar(0.2, 1.5, 1.8, copt.Consts.CONTINUOUS, "y");
    Var z = model.addVar(0.3, 2.8, 2.1, copt.Consts.CONTINUOUS, "z");
```

The first and second arguments to the addVar() call are the variable lower and upper bounds, respectively. The third argument is the linear objective coefficient. The fourth argument is the variable type. Our variables are all continuous in this example. The final argument is the name of the variable.

The addVar() method has been overloaded to accept several different argument lists. Please refer to *Java Modeling Classes* of Java API reference for further details.

### Adding constraints

The next step in the example is to add the linear constraints. As with variables, constraints are always associated with a specific model. They are created using addConstr() or addConstrs() methods on the model object.

```
/*
 * Add two constraints using linear expression
 *
 * r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
 * r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
 */
Expr e0 = new Expr(x, 1.5);
e0.addTerm(y, 1.2);
e0.addTerm(z, 1.8);
model.addConstr(e0, copt.Consts.LESS_EQUAL, 2.6, "r0");

Expr e1 = new Expr(x, 0.8);
e1.addTerm(y, 0.6);
e1.addTerm(z, 0.9);
model.addConstr(e1, copt.Consts.GREATER_EQUAL, 1.2, "r1");
```

Two constraints here are created by building linear expressions incrementally. That is, an expression can be built by constructor of a variable and its coefficient, and then by addTerm() method.

### Setting parameters and attributes

The next step in the example is to set parameters and attributes of the problem before optimization.

```
// Set parameters and attributes
model.setDblParam(copt.DblParam.TimeLimit, 10);
model.setObjSense(copt.Consts.MAXIMIZE);
```

The setDblParam() call here with copt.DblParam.TimeLimit argument sets solver to optimize up to 10 seconds. The setObjSense() call with copt.Consts.MAXIMIZE argument sets objective sense as maximization.

### Solving problem

Now that the model has been built, the next step is to optimize it:

```
// Solve problem
model.solve();
```

This routine performs the optimization and populates several internal model attributes (including the status of the optimization, the solution, etc.).

### Outputting solution

After solving the problem, one can query the values of the attributes for various of purposes.

```
      // Output solution
      if (model.getIntAttr(copt.IntAttr.HasSol) != 0) {
        System.out.println("\nFound optimal solution:");
        VarArray vars = model.getVars();
        for (int i = 0; i < vars.size(); i++) {
          Var xi = vars.getVar(i);
          System.out.println("  " + xi.getName() + " = " + xi.get(copt.DblInfo.
↪Value));
        }
        System.out.println("Obj = " + model.getDblAttr(copt.DblAttr.ObjVal));
      }

      System.out.println("\nDone");
```

Specifically, one can query the copt.IntAttr.HasLpSol attribute on the model to know whether we have optimal LP solution; query the copt.DblInfo.Value attribute of a variable to obtain its solution value; query the copt.DblAttr.LpObjVal attribute on the model to obtain the objective value for the current solution.

The names and types of all model, variable, and constraint attributes can be found in *Constants* of Java API reference.

### Error Handling

Errors in the COPT Java interface are handled through the Java exception mechanism. In the example, all COPT statements are enclosed inside a try block, and any associated errors would be caught by the catch block.

```
    } catch (CoptException e) {
      System.out.println("Error Code = " + e.getCode());
      System.out.println(e.getMessage());
    }
```

### Releasing environments and models

The native resources behind `Envr` and `Model` objects are not managed by the Java garbage collector. Once they are no longer needed, release them explicitly with `dispose()`, on the same thread that used them - typically in `finally` blocks, disposing the model before the environment:

```
Envr env = new Envr();
try {
  Model model = env.createModel("lp_ex1");
  try {
    // build and solve the model
    model.solve();
  } finally {
    model.dispose();
  }
} finally {
  env.dispose();
}
```

Both `dispose()` methods are idempotent. Do not rely on the garbage collector to release environments or models: finalization runs at an unpredictable time, which matters for long-running applications that solve many problems, e.g. on a thread pool. The native problem data is shared by the model and the `Var`/`Constraint` objects created from it, and is freed once the last of these handles has been released.

Note that `Envr.close()` only closes the connection to a remote server and releases the license token; it does not release the environment or the models created from it, which `dispose()` does. For a floating or cluster license both calls release the token, but `close()` releases it sooner.

### 7.4.2 Build and Run

To build and run java example, users may refer to files under `$COPT_HOME/examples/java`. Specifically, We provide an example file in java and a script file to build. This single example runs on all platforms that support Java.

First of all, download and install *Java 8 or above* on your platform.

#### Java example detail

In the java example folder `$COPT_HOME/examples/java`, the easiest way to run the example is to enter the java example folder in console or terminal and then execute command `'sh run.sh'`.

This java project has dependency on COPT java package `copt_java.jar`, which defines all java classes of COPT solver. In addition, `copt_java.jar` loads two shared libraries, that is, `coptjniwrap.dll` and `copt_cpp.dll` on Windows, `libcoptjniwrap.so` and `libcopt_cpp.so` on Linux, `libcoptjniwrap.dylib` and `libcopt_cpp.dylib` on Mac respectively. Note that `coptjniwrap` library is a JNI swig wrapper and acts as a bridge between COPT java package and native library `copt_cpp`, which declares and implements COPT constants, interfaces and methods. So users should make sure they are installed properly on runtime search paths.

In summary, to run java example, users should have COPT installed properly. Specifically, it requires two related COPT shared libraries existing on runtime search paths, and valid license files to run. Please refer to *Install Guide for Cardinal Optimizer* for further details.

## 7.5 Python Interface

### 7.5.1 Installation guide

Currently, the Python interface of Cardinal Optimizer supports Python 3.7-3.14. Among them, for Python 3.8-3.14 versions, COPT's MacOS-Universal can provide support. For Python 3.7, only MacOS-X86 is available.

Before using the Python interface, please ensure that COPT has been installed and configured correctly. For details, please refer to *How to install the Cardinal Optimizer*. Users can download Python from Anaconda distribution or Python official distribution . We recommend users install the Anaconda distribution, because it is more user-friendly and convenient for Python novices (For Windows, please don't install Python via Microsoft Store). If you use official Python distribution or Python shipped with system, then make sure you have installed the `pip` and `setuptools` Python packages beforehand.

**Note**

1. We recommend using Python 3.8 or later. The matrix modeling functionality of the COPT-Python interface requires Python 3.8 or later.
2. Python 3.12 and later have removed `distutils` from the standard library. Therefore, the `python setup.py install` command used by Method 2 below requires the `distutils` module

provided by `setuptools`. To deploy in an environment where `setuptools` cannot be installed, please refer to *Offline Installation*.

### Windows

#### Method 1: via pip install (recommended)

Open the cmd command window (if Python is an Anaconda distribution, open the Anaconda command line window) and enter the following command:

```
pip install coptpy
```

If an older version of the `coptpy` package has been installed, please open the cmd command window (if Python is an Anaconda distribution, open the Anaconda command line window) and enter the following command to upgrade to the latest version of the `coptpy` package:

```
pip install --upgrade coptpy
```

#### Method 2: via COPT installation package

For Windows, assuming the installation path of COPT is: `"C:\Program Files\COPT"`, please switch to the directory `"C:\Program Files\COPT\lib\python"` and execute the following commands on command line:

```
python setup.py install
```

Note that if COPT is installed on the system disk, you need to **execute with administrator privileges** to open the command prompt. To test whether the Python interface is installed correctly, users can switch to the directory `"C:\Program Files\COPT\examples\python"` and execute the following commands on the command line:

```
python lp_ex1.py
```

If the model is solved correctly, it means that the Python interface of COPT has been installed correctly.

**Note** If you use the official release of Python 3.8, assume that its installation path is: `"C:\Program Files\Python38"`, you need to copy the `copt_cpp.dll` file in the `"bin"` subdirectory of the COPT installation path to `"C:\Program Files\Python38\Lib\site-packages\coptpy"` to solve the problem of dynamic library dependency.

Currently, `coptpy` supports type hints. After `coptpy` is successfully installed, when you code in Python IDE, variable name completion and function prototype information will be automatically prompted.

### Linux

#### Method 1: via pip install (recommended)

Open the terminal and enter the following command:

```
pip install coptpy
```

If an older version of the `coptpy` package has been installed, please open the terminal and enter the following command to upgrade to the latest version of the `coptpy` package:

```
pip install --upgrade coptpy
```

#### Method 2: via COPT installation package

For Linux, suppose the installation path of COPT is: /opt/copt80, please switch to the directory /opt/copt80/lib/python and execute the following commands on terminal:

```
sudo python setup.py install
```

For users using Python from Anaconda distribution, if above commands fails, assuming the installation path of Anaconda is: `"/opt/anaconda3"`, please execute the following commands instead on terminal to install the Python interface of COPT:

```
sudo /opt/anaconda3/bin/python setup.py install
```

To test whether the Python interface is installed correctly, users can switch to the directory /opt/copt80/examples/python and execute the following commands on terminal:

```
python lp_ex1.py
```

If the model solved correctly, it means that the Python interface of COPT has been installed correctly.

Currently, `coptpy` supports type hints. After `coptpy` is successfully installed, when you code in Python IDE, variable name completion and function prototype information will be automatically prompted.

### MacOS

#### Method 1: via pip install (recommended)

Open the terminal and enter the following command:

```
pip install coptpy
```

If an older version of the `coptpy` package has been installed, please open the terminal and enter the following command to upgrade to the latest version of the `coptpy` package:

```
pip install --upgrade coptpy
```

#### Method 2: via COPT installation package

For MacOS, assuming that the installation path of COPT is: /Applications/copt80, please switch the directory to /Applications/copt80/lib/python and execute the following commands on terminal:

```
sudo python setup.py install
```

To test whether the Python interface is installed correctly, users can switch to the directory /Applications/copt80/examples/python and execute the following commands on terminal:

```
python lp_ex1.py
```

If the model solved correctly, it means that the Python interface of COPT has been installed correctly.

Currently, `coptpy` supports type hints. After `coptpy` is successfully installed, when you code in Python IDE, variable name completion and function prototype information will be automatically prompted.

### 7.5.2 Example details

This chapter illustrate the use of C interface of Cardinal Optimizer through a simple Python example. The problem to solve is shown in Eq. 7.5:

$$\begin{aligned}
&\text{Maximize:} \\
&\qquad 1.2x + 1.8y + 2.1z \\
&\text{Subject to:} \\
&\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\
&\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\
&\text{Bounds:} \\
&\qquad 0.1 \leq x \leq 0.6 \\
&\qquad 0.2 \leq y \leq 1.5 \\
&\qquad 0.3 \leq z \leq 2.8
\end{aligned} \tag{7.5}$$

The source code for solving the above problem using Python API of Cardinal Optimizer is shown in Listing 7.5:

Listing 7.5: `lp_ex1.py`

```
#
# This file is part of the Cardinal Optimizer, all rights reserved.
#

"""
The problem to solve:

  Maximize:
    1.2 x + 1.8 y + 2.1 z

  Subject to:
    1.5 x + 1.2 y + 1.8 z <= 2.6
    0.8 x + 0.6 y + 0.9 z >= 1.2

  where:
    0.1 <= x <= 0.6
    0.2 <= y <= 1.5
    0.3 <= z <= 2.8
"""

import coptpy as cp
from coptpy import COPT

# Create COPT environment
env = cp.Envr()

# Create COPT model
model = env.createModel("lp_ex1")

# Add variables: x, y, z
x = model.addVar(lb=0.1, ub=0.6, name="x")
y = model.addVar(lb=0.2, ub=1.5, name="y")
z = model.addVar(lb=0.3, ub=2.8, name="z")

# Add constraints
model.addConstr(1.5*x + 1.2*y + 1.8*z <= 2.6)
model.addConstr(0.8*x + 0.6*y + 0.9*z >= 1.2)
```

(continues on next page)

(continued from previous page)

```
38
39  # Set objective function
40  model.setObjective(1.2*x + 1.8*y + 2.1*z, sense=COPT.MAXIMIZE)
41
42  # Set parameter
43  model.setParam(COPT.Param.TimeLimit, 10.0)
44
45  # Solve the model
46  model.solve()
47
48  # Analyze solution
49  if model.status == COPT.OPTIMAL:
50    print("Objective value: {}".format(model.objval))
51    allvars = model.getVars()
52
53    print("Variable solution:")
54    for var in allvars:
55      print(" x[{0}]: {1}".format(var.index, var.x))
56
57    print("Variable basis status:")
58    for var in allvars:
59      print(" x[{0}]: {1}".format(var.index, var.basis))
60
61    # Write model, solution and modified parameters to file
62    model.write("lp_ex1.mps")
63    model.write("lp_ex1.bas")
64    model.write("lp_ex1.sol")
65    model.write("lp_ex1.par")
```

We will explain how to use the Python API step by step based on code above, please refer to *C API Reference* for detailed usage of Python API.

### Import Python interface

To use the Python interface of COPT, users need to import the Python interface library first.

```
import coptpy as cp
from coptpy import COPT
```

### Create environment

To solve any problem with COPT, users need to create optimization environment before creating any model.

```
# Create COPT environment
env = cp.Envr()
```

### Create model

If the optimization environment was created successfully, users need to create the model to solve, which includes variables and constraints information.

```
# Create COPT model
model = env.createModel("lp_ex1")
```

### Add variables

Users can specify information such as objective costs, lower and upper bounds of variables when creating them. In this example, we just set the lower and upper bounds of variables and their names.

```
# Add variables: x, y, z
x = model.addVar(lb=0.1, ub=0.6, name="x")
y = model.addVar(lb=0.2, ub=1.5, name="y")
z = model.addVar(lb=0.3, ub=2.8, name="z")
```

### Add constraints

After adding variables, we can then add constraints to the model.

```
# Add constraints
model.addConstr(1.5*x + 1.2*y + 1.8*z <= 2.6)
model.addConstr(0.8*x + 0.6*y + 0.9*z >= 1.2)
```

### Set objective function

After adding variables and constraints, we can further specify objective function for the model.

```
# Set objective function
model.setObjective(1.2*x + 1.8*y + 2.1*z, sense=COPT.MAXIMIZE)
```

### Set parameters

Users can set optimization parameters before solving the model, e.g. set optimization time limit to 10 seconds.

```
# Set parameter
model.setParam(COPT.Param.TimeLimit, 10.0)
```

#### Solve model

Solve the model via `solve` method.

```
# Solve the model
model.solve()
```

#### Analyze solution

When solving finished, we should query the solution status first. If the solution status is optimal, then we can retrieve objective value, solution and basis status of variables.

```
# Analyze solution
if model.status == COPT.OPTIMAL:
  print("Objective value: {}".format(model.objval))
  allvars = model.getVars()

  print("Variable solution:")
  for var in allvars:
    print(" x[{0}]: {1}".format(var.index, var.x))

  print("Variable basis status:")
  for var in allvars:
    print(" x[{0}]: {1}".format(var.index, var.basis))
```

#### Write files

Users can write current model to MPS format file, and write solution, basis status and modified parameters to file.

```
  # Write model, solution and modified parameters to file
  model.write("lp_ex1.mps")
  model.write("lp_ex1.bas")
  model.write("lp_ex1.sol")
  model.write("lp_ex1.par")
```

### 7.5.3 Best Practice

#### Release environments and models

`Envr` and `Model` support the context manager protocol. Use `with` blocks so their native resources are released deterministically when the block exits, with the model released before the environment:

```
import coptpy as cp

with cp.Envr() as env:
    with env.createModel("lp_ex1") as model:
        # build and solve the model
        model.solve()
```

Alternatively, call `env.dispose()` and `model.dispose()` explicitly, e.g. in a `finally` block; both are idempotent.

Explicit release is especially important for models that set a callback: the model and the callback reference each other, and objects caught in such a reference cycle are only reclaimed by Python's cyclic garbage

collector - at an unpredictable time and on an arbitrary thread - unless the model is disposed explicitly. For long-running applications that solve many problems, always release environments and models on the thread that used them.

Note that `Envr.close()` only closes the connection to a remote server and releases the license token; it does not release the environment or the models created from it, which `dispose()` does. For a floating or cluster license both calls release the token, but `close()` releases it sooner.

### Upgrade to the newer version

If users have COPT Python Interface `coptpy` installed and need to upgrade latest version, it is recommended to remove previous version before installing new version. To remove previous version, it is as simple as deleting the folder `coptpy` at `site-package`.

If the user is accessible to the Internet, we recommend using `pip` to uninstall or upgrade `coptpy`.

**Uninstall the old package**

```
pip uninstall coptpy
```

**Upgrade to the new package**

```
pip install --upgrade coptpy
```

If the user installed `coptpy` from the offline COPT installation package by `python setup.py install`, please refer to the following steps to uninstall and upgrade:

1.Firstly, please confirm the installation location of `coptpy`, which can be viewed through `pip show coptpy`, and the output is as follows:

```
Name: coptpy
Version: 8.0.1
Summary: The Python interface of the Cardinal Optimizer
Home-page: www.shanshu.ai
Author: Cardinal Operations, LLC
Author-email: coptsales@shanshu.ai
License: Cardinal Operations, LLC
End User License Agreement
Location: /Users/user/anaconda3/lib/python3.11/site-packages
Requires:
Required-by:
```

2.Secondly, please manually delete the `coptpy-*.egg-info` and `coptpy` under the `site-packages` directory of the `Location` above.

3.Finally, please install the new version of `coptpy`: enter the COPT installation package directory (such as copt80/lib/python), and execute:

```
python setup.py install
```

### Offline Installation

Python 3.12 and later have removed `distutils` from the standard library. Therefore, running `python setup.py install` requires `setuptools` to be installed in the environment, which may not be available on machines prepared for offline deployment.

In this case, we recommend installing from a local `wheel` file. This method does not require `setuptools` or a network connection. It requires `pip`, which is typically included with both the Anaconda distribution and the official Python distribution.

Please contact the COPT team to obtain the `coptpy wheel` file corresponding to your operating system, CPU architecture, and Python version. Copy the file to the target machine, switch to the directory containing the file, and run the following command:

```
pip install --no-index --find-links=. coptpy
```

### Multi-Thread Programming

COPT does not guarantee thread safe and modeling APIs are not reentrant in general. It is safe to share COPT Envr objects among threads. However, it is not recommended to share Model objects among threads, unless you understand what you are doing. For instance, if you share the same model between two threads. One thread is responsible for modeling and solving. The other thread is used to monitor the progress and may interrupt at some circumstances, such as running out of time.

### Dictionary order guaranteed after Python 3.7

As you know, Python dictionaries did not preserve the order in which items were added to them. For instance, you might type `{'fruits': ['apple', 'orange'], 'veggies': ['carrot', 'pea']}` and get back `{'veggies': ['carrot', 'pea'], 'fruits': ['apple', 'orange']}`.

However, the situation is changed. Standard dict objects preserve order in the implementation of Python 3.6. This order-preserving property is becoming a language feature in Python 3.7.

If your program has dependency on dictionary orders, please install COPT Python Interface `coptpy` for Python 3.7 or later version. For instance, if your model is implemented in Python 2.7 as follows:

```
m = Envr().createModel("customized model")
vx = m.addVars(['hello', 'world'], [0, 1, 2], nameprefix = "X")
# add a constraint for each var in tupledict 'vx'
m.addConstrs(vx[key] >= 1.0 for key in vx)
```

Your model might end up with rows `{R(hello,1), R(hello,0), R(world,1), R(world,0), R(hello,2), R(world,2)}` .

### Use quicksum and psdquicksum when possible

The Python interface of COPT supports building linear expression, quadratic expression and PSD expression in natural way. For linear and quadratic expression, it is recommended to use quicksum() to build expression objects. For linear and PSD expression, it is recommended to use psdquicksum() to build expression objects. Both of them implement inplace summation, which is much faster than standard plus operator.

#### Operate the model in batch

The Python interface of COPT supports performing batch operations on the models, such as:

- Add multiple variables or constraints: `Model.addVars()/Model.addConstrs()` .
- Set the coefficients of multiple variables in the linear constraints: `Model.setCoeffs()` (**Note:** The index pairs of variables and constraints cannot appear repeatedly).
- Set the names of multiple variables or constraints: `Model.setNames()` .

Please refer to *Python API Reference: Model Class* for function descriptions.

## 7.6 AMPL Interface

AMPL is an algebraic modeling language for describing large-scale complex mathematical problems, it was hooked to many commercial and open-source mathematical optimizers, with various data interfaces and extensions, and received high popularity among both industries and institutes, see Who uses AMPL? for more information. The solver `coptampl` (located in the `"\bin"` directory of the COPT installation package) uses **Cardinal Optimizer** to solve linear programming, convex quadratic programming, convex quadratic constrainted programming and mixed integer programming problems. Normally `coptampl` is invoked by AMPL's solve command, which gives the invocation:

```
coptampl stub -AMPL
```

in which `stub.nl` is an AMPL generic output file (possibly written by `'ampl -obstub'` or `'ampl -ogstub'`). After solving the problem, `coptampl` writes a `stub.sol` file for use by AMPL's solve and solution commands. When you run AMPL, this all happens automatically if you give the AMPL commands:

```
ampl: option solver coptampl;
ampl: solve;
```

### 7.6.1 Installation Guide

To use `coptampl` in AMPL, you must have a valid AMPL license and make sure that you have installed Cardinal Optimizer and setup its license properly, see *How to install Cardinal Optimizer* for details. Be sure to check if it satisfies the following requirements for different operating systems.

#### Windows

On Windows platform, the `coptampl.exe` utility and the `copt.dll` dynamic library contained in the Cardinal Optimizer must appear somewhere in your user or system `PATH` environment variable (or in the current directory).

To test if your setting meets the above requirements, you can check it by executing commands below in command prompt:

```
coptampl -v
```

And you are expected to see output similar to the following on screen:

```
AMPL/x-COPT Optimizer [8.0.7] (windows-x86), driver(20220526), MP(20220526)
```

If the commands failed, then you should recheck your settings.

#### Linux

On Linux platform, the `coptampl` utility must appear somewhere in your `$PATH` environment variable, while the `libcopt.so` shared library must appears somewhere in your `$LD_LIBRARY_PATH` environment variable.

Similarly, to test if your setting meets the above requirements, just execute commands below in shell:

```
coptampl -v
```

And you are expected to see output similar to the following on screen:

```
AMPL/x-COPT Optimizer [8.0.7] (linux-x86), driver(20220526), MP(20220526)
```

If the commands failed, please recheck your settings.

#### MacOS

On MacOS platform, the `coptampl` utility must appear somewhere in your `$PATH` environment variable, while the `libcopt.dylib` dynamic library must appear somewhere in your `$DYLD_LIBRARY_PATH` environment variable.

You can execute commands below in shell to see if your settings meets the above requirements:

```
coptampl -v
```

And you are expected to see output similar to the following on screen:

```
AMPL/x-COPT Optimizer [8.0.7] (macos-x86), driver(20220526), MP(20220526)
```

If the commands failed, then please recheck your settings.

### 7.6.2 Solver Options and Exit Codes

The `coptampl` utility offers some options to customize its behavior. Users can control it by setting the environment variable `copt_options` or use AMPL's `option` command. To see all available options, please invoke:

```
coptampl -=
```

The supported parameters and their interpretation for current version are shown in Table 7.1:

Table 7.1: Parameters of `coptampl`

| Parameter | Interpretation |
|---|---|
| barhomogeneous | whether to use homogeneous self-dual form in barrier |
| bariterlimit | iteration limit of barrier method |
| barthreads | number of threads used by barrier |
| basis | whether to use or return basis status |
| bestbound | whether to return best bound by suffix |
| conflictanalysis | whether to perform conflict analysis |
| crossoverthreads | number of threads used by crossover |
| cutlevel | level of cutting-planes generation |
| divingheurlevel | level of diving heuristics |
| dualize | whether to dualize a problem before solving it |
| dualperturb | whether to allow the objective function perturbation |
| dualprice | specifies the dual simplex pricing algorithm |
| dualtol | the tolerance for dual solutions and reduced cost |
| feastol | the feasibility tolerance |

continues on next page

Table 7.1 – continued from previous page

| | |
|---|---|
| heurlevel | level of heuristics |
| iisfind | whether to compute IIS and return result |
| iismethod | specify the IIS method |
| inttol | the integrality tolerance for variables |
| logging | whether to print solving logs |
| logfile | name of log file |
| exportfile | name of model file to be exported |
| lpmethod | method to solve the LP problem |
| matrixtol | input matrix coefficient tolerance |
| qmatrixtol | input Q matrix coefficient tolerance |
| mipstart | whether to use initial values for MIP problem |
| miptasks | number of MIP tasks in parallel |
| nodecutrounds | rounds of cutting-planes generation of tree node |
| nodelimit | node limit of the optimization |
| objno | objective number to solve |
| count | whether to count the number of solutions |
| stub | name prefix for alternative MIP solutions written |
| presolve | level of presolving before solving a problem |
| relgap | the relative gap of optimization |
| absgap | the absolute gap of optimization |
| return_mipgap | whether to return absolute/relative gap by suffix |
| rootcutlevel | level of cutting-planes generation of root node |
| rootcutrounds | rounds of cutting-planes generation of root node |
| roundingheurlevel | level of rounding heuristics |
| scaling | whether to perform scaling before solving a problem |
| simplexthreads | number of threads used by dual simplex |
| sos | whether to use '.sosno' and '.ref' suffix |
| sos2 | whether to use SOS2 to represent piecewise linear terms |
| strongbranching | level of strong branching |
| submipheurlevel | level of Sub-MIP heuristics |
| threads | number of threads to use |
| timelimit | time limit of the optimization |
| treecutlevel | level of cutting-planes generation of search tree |
| wantsol | whether to generate '.sol' file |

Please refer to *COPT Parameters* for details.

AMPL uses suffix to store or pass model and solution information, and also some extension features, such as support for SOS constraints. Currently, `coptampl` support suffix information as shown in *Suffix supported by coptampl* :

Table 7.2: Suffix supported by `coptampl`

| **Suffix** | **Interpretation** |
|---|---|
| absmipgap | absolute gap for MIP problem |
| bestbound | best bound for MIP problem |
| iis | store IIS status of variables or constraints |
| nsol | number of pool solutions written |
| ref | weight of variable in SOS constraint |
| relmipgap | relative gap for MIP problem |
| sos | store type of SOS constraint |
| sosno | type of SOS constraint |
| sosref | store variable weight in SOS constraint |
| sstatus | basis status of variables and constraints |

Users who want to know how to use SOS constraints in AMPL, please refer to resources in AMPL's website: How to use SOS constraints in AMPL .

When solving finished, `coptampl` will display a status message and return exit code to AMPL. The exit code can be displayed by:

```
ampl: display solve_result_num;
```

If no solution was found or something unexpected happened, `coptampl` will return non-zero code to AMPL from Table 7.3:

Table 7.3: Exit codes of `coptampl`

| Exit Code | Interpretation |
|---|---|
| 0 | optimal solution |
| 200 | infeasible |
| 300 | unbounded |
| 301 | infeasible or unbounded |
| 600 | user interrupted |

### 7.6.3 Example Usage

The following section will illustrate the use of AMPL by a well-known example called "Diet problem", which finds a mix of foods that satisfies requirements on the amounts of various vitamins, see AMPL book for details.

Suppose the following kinds of foods are available for the following prices per unit, see Table 7.4:

Table 7.4: Prices of foods

| Food | Price |
|---|---|
| BEEF | 3.19 |
| CHK | 2.59 |
| FISH | 2.29 |
| HAM | 2.89 |
| MCH | 1.89 |
| MTL | 1.99 |
| SPG | 1.99 |
| TUR | 2.49 |

These foods provide the following percentages, per unit, of the minimum daily requirements for vitamins A, C, B1 and B2, see Table 7.5:

Table 7.5: Nutrition of foods (%)

| | A | C | B1 | B2 |
|---|---|---|---|---|
| BEEF | 60% | 20% | 10% | 15% |
| CHK | 8 | 0 | 20 | 20 |
| FISH | 8 | 10 | 15 | 10 |
| HAM | 40 | 40 | 35 | 10 |
| MCH | 15 | 35 | 15 | 15 |
| MTL | 70 | 30 | 15 | 15 |
| SPG | 25 | 50 | 25 | 15 |
| TUR | 60 | 20 | 15 | 10 |

The problem is to find the cheapest combination that meets a week's requirements, that is, at least 700% of the daily requirements for each nutrient.

To summarize, the mathematical form for the above problem can be modeled as shown in Eq. 7.6:

$$\begin{aligned} &\text{Minimize:} \\ &\quad \sum_{j \in J} cost_j \cdot buy_j \\ &\text{Subject to:} \\ &\quad n_min_i \leq \sum_{j \in J} amt_{i,j} \cdot buy_j \leq n_max_i \ \ \forall i \in I \\ &\quad f_min_j \leq buy_j \leq f_max_j \ \ \forall j \in J \end{aligned} \tag{7.6}$$

The AMPL model for above problem is shown in `diet.mod`, see Listing 7.6:

Listing 7.6: `diet.mod`

```
# The code is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.mod
#
# with some modification by developer of the Cardinal Optimizer

set NUTR;
set FOOD;

param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];

param amt {NUTR, FOOD} >= 0;

var Buy {j in FOOD} >= f_min[j], <= f_max[j];

minimize Total_Cost:
  sum {j in FOOD} cost[j] * Buy[j];

subject to Diet {i in NUTR}:
  n_min[i] <= sum {j in FOOD} amt[i, j] * Buy[j] <= n_max[i];
```

The data file for above problem is shown in `diet.dat`, see Listing 7.7:

Listing 7.7: `diet.dat`

```
# The data is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.dat
#
# with some modification by developer of the Cardinal Optimizer

data;

set NUTR := A B1 B2 C ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:   cost  f_min  f_max :=
  BEEF   3.19    0     100
  CHK    2.59    0     100
```

(continues on next page)

(continued from previous page)

```
  FISH   2.29    0     100
  HAM    2.89    0     100
  MCH    1.89    0     100
  MTL    1.99    0     100
  SPG    1.99    0     100
  TUR    2.49    0     100 ;

param:   n_min  n_max :=
   A      700   10000
   C      700   10000
   B1     700   10000
   B2     700   10000 ;

param amt (tr):
           A    C   B1   B2 :=
   BEEF   60   20   10   15
   CHK     8    0   20   20
   FISH    8   10   15   10
   HAM    40   40   35   10
   MCH    15   35   15   15
   MTL    70   30   15   15
   SPG    25   50   25   15
   TUR    60   20   15   10 ;
```

To solve the problem with `coptampl` in AMPL, just type commands in command prompt on Windows or shell on Linux and MacOS:

```
ampl: model diet.mod;
ampl: data diet.dat;
ampl: option solver coptampl;
ampl: option copt_options 'logging 1';
ampl: solve;
```

`coptampl` solve it quickly and display solving log and status message on screen:

```
x-COPT 5.0.1: optimal solution; objective 88.2
1 simplex iterations
```

So `coptampl` claimed it found the optimal solution, and the minimal cost is 88.2 units. You can further check the solution by:

```
ampl: display Buy;
```

And you will get:

```
Buy [*] :=
BEEF   0
 CHK   0
FISH   0
 HAM   0
 MCH  46.6667
 MTL   0
 SPG   0
 TUR   0
;
```

So if we buy 46.667 units of MCH, we will have a minimal cost of 88.2 units.

## 7.7 Pyomo Interface

Pyomo is a Python based, open source optimization modeling language with a diverse set of optimization capabilities. It is used by researchers to solve complex real-world applications, see Who uses Pyomo? for more introduction. The following documentation explains how to use the **Cardinal Optimizer**.

### 7.7.1 Installation Guide

To use the Cardinal Optimizer in Pyomo, you should setup Pyomo and Cardinal Optimizer correctly first. Pyomo currently supports Python 2.7, 3.6-3.9, you can install Python from Anaconda Distribution of Python or from Official Python. We recommend install Python from Anaconda since it is much more friendly and convenient for fresh users.

#### Using conda

The recommended way to install Pyomo in Anaconda Distribution of Python is to use `conda` which is built-in supported. Just execute the following commands in command prompt on Windows or shell on Linux and MacOS:

```
conda install -c conda-forge pyomo
```

Pyomo also has conditional dependencies on a variety of third-party Python packages, they can be installed using `conda` with commands:

```
conda install -c conda-forge pyomo.extras
```

#### Using pip

The alternative way to install Pyomo is to use the standard `pip` utility, just execute the following commands in command prompt on Windows or shell on Linux and MacOS:

```
pip install pyomo
```

If you encounter any problems when installing Pyomo, please refer to How to install Pyomo for details. To install Cardinal Optimizer and setup its license properly, please refer to *How to install Cardinal Optimizer* for details.

### 7.7.2 Example Usage

We are going to make a simple introduction on how to use the Cardinal Optimizer in Pyomo by solving the example described in *AMPL Interface - Example Usage*. Users who want to learn more information about Pyomo may refer to Pyomo documentation for details.

#### Abstract Model

Pyomo provides two major approaches to construct any supported model types, here we show the **Abstract Model** approach to solve the above problem.

The source code `pydiet_abstract.py` is shown below, see Listing 7.8:

Listing 7.8: pydiet_abstract.py

```
# The code is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.py
#
# with some modification by developer of the Cardinal Optimizer

from pyomo.core import *

model = AbstractModel()

model.NUTR = Set()
model.FOOD = Set()

model.cost  = Param(model.FOOD, within=NonNegativeReals)
model.f_min = Param(model.FOOD, within=NonNegativeReals)

model.f_max = Param(model.FOOD)
model.n_min = Param(model.NUTR, within=NonNegativeReals)
model.n_max = Param(model.NUTR)
model.amt   = Param(model.NUTR, model.FOOD, within=NonNegativeReals)

def Buy_bounds(model, i):
  return (model.f_min[i], model.f_max[i])
model.Buy = Var(model.FOOD, bounds=Buy_bounds)

def Objective_rule(model):
  return sum_product(model.cost, model.Buy)
model.totalcost = Objective(rule=Objective_rule, sense=minimize)

def Diet_rule(model, i):
  expr = 0

  for j in model.FOOD:
    expr = expr + model.amt[i, j] * model.Buy[j]

  return (model.n_min[i], expr, model.n_max[i])
model.Diet = Constraint(model.NUTR, rule=Diet_rule)
```

And the data file pydiet_abstract.dat in Listing 7.9:

Listing 7.9: pydiet_abstract.dat

```
# The data is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.dat
#
# with some modification by developer of the Cardinal Optimizer

data;

set NUTR := A B1 B2 C ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:   cost  f_min  f_max :=
  BEEF   3.19    0     100
  CHK    2.59    0     100
```

(continues on next page)

(continued from previous page)

```
  FISH   2.29    0     100
  HAM    2.89    0     100
  MCH    1.89    0     100
  MTL    1.99    0     100
  SPG    1.99    0     100
  TUR    2.49    0     100 ;

param:   n_min  n_max :=
   A      700   10000
   C      700   10000
   B1     700   10000
   B2     700   10000 ;

param amt (tr):
           A    C   B1   B2 :=
   BEEF   60   20   10   15
   CHK     8    0   20   20
   FISH    8   10   15   10
   HAM    40   40   35   10
   MCH    15   35   15   15
   MTL    70   30   15   15
   SPG    25   50   25   15
   TUR    60   20   15   10 ;
```

To solve the problem using Pyomo and the Cardinal Optimizer, just type commands below in command prompt on Windows or Bash shell on Linux and MacOS.

```
pyomo solve --solver=coptampl pydiet_abstract.py pydiet_abstract.dat
```

When solving the problem, Pyomo write log information to the screen:

```
[    0.00] Setting up Pyomo environment
[    0.00] Applying Pyomo preprocessing actions
[    0.00] Creating model
[    0.01] Applying solver
[    0.05] Processing results
    Number of solutions: 1
    Solution Information
      Gap: None
      Status: optimal
      Function Value: 88.19999999999999
    Solver results file: results.yml
[    0.05] Applying Pyomo postprocessing actions
[    0.05] Pyomo Finished
```

Upon completion, you can check the solution summary in `results.yml`:

```
# ==========================================================
# = Solver Results                                         =
# ==========================================================
# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Lower bound: -inf
  Upper bound: inf
  Number of objectives: 1
```

(continues on next page)

(continued from previous page)

```
  Number of constraints: 4
  Number of variables: 8
  Sense: unknown
# ------------------------------------------------------------
#   Solver Information
# ------------------------------------------------------------
Solver:
- Status: ok
  Message: COPT-AMPL\x3a optimal solution; objective 88.2, iterations 1
  Termination condition: optimal
  Id: 0
  Error rc: 0
  Time: 0.03171110153198242
# ------------------------------------------------------------
#   Solution Information
# ------------------------------------------------------------
Solution:
- number of solutions: 1
  number of solutions displayed: 1
- Gap: None
  Status: optimal
  Message: COPT-AMPL\x3a optimal solution; objective 88.2, iterations 1
  Objective:
    totalcost:
      Value: 88.19999999999999
  Variable:
    Buy[MCH]:
      Value: 46.666666666666664
  Constraint: No values
```

So the minimal total cost is about 88.2 units when buying 46.67 units of MCH.

### Concrete Model

The other approach to construct model in Pyomo is to use **Concrete Model**, we will show how to model and solve the above problem in this way.

Concrete models can be solved using the `"Direct"` and `"Persistent"` interface methods. This method relies on the Pyomo plugin file `"copt_pyomo.py"` of COPT, which is located in the `"lib/pyomo"` subfolder of the installation package.

To use this plugin, you need to copy the `"copt_pyomo.py"` file to the same directory of your program, and have correctly installed the corresponding version of the Python interface of COPT(`coptpy`).

The source code `pydiet_concrete.py` is shown in Listing 7.10:

Listing 7.10: `pydiet_concrete.py`

```
# The code is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.py
#
# with some modification by developer of the Cardinal Optimizer

from __future__ import print_function, division

import pyomo.environ as pyo
```

(continues on next page)

(continued from previous page)

```
import pyomo.opt as pyopt

from copt_pyomo import *

# Nutrition set
NUTR = ["A", "C", "B1", "B2"]
# Food set
FOOD = ["BEEF", "CHK", "FISH", "HAM", "MCH", "MTL", "SPG", "TUR"]

# Price of foods
cost = {"BEEF": 3.19, "CHK": 2.59, "FISH": 2.29, "HAM": 2.89, "MCH": 1.89,
        "MTL":  1.99, "SPG": 1.99, "TUR":  2.49}
# Nutrition of foods
amt = {"BEEF": {"A": 60, "C": 20, "B1": 10, "B2": 15},
       "CHK":  {"A": 8,  "C": 0,  "B1": 20, "B2": 20},
       "FISH": {"A": 8,  "C": 10, "B1": 15, "B2": 10},
       "HAM":  {"A": 40, "C": 40, "B1": 35, "B2": 10},
       "MCH":  {"A": 15, "C": 35, "B1": 15, "B2": 15},
       "MTL":  {"A": 70, "C": 30, "B1": 15, "B2": 15},
       "SPG":  {"A": 25, "C": 50, "B1": 25, "B2": 15},
       "TUR":  {"A": 60, "C": 20, "B1": 15, "B2": 10}}

# The "diet problem" using ConcreteModel
model = pyo.ConcreteModel()

model.NUTR = pyo.Set(initialize=NUTR)
model.FOOD = pyo.Set(initialize=FOOD)

model.cost = pyo.Param(model.FOOD, initialize=cost)

def amt_rule(model, i, j):
  return amt[i][j]
model.amt  = pyo.Param(model.FOOD, model.NUTR, initialize=amt_rule)

model.f_min = pyo.Param(model.FOOD, default=0)
model.f_max = pyo.Param(model.FOOD, default=100)

model.n_min = pyo.Param(model.NUTR, default=700)
model.n_max = pyo.Param(model.NUTR, default=10000)

def Buy_bounds(model, i):
  return (model.f_min[i], model.f_max[i])
model.buy = pyo.Var(model.FOOD, bounds=Buy_bounds)

def Objective_rule(model):
  return pyo.sum_product(model.cost, model.buy)
model.totalcost = pyo.Objective(rule=Objective_rule, sense=pyo.minimize)

def Diet_rule(model, j):
  expr = 0

  for i in model.FOOD:
    expr = expr + model.amt[i, j] * model.buy[i]

  return (model.n_min[j], expr, model.n_max[j])
model.Diet = pyo.Constraint(model.NUTR, rule=Diet_rule)
```

(continues on next page)

(continued from previous page)

```

# Reduced costs of variables
model.rc = pyo.Suffix(direction=pyo.Suffix.IMPORT)

# Activities and duals of constraints
model.slack = pyo.Suffix(direction=pyo.Suffix.IMPORT)
model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

# Use 'copt_direct' solver to solve the problem
solver = pyopt.SolverFactory('copt_direct')

# Use 'copt_persistent' solver to solve the problem
# solver = pyopt.SolverFactory('copt_persistent')
# solver.set_instance(model)

results = solver.solve(model, tee=True)

# Check result
print("")
if results.solver.status == pyopt.SolverStatus.ok and \
   results.solver.termination_condition == pyopt.TerminationCondition.optimal:
  print("Optimal solution found")
else:
  print("Something unexpected happened: ", str(results.solver))

print("")
print("Optimal objective value:")
print("  totalcost: {0:6f}".format(pyo.value(model.totalcost)))

print("")
print("Variables solution:")
for i in FOOD:
  print("  buy[{0:4s}] = {1:9.6f} (rc: {2:9.6f})".format(i, \
                                                  pyo.value(model.buy[i]), \
                                                  model.rc[model.buy[i]]))

print("")
print("Constraint solution:")
for i in NUTR:
  print("  diet[{0:2s}] = {1:12.6f} (dual: {2:9.6f})".format(i, \
                                                      model.slack[model.Diet[i]], \
                                                      model.dual[model.Diet[i]]))
```

To solve the problem using Pyomo and the Cardinal Optimizer, just execute commands below:

```
python pydiet_concrete.py
```

Up completion, you should see solution summary on screen as below:

```
Optimal solution found
Objective:
  totalcost: 88.200000
Variables:
  buy[BEEF] = 0.000000
  buy[CHK ] = 0.000000
  buy[FISH] = 0.000000
```

(continues on next page)

(continued from previous page)

```
  buy[HAM ] = 0.000000
  buy[MCH ] = 46.666667
  buy[MTL ] = 0.000000
  buy[SPG ] = 0.000000
  buy[TUR ] = 0.000000
```

So the Cardinal Optimizer found the optimal solution, which is about 88.2 units when buying about 46.67 units of MCH.

## 7.8 PuLP Interface

PuLP is an open source modeling tool based on Python, it is mainly used for modeling integer programming problems. This chapter introduces how to use the Cardinal Optimizer (COPT) in PuLP.

### 7.8.1 Installation guide

Before calling COPT in PuLP to solve problem, users need to setup PuLP and COPT correctly. PuLP currently supports Python 2.7 and later versions of Python. Users can download Python from Anaconda distribution or Python official distribution . We recommend users install the Anaconda distribution, because it is more user-friendly and convenient for Python novices.

#### Install via conda

We recommend that users who have installed the Anaconda distribution of Python use its own `conda` tool to install PuLP. Execute the following commands in Windows command prompt or terminal on Linux and MacOS:

```
conda install -c conda-forge pulp
```

#### Install via pip

Users can also install PuLP through the standard `pip` tool, execute the following command in Windows command prompt or Linux and MacOS terminal:

```
pip install pulp
```

#### Setup PuLP interface

For PuLP V2.8.0 and above, COPT can be applied directly. After installing and configuring the COPT, users can proceed with the following steps:

```
from pulp import *
```

In the solving function `solve`, specify the solver to COPT to solve:

```
solver = COPT()
result = prob.solve(solver)
```

**Notes**

1. Calling COPT in PuLP via `solver = COPT()` relies on the Python interface of COPT so `coptpy` needs to be installed first.
2. By specifying: `solver = COPT_DLL()` , `solver = COPT_CMD()` , you can also call COPT, depending on COPT installation package.

### 7.8.2 Introduction of features

The PuLP interface of COPT provides two methods: command line and dynamic library, which are introduced as follows:

#### Command line

The command-line method actually calls the interactive tool `copt_cmd` of COPT to solve problems. In this way, PuLP generates the MPS format file corresponding to the model, and combines the parameter settings passed by the user to generate the solving commands. Upon finish of solving, COPT writes and reads the result file, and assigns values to the corresponding variables and return them to PuLP.

Functions of the command line method are encapsulated as class `COPT_CMD`. Users can set parameters when creating the object of the class and provides the following parameters:

- `keepFiles`

  This option controls whether to keep the generated temporary files. The default value is `0`, which means no temporary files are kept.

- `mip`

  This option controls whether to support solving integer programming models. The default value is `True`, which means support solving integer programming models.

- `msg`

  This option controls whether to print log information on the screen. The default value is `True`, that is, print log information.

- `mip_start`

  This option controls whether to use initial solution information for integer programming models. The default value is `False`, that is, the initial solution information will not be used.

- `logfile`

  This option specifies the solver log. The default value is `None`, which means no solver log will be generated.

- `params`

  This option sets optimization parameters in the form of `key=value`. Please refer to the chapter *Parameters* for currently supported parameters.

### Dynamic library

The dynamic library method directly calls COPT C APIs to solve problems. In this way, PuLP generates problem data and call COPT APIs to load the problem and parameters set by the user. When optimization finishes, the solution is obtained by calling COPT APIs, and then assigned to the corresponding variables and constraints, and passed back to PuLP.

Functions of the dynamic library method are encapsulated as class `COPT_DLL`. Users can set parameters when creating the object of the class and provides the following parameters:

- `mip`

  This option controls whether to support solving integer programming models. The default value is `True`, which means support solving integer programming models.

- `msg`

  This option controls whether to print log information on the screen. The default value is `True`, that is, print log information.

- `mip_start`

  This option controls whether to use initial solution information for integer programming models. The default value is `False`, that is, the initial solution information will not used.

- `logfile`

  This option specifies the solver log. The default value is `None`, which means no solver log will be generated.

- `params`

  This option sets optimization parameters in the form of `key=value`. Please refer to the chapter *Parameters* for currently supported parameters.

In addition, the following methods are provided:

- `setParam(self, name, val)`

  Set optimization solution parameters.

- `getParam(self, name)`

  Obtain optimized solution parameters.

- `getAttr(self, name)`

  Get the attribute information of the model.

- `write(self, filename)`

  Output MPS/LP format model file, COPT binary format model file, result file, basic solution file, initial solution file and parameter setting file.

## 7.9 CVXPY Interface

CVXPY is an open source Python-embedded modeling language for convex optimization problems. It lets you express your problem in a natural way that follows the math, which is quite flexible and efficient. This chapter introduces how to use the Cardinal Optimizer (COPT) in CVXPY.

### 7.9.1 Installation guide

Before calling COPT in CVXPY to solve problem, users need to setup CVXPY and COPT correctly. CVXPY currently supports Python 3.7 and later versions of Python. Users can download Python from Anaconda distribution or Python official distribution . We recommend users install the Anaconda distribution, because it is more user-friendly and convenient for Python novices.

#### Install via conda

We recommend that users who have installed the Anaconda distribution of Python use its own `conda` tool to install CVXPY. Execute the following commands in Windows command prompt or terminal on Linux and MacOS:

```
conda install -c conda-forge cvxpy
```

#### Install via pip

Users can also install CVXPY through the standard `pip` tool, execute the following command in Windows command prompt or Linux and MacOS terminal:

```
pip install cvxpy
```

#### Setup CVXPY interface

**CVXPY V1.3** and its above versions support calling COPT directly. users need to install and configure COPT in advance and then:

```
import cvxpy as cp
```

In CVXPY's solving function `solve`, specify the parameter `solver="COPT"` to use the COPT solver to solve:

```
prob.solve(solver="COPT")
```

### 7.9.2 Introduction of features

The CVXPY interface of COPT supports Linear Programming (LP), Mixed Integer Programming (MIP), Convex Quadratic Programming (QP), Second-Order-Cone Programming (SOCP), Semi-definite Programming (SDP), Mixed Integer Convex Quadratic Programming (MIQP) and Mixed Integer Second-Order-Cone Programming (MISOCP), common used parameters are:

- `verbose`

  CVXPY builtin parameter, which controls whether to display solving log to the screen. The default value is `False`, which means no log to be displayed.

- `params`

  This option sets optimization parameters in the form of `key=value`. Please refer to the chapter *Parameters* for currently supported parameters.

# Chapter 8

# General Constants

There are three types of constants.

1. Constructing models, such as optimization directions, constraint senses or variable types.
2. Accessing solution results, such as API return code, basis status and LP status.
3. Monitoring optimization progress, such as callback context.

## 8.1 Version information

- `VERSION_MAJOR`

  The major version number.

- `VERSION_MINOR`

  The minor version number.

- `VERSION_TECHNICAL`

  The technical version number.

## 8.2 Optimization directions

For different optimization scenarios, it may be required to either maximize or minimize the objective function. There are two optimization directions:

- `MINIMIZE`

  For minimizing the objective function.

- `MAXIMIZE`

  For maximizing the objective function.

The optimization direction is automatically set when reading in a problem from file. Besides, COPT provides relevant functions, allowing user to explicitly set. Functions for different APIs are listed below:

Table 8.1: Functions for setting optimization directions

| Programming API | Function |
|---|---|
| C | `COPT_SetObjSense` |
| C++ | `Model::SetObjSense()` |
| C# | `Model.SetObjSense()` |
| Java | `Model.setObjSense()` |
| Python | `Model.setObjSense()` |

**NOTE:** The function names, calling methods and parameter names in different programming interfaces are slightly different. For details, please refer to the API parameters of each programming language.

## 8.3 Infinity and undefined value

**Infinity**

In COPT, the infinite bound is represented by a large value, whose default value is also available as a constant:

- `INFINITY`

  The default value (`1e30`) of the infinite bound.

**Undefined Value**

In COPT, the undefined value is represented by another large value. For example, the default solution value of MIP start is set to a constant:

- `UNDEFINED`

  Undefined value(`1e40`).

## 8.4 Constraint senses

Traditionally, for optimization models, constraints are defined using **senses**. The most common constraint senses are:

- `LESS_EQUAL`

  For constraint in the form of $g(x) \leq b$

- `GREATER_EQUAL`

  For constraint in the form of $g(x) \geq b$

- `EQUAL`

  For constraint in the form of $g(x) = b$

In additional, there are two less used constraint senses:

- `FREE`

  For unconstrained expression

- `RANGE`

  For constraints with both lower and upper bounds in the form of $l \leq g(x) \leq u$.

**NOTE:** Using constraint senses is supported by COPT but not recommended. We recommend defining constraints using explicit lower and upper bounds.

## 8.5 Variable types

Variable types are used for defining whether a variable is continuous or integral.

- `CONTINUOUS`

  Non-integer continuous variables

- `BINARY`

  Binary variables

- `INTEGER`

  Integer variables

## 8.6 SOS constraint types

SOS constraint (Special Ordered Set) is a kind of special constraint that places restrictions on the values that a set of variables can take. COPT currently support two types of SOS constraints:

- `SOS_TYPE1`

  SOS1 constraint

  At most one variable in the constraint is allowed to take a non-zero value.

- `SOS_TYPE2`

  SOS2 constraint

  At most two variables in the constraint are allowed to take non-zero value, and those non-zero variables must be contiguous.

**NOTE:** Variables in SOS constraints are allowed to be continuous, binary and integer.

## 8.7 Indicator constraint types

Indicator constraint is a kind of logical constraints in COPT, used to describe the relationship between the value of the binary variable $y$ and whether the linear constraint $a^T x \leq b$ is satisfied. Currently, COPT supports three types of indicator constraints:

- `INDICATOR_IF`

  If-Then:

  If $y = f$ , then the linear constraint $a^T x \leq b$ is satisfied.

  If $y \neq f$ , then the linear constraint $a^T x \leq b$ is invalid (may be violated).

$$\begin{aligned} y = f \rightarrow a^T x \leq b \\ f \in \{0, 1\} \end{aligned} \tag{8.1}$$

- `INDICATOR_ONLYIF`

  Only-If:

  If the linear constraint $a^T x \leq b$ is satisfied, then $y = f$ .

  If the linear constraint $a^T x \leq b$ is not satisfied, then $y$ can be 0 or 1.

$$\begin{aligned} a^T x \leq b \rightarrow y = f \\ f \in \{0, 1\} \end{aligned} \tag{8.2}$$

- `INDICATOR_IFANDONLYIF`

  If-and-Only-If:

  The linear constraint $a^T x \leq b$ and $y = f$ must be satisfied simultaneously or not satisfied simultaneously.

$$\begin{aligned} a^T x \leq b \leftrightarrow y = f \\ f \in \{0, 1\} \end{aligned} \tag{8.3}$$

## 8.8 SOC constraint types

The Second-Order-Cone (SOC) constraint is a special type of quadratic constraints. COPT supports two types of SOC constraints:

- `CONE_QUAD` : Regular Second-Order-Cone

$$Q^n = \left\{ x \in \mathbb{R}^n \;\middle|\; x_0 \geq \sqrt{\sum_{i=1}^{n-1} x_i^2}, x_0 \geq 0 \right\} \tag{8.4}$$

- `CONE_RQUAD` : Rotated Second-Order-Cone

$$Q_r^n = \left\{ x \in \mathbb{R}^n \;\middle|\; 2x_0 x_1 \geq \sum_{i=2}^{n-1} x_i^2, x_0 \geq 0, x_1 \geq 0 \right\} \tag{8.5}$$

## 8.9 Exponential Cone type

COPT supports two types of exponential cone constraints:

- `EXPCONE_PRIMAL` : Primal exponential cone

$$\mathrm{cl}(S_1) = S_1 \cup S_2 \tag{8.6}$$

$$\begin{aligned} S_1 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid s > 0,\ t \geq s \exp\left(\frac{r}{s}\right) \right\}, \\ S_2 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid s = 0,\ t \geq 0,\ r \leq 0 \right\} \end{aligned} \tag{8.7}$$

- `EXPCONE_DUAL` : Dual exponential cone

$$\mathrm{cl}(S_1) = S_1 \cup S_2 \tag{8.8}$$

$$\begin{aligned} S_1 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid r < 0,\ t \geq -r \exp\left(\frac{s}{r} - 1\right) \right\}, \\ S_2 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid r = 0,\ t \geq 0,\ s \geq 0 \right\} \end{aligned} \tag{8.9}$$

## 8.10 Quadratic objective function

Besides linear objective function, COPT also supports general convex/nonconvex quadratic objective function.

The mathematical form is:

$$x^T Q x + c^T x \tag{8.10}$$

Where, $x$ is an array of variables, $Q$ is the quadratic part of the quadratic objective function and $c$ is the linear part.

## 8.11 Quadratic constraint

Besides the special type of quadratic constraint, Second-Order-Cone (SOC) constraint, COPT also supports general convex/nonconvex quadratic constraint.

The mathematical form is:

$$x^T Q x + q^T x \leq b \tag{8.11}$$

Where, $x$ is an array of variables, $Q$ is the quadratic part of the quadratic constraint and $q$ is the linear part.

## 8.12 Basis status

For an LP problem with $n$ variables and $m$ constraints, the constraints are treated as slack variables *internally*, resulting in $n + m$ variables. When solving an LP problem using the simplex method, the simplex method fixes $n$ variables at one of their bounds, and then computes solutions for the other $m$ variables. The $m$ variables with computed solution are called *basic* variables, and the other $n$ variables are called *non-basic* variables. The simplex progress and its final solution can be defined using the basis status of all the variables and constraints. The basis status supported by COPT are listed below:

Table 8.2: Basis status values and descriptions

| Basis status codes | Value | Description |
|---|---|---|
| `BASIS_LOWER` | 0 | The variable is non-basic at its lower bound |
| `BASIS_BASIC` | 1 | The variable is basic |
| `BASIS_UPPER` | 2 | The variable is non-basic at its upper bound |
| `BASIS_SUPERBASIC` | 3 | The variable is non-basic but not any of its bounds |
| `BASIS_FIXED` | 4 | The variable is non-basic and fixed at its bound |

## 8.13 Solution status

Possible solution status values are listed below:

Table 8.3: Solution Status

| Status Codes | Value | Description |
|---|---|---|
| UNSTARTED | 0 | The solving process has not been started yet |
| OPTIMAL | 1 | The optimal solutions are found |
| INFEASIBLE | 2 | The model is infeasible |
| UNBOUNDED | 3 | The objective is unbounded |
| INF_OR_UNB | 4 | The model is infeasible or unbounded |
| NUMERICAL | 5 | The solver encountered numerical difficulties |
| NODELIMIT | 6 | The solving process was not finished within node limit |
| IMPRECISE | 7 | The solution is imprecise |
| TIMEOUT | 8 | The solving process was not finished within time limit |
| UNFINISHED | 9 | The solving process was not finished due to an internal error |
| INTERRUPTED | 10 | The solving process was stopped by user interruption |
| ITERLIMIT | 11 | The solving process was not finished within iteration limit |
| MEMLIMIT | 12 | The solving process was not finished within memory limit |
| LOCAL_OPTIMAL | 20 | Locally optimal solution satisfying KKT conditions for non-convex model |
| LOCAL_INFEASIBLE | 21 | Locally infeasible for non-convex model |

**Notes**

- In the Python API, solution status are defined in COPT's General Constants Class. They can be accessed via the `"COPT."` prefix or `Model.status`.
- In the Constant class of the Java API and C# API, the constants about the solution status are defined in the `Status` class.
- The optimization status can be obtained through the attribute `"Status"`.
- The LP-relaxation status of the current node can be obtained through the Callback information `"NodeStatus"` . The return value is as above, except for `NODELIMIT`, `UNSTARTED`, `INF_OR_UNB`.

## 8.14 Callback context

- `CBCONTEXT_INCUMBENT`

  Invokes the callback after a new incumbent was found.

- `CBCONTEXT_MIPRELAX`

  Invokes the callback when a new LP-relaxation solution is found.

- `CBCONTEXT_MIPSOL`

  Invokes the callback when a new MIP candidate solution is found.

- `CBCONTEXT_MIPNODE`

  Invokes the callback when a MIP node is finished and LP-relaxation has been solved.

## 8.15 Nonlinear Model Callback Evaluation Types

- `EVALTYPE_OBJVAL`

  Indicates that the callback object can provide the objective function value.

- `EVALTYPE_CONSTRVAL`

  Indicates that the callback object can provide constraint function values.

- `EVALTYPE_GRADIENT`

  Indicates that the callback object can provide first-order derivative information (gradient) of the objective function.

- `EVALTYPE_JACOBIAN`

  Indicates that the callback object can provide first-order derivative information (Jacobian matrix) of the constraint functions.

- `EVALTYPE_HESSIAN`

  Indicates that the callback object can provide second-order derivative information (Hessian matrix) of the nonlinear model Lagrangian function.

## 8.16 Dense Storage Types for Nonlinear Matrices

- `DENSETYPE_ROWMAJOR`

  Indicates that the corresponding gradient, Jacobian, or Hessian matrix is provided in dense format and stored in row-major order.

- `DENSETYPE_COLMAJOR`

  Indicates that the corresponding gradient, Jacobian, or Hessian matrix is provided in dense format and stored in column-major order.

**Notes**

When these two flag is used, the related index array parameters in `Model.loadNlData` may be set to `None`.

## 8.17 Client configuration

For floating, cluster and Web License, the user can set client configuration parameters by calling API functions. The available parameters are:

- `CLIENT_CAFILE`

  Path to the CA public certificate file for configuring SSL proxy.

- `CLIENT_CERTFILE`

  Path to the client certificate file for configuring SSL proxy.

- `CLIENT_CERTKEYFILE`

  Path to the client private key file for configuring SSL proxy.

- `CLIENT_CLUSTER`

  IP address of the remote cluster server.

- `CLIENT_FLOATING`

IP address of the remote floating token server.

- `CLIENT_PASSWORD`

  Password for accessing the remote server.

- `CLIENT_PORT`

  Connection port of the remote server.

- `CLIENT_PRIORITY`

  Task priority for the remote cluster server. Possible values range from 0 to 99, with higher values indicating higher priority.

- `CLIENT_WAITTIME`

  Wait time of the client connection .

- `CLIENT_WEBSERVER`

  Domain name of the COPT Web License server.

- `CLIENT_WEBLICENSEID`

  Unique identifier for the Web License.

- `CLIENT_WEBACCESSKEY`

  Each Web License can have multiple API keys, with each API key having a unique `WebAccessKey`.

- `CLIENT_WEBTOKENDURATION`

  Validity period for each authorization from the Web License.

**Notes**

The above client configuration related parameters should be set within the API function `EnvrConfig` .

## 8.18 API function return code

When an API function finishes, it returns an integer **return code**, which indicates whether the API call was finished okay or failed. In case of failure, it specifies the reason of the failure.

Possible COPT API function return codes are listed below:

Table 8.4: COPT API Function Return Codes

| Return Codes | Value | Description |
|---|---|---|
| `OK` | 0 | The API call finished successfully |
| `MEMORY` | 1 | The API call failed because of memory allocation failure |
| `FILE` | 2 | The API call failed because of file input or output failure |
| `INVALID` | 3 | The API call failed because of invalid data |
| `LICENSE` | 4 | The API call failed because of license validation failure |
| `INTERNAL` | 5 | The API call failed because of internal error |
| `THREAD` | 6 | The API call failed because of thread error |
| `SERVER` | 7 | The API call failed because of remote server error |
| `NONCONVEX` | 8 | The API call failed because of problem is nonconvex” |
| `MEMORY_GPU` | 9 | The API call failed because of GPU memory allocation failure” |

## 8.19 Methods for accessing constants

In different programming interfaces, the ways of accessing constants are slightly different. In the C language interface, the constant name is prefixed with `"COPT_"` (like `COPT_MAXIMIZE`). For details, please refer to the corresponding chapters for each programming language API:

- C API: *C API Reference: Constants*
- C++ API: *C++ API Reference: Constants*
- C# API: *C# API Reference: General Constants*
- Java API: *Java API Reference: General Constants*
- Python API: *Python API Reference: General Constants*

# Chapter 9

# Attributes

To query and modify properties of a COPT model is through the attribute interface. A variety of different attributes are available, and they can be associated with solutions, or the model.

## 9.1 Problem related

Problem related attributes provide the relevant information of the model composition and description. The names and descriptions of these attributes are summarized below.

Table 9.1: Problem related attributes

| Name | Type | Description |
|---|---|---|
| *Cols* | Integer | Number of variables (columns) in the problem |
| *Bins* | Integer | Number of binary variables |
| *Ints* | Integer | Number of integer variables |
| *PSDCols* | Integer | Number of PSD variables in the problem |
| *Rows* | Integer | Number of constraints (rows) in the problem |
| *Indicators* | Integer | Number of indicator constraints |
| *Soss* | Integer | Number of SOS constraints |
| *Cones* | Integer | Number of Second-Order-Cone constraints |
| *ExpCones* | Integer | Number of exponential cone constraints |
| *AffineCones* | Integer | Number of affine cone constraints |
| *QConstrs* | Integer | Number of quadratic constraints |
| *PSDConstrs* | Integer | Number of PSD constraints |
| *LMIConstrs* | Integer | Number of LMI (Linear Matrix Inequalities) constraints |
| *NLConstrs* | Integer | Number of nonlinear expression constraints in the model |
| *Elems* | Integer | Number of non-zero elements in the coefficient matrix |
| *QElems* | Integer | Number of non-zero quadratic elements in the quadratic objective function |
| *PSDElems* | Integer | Number of PSD terms in objective function |
| *NLElems* | Integer | Number of nonlinear expression terms in the objective of the model |
| *SymMats* | Integer | Number of symmetric matrices in the problem |
| *MultiObjs* | Integer | Number of objectives in a multi-objective model |
| *ObjSense* | Integer | The optimization direction |
| *ObjConst* | Double | The constant part of the objective function |
| *HasQObj* | Integer | Whether the problem has quadratic terms in its objective |
| *HasPSDObj* | Integer | Whether the problem has PSD terms in its objective |
| *HasNLObj* | Integer | Whether the problem has nonlinear terms in its objective |
| *IsMIP* | Integer | Whether the problem is a MIP |

continues on next page

Table 9.1 – continued from previous page

| Name | Type | Description |
|---|---|---|
| *MinElem* | Double | Minimum absolute value of the non-zero elements of the coefficient matrix in the linear constraints |
| *MaxElem* | Double | Maximum absolute value of the non-zero elements of the coefficient matrix in the linear constraints |
| *MinBound* | Double | Minimum absolute value of the non-zero variable bounds |
| *MaxBound* | Double | Maximum absolute value of the non-zero variable bounds |
| *MinRHS* | Double | Minimum absolute value of the non-zero bounds (right-hand sides) of the linear constraints |
| *MaxRHS* | Double | Maximum absolute value of the non-zero bounds (right-hand sides) of the linear constraints |
| *MinCost* | Double | Minimum absolute value of the non-zero linear objective costs |
| *MaxCost* | Double | Maximum absolute value of the non-zero linear objective costs |
| *MinQElem* | Double | Minimum absolute value of the non-zero quadratic elements of the quadratic constraints |
| *MaxQElem* | Double | Maximum absolute value of the non-zero quadratic elements of the quadratic constraints |
| *MinQLElem* | Double | Minimum absolute value of the non-zero linear elements of the quadratic constraints |
| *MaxQLElem* | Double | Maximum absolute value of the non-zero linear elements of the quadratic constraints |
| *MinQRHS* | Double | Minimum absolute value of the non-zero right-hand sides of the quadratic constraints |
| *MaxQRHS* | Double | Maximum absolute value of the non-zero right-hand sides of the quadratic constraints |
| *MinQCost* | Double | Minimum absolute value of the non-zero quadratic objective costs |
| *MaxQCost* | Double | Maximum absolute value of the non-zero quadratic objective costs |

- `Cols`

  Integer attribute.

  Number of variables (columns) in the problem.

- `Bins`

  Integer attribute.

  Number of binary variables.

- `Ints`

  Integer attribute.

  Number of integer variables.

- `PSDCols`

  Integer attribute.

  Number of PSD variables in the problem.

- `Rows`

  Integer attribute.

  Number of constraints (rows) in the problem.

- `Indicators`

  Integer attribute.

  Number of indicator constraints.

- `Soss`

Integer attribute.

Number of SOS constraints.

- `Cones`

  Integer attribute.

  Number of Second-Order-Cone constraints.

- `ExpCones`

  Integer attribute.

  Number of exponential cone constraints.

- `AffineCones`

  Integer attribute.

  Number of affine cone constraints.

- `QConstrs`

  Integer attribute.

  Number of quadratic constraints.

- `PSDConstrs`

  Integer attribute.

  Number of PSD constraints.

- `LMIConstrs`

  Integer attribute.

  Number of LMI (Linear Matrix Inequalities) constraints.

- `NLConstrs`

  Integer attribute.

  The number of nonlinear expression constraints in the model.

- `Elems`

  Integer attribute.

  Number of non-zero elements in the coefficient matrix.

- `QElems`

  Integer attribute.

  Number of non-zero quadratic elements in the quadratic objective function.

- `PSDElems`

  Integer attribute.

  Number of PSD terms in objective function.

- `NLElems`

  Integer attribute.

  The number of nonlinear expression terms in the objective of the model.

- `SymMats`

  Integer attribute.

  Number of symmetric matrices in the problem.

- `MultiObjs`

Integer attribute.

Number of objectives in a multi-objective model.

- `ObjSense`

  Integer attribute.

  The optimization direction.

- `ObjConst`

  Double attribute.

  The constant part of the objective function.

- `HasQObj`

  Integer attribute.

  Whether the problem has quadratic terms in its objective.

- `HasPSDObj`

  Integer attribute.

  Whether the problem has PSD terms in its objective.

- `HasNLObj`

  Integer attribute.

  Whether the problem has nonlinear terms in its objective.

- `IsMIP`

  Integer attribute.

  Whether the problem is a MIP.

- `MinElem`

  Double attribute.

  Minimum absolute value of the non-zero elements of the coefficient matrix in the linear constraints.

- `MaxElem`

  Double attribute.

  Maximum absolute value of the non-zero elements of the coefficient matrix in the linear constraints.

- `MinBound`

  Double attribute.

  Minimum absolute value of the non-zero variable bounds.

  Bounds equal to zero and infinite bounds are not taken into account. A bound is treated as infinite when its absolute value reaches the infinite-bound value used by COPT, which is 1e30 by default.

- `MaxBound`

  Double attribute.

  Maximum absolute value of the non-zero variable bounds.

  Please refer to *MinBound* for details.

- `MinRHS`

Double attribute.

Minimum absolute value of the non-zero bounds (right-hand sides) of the linear constraints.

In COPT, a linear constraint is defined by a lower and an upper bound rather than by a single right-hand side, and both bounds are taken into account. A ranged constraint therefore contributes two values, while a constraint with a single sense contributes only its finite bound. Bounds equal to zero and infinite bounds are not taken into account.

- `MaxRHS`

  Double attribute.

  Maximum absolute value of the non-zero bounds (right-hand sides) of the linear constraints.

  Please refer to *MinRHS* for details.

- `MinCost`

  Double attribute.

  Minimum absolute value of the non-zero linear objective costs.

- `MaxCost`

  Double attribute.

  Maximum absolute value of the non-zero linear objective costs.

- `MinQElem`

  Double attribute.

  Minimum absolute value of the non-zero quadratic elements of the quadratic constraints.

  This attribute describes the quadratic constraints. For the quadratic objective function, whose elements are counted by *QElems*, see *MinQCost* and *MaxQCost*.

- `MaxQElem`

  Double attribute.

  Maximum absolute value of the non-zero quadratic elements of the quadratic constraints.

  Please refer to *MinQElem* for details.

- `MinQLElem`

  Double attribute.

  Minimum absolute value of the non-zero linear elements of the quadratic constraints.

- `MaxQLElem`

  Double attribute.

  Maximum absolute value of the non-zero linear elements of the quadratic constraints.

- `MinQRHS`

  Double attribute.

  Minimum absolute value of the non-zero right-hand sides of the quadratic constraints.

  Unlike a linear constraint, a quadratic constraint is defined by a single right-hand side together with a sense. Right-hand sides equal to zero and infinite right-hand sides are not taken into account.

- `MaxQRHS`

Double attribute.

Maximum absolute value of the non-zero right-hand sides of the quadratic constraints.

Please refer to *MinQRHS* for details.

- `MinQCost`

  Double attribute.

  Minimum absolute value of the non-zero quadratic objective costs.

  Coefficients are reported as supplied to the model, without applying any additional factor.

- `MaxQCost`

  Double attribute.

  Maximum absolute value of the non-zero quadratic objective costs.

  Please refer to *MinQCost* for details.

**Note**

The attributes from `MinElem` to `MaxQCost` describe the problem as it was last loaded into the solver. Because model modifications are buffered, they return 0.0 before the model is first loaded, and their previous values after any later modification. Load the model before querying them, for example by solving or updating it.

A value of 0.0 is also returned when the model contains no elements of the corresponding kind: all quadratic attributes do so for a model with neither quadratic constraints nor a quadratic objective function. Use `QConstrs` and `HasQObj` to tell the two cases apart.

These attributes are retrieved from the original problem, and will therefore generally differ from the ranges of coefficients printed in the solver log, which are those of the problem actually passed to the solver, after presolving, scaling and reformulation.

## 9.2 Solution related

Solution related attributes provide the relevant information of the solution composition and description. The names and descriptions of these attributes are summarized below.

Table 9.2: Solution related attributes

| Name | Type | Description |
|---|---|---|
| *Status* | Integer | Optimization status |
| *LpStatus* | Integer | (deprecated) The LP status |
| *MipStatus* | Integer | (deprecated) The MIP status |
| *HasSol* | Integer | Whether a solution is available |
| *HasLpSol* | Integer | (deprecated) Whether LP solution is available |
| *HasMipSol* | Integer | (deprecated) Whether MIP solution is available |
| *HasBasis* | Integer | Whether LP basis is available |
| *HasSensitivity* | Integer | Whether sensitivity analysis results are available for LP problem |
| *HasDualFarkas* | Integer | Whether the dual Farkas of an infeasible LP problem is available |
| *HasPrimalRay* | Integer | Whether the primal ray of an unbounded LP problem is available |
| *HasFeasRelaxSol* | Integer | Whether feasibility LP-relaxation solution is available |
| *HasIIS* | Integer | Whether IIS is available |
| *IsMinIIS* | Integer | Whether the computed IIS is minimal |
| *IISCols* | Integer | Number of bounds of columns in IIS |
| *IISRows* | Integer | Number of rows in IIS |

continues on next page

Table 9.2 – continued from previous page

| Name | Type | Description |
|---|---|---|
| *IISSOSs* | Integer | Number of SOS constraints in IIS |
| *IISIndicators* | Integer | Number of indicator constraints in IIS |
| *SimplexIter* | Integer | Number of simplex iterations performed |
| *BarrierIter* | Integer | Number of barrier iterations performed |
| *PDLPIter* | Integer | Number of first-order method (PDLP) iterations performed |
| *NodeCnt* | Integer | Number of explored nodes |
| *PoolSols* | Integer | Number of solutions in solution pool |
| *TuneResults* | Integer | Number of parameter tuning results |
| *ObjVal* | Double | Current best objective value |
| *ObjBound* | Double | Current best bound on the optimal objective value |
| *LpObjVal* | Double | (deprecated) Optimal objective value for continuous convex optimization problem |
| *BestObj* | Double | (deprecated) Current best objective value for MIP |
| *BestBnd* | Double | (deprecated) Current best bound for MIP |
| *BestGap* | Double | Current best relative gap for MIP |
| *FeasRelaxObj* | Double | Feasibility relaxation objective value |
| *SolvingTime* | Double | The time spent for the optimization (in seconds) |
| *MemPeak* | Double | Peak memory usage (MB) for the optimization |

- `Status`

  Integer attribute.

  Optimization status.

  Please refer to *General Constants: Solution status* for possible values.

- `LpStatus`

  Integer attribute.

  The LP status.

  Deprecated. Use *Status* instead.

- `MipStatus`

  Integer attribute.

  The MIP status.

  Deprecated. Use *Status* instead.

- `HasSol`

  Integer attribute.

  Whether a solution is available.

- `HasLpSol`

  Integer attribute.

  Whether LP solution is available.

  Deprecated. Use *HasSol* instead.

- `HasMipSol`

  Integer attribute.

  Whether MIP solution is available.

  Deprecated. Use *HasSol* instead.

- `HasBasis`

Integer attribute.

Whether LP basis is available.

- `HasSensitivity`

  Integer attribute.

  Whether sensitivity analysis results are available for LP problem.

- `HasDualFarkas`

  Integer attribute.

  Whether the dual Farkas of an infeasible LP problem is available.

- `HasPrimalRay`

  Integer attribute.

  Whether the primal ray of an unbounded LP problem is available.

- `HasFeasRelaxSol`

  Integer attribute.

  Whether feasibility LP-relaxation solution is available.

- `HasIIS`

  Integer attribute.

  Whether IIS is available.

- `IsMinIIS`

  Integer attribute.

  Whether the computed IIS is minimal.

- `IISCols`

  Integer attribute.

  Number of bounds of columns in IIS.

- `IISRows`

  Integer attribute.

  Number of rows in IIS.

- `IISSOSs`

  Integer attribute.

  Number of SOS constraints in IIS.

- `IISIndicators`

  Integer attribute.

  Number of indicator constraints in IIS.

- `SimplexIter`

  Integer attribute.

  Number of simplex iterations performed.

- `BarrierIter`

  Integer attribute.

  Number of barrier iterations performed.

- `PDLPIter`

Integer attribute.

Number of first-order method (PDLP) iterations performed.

- `NodeCnt`

  Integer attribute.

  Number of explored nodes.

- `PoolSols`

  Integer attribute.

  Number of solutions in solution pool.

- `TuneResults`

  Integer attribute.

  Number of parameter tuning results.

- `ObjVal`

  Double attribute.

  Current best objective value. When `Status` is `OPTIMAL`, this is the optimal objective value.

- `ObjBound`

  Double attribute.

  Current best bound on the optimal objective value.

- `LpObjVal`

  Double attribute.

  Optimal objective value for continuous convex optimization problem.

  Deprecated. Use *ObjVal* instead.

- `BestObj`

  Double attribute.

  Current best objective value for MIP.

  Deprecated. Use *ObjVal* instead.

- `BestBnd`

  Double attribute.

  Current best bound for MIP.

  Deprecated. Use *ObjBound* instead.

- `BestGap`

  Double attribute.

  Current best relative gap for MIP.

- `FeasRelaxObj`

  Double attribute.

  Feasibility relaxation objective value.

- `SolvingTime`

  Double attribute.

  The time spent for the optimization (in seconds).

- `MemPeak`

  Double attribute.

  Peak memory usage (MB) for the optimization.

**Note**

It is recommended to use `Status`, `HasSol`, `ObjVal` and `ObjBound` instead of the attributes marked `(deprecated)` above, as they provide a unified interface for querying the solution status, whether a solution is available, and optimal objective value across all supported problem types, including linear programming, integer programming, and convex/non-convex optimization problems.

## 9.3 Methods for accessing attributes

In different programming interfaces, the ways to access attributes are slightly different. For details, please refer to the corresponding chapters for each programming language API:

- C API: *C API Functions: Accessing attributes*
- C++ API: *C++ API Reference: Attributes*
- C# API: *C# API Reference: Attributes*
- Java API: *Java API Reference: Attributes*
- Python API: *Python API Reference: Attributes*

# Chapter 10

# Information

The information constants describe the relevant information of the model components (objective function, constraints and variables), solution results, and feasibility relaxation calculation results. This chapter will introduce the information constants provided by COPT and their meanings.

Table 10.1: COPT Information

| Name | Type | Description |
|---|---|---|
| *Obj* | Double | Objective cost of columns |
| *LB* | Double | Lower bounds of columns or rows |
| *UB* | Double | Upper bounds of columns or rows |
| *Value* | Double | Solution of columns |
| *Slack* | Double | Solution of slack variables, also known as activities of constraints. Only available for LP problem |
| *Dual* | Double | Solution of dual variables. Only available for LP problem |
| *RedCost* | Double | Reduced cost of columns. Only available for LP problem |
| *SAObjLow* | Double | Indicates the minimum value to which the objective coefficient of a variable can be reduced while keeping the current basis optimal |
| *SAObjUp* | Double | Indicates the maximum value to which the objective coefficient of a variable can be increased while keeping the current basis optimal |
| *SALBLow* | Double | Indicates the minimum value to which the lower bound of the variable/constraint can be reduced while keeping the current basis optimal |
| *SALBUp* | Double | Indicates the maximum value to which the lower bound of the variable/constraint can be increased while keeping the current basis optimal |
| *SAUBLow* | Double | Indicates the minimum value to which the upper bound of the variable/constraint can be reduced while keeping the current basis optimal |
| *SAUBUp* | Double | Indicates the maximum value to which the upper bound of the variable/constraint can be increased while keeping the current basis optimal |
| *DualFarkas* | Double | The dual Farkas for constraints of an infeasible LP problem |
| *PrimalRay* | Double | The primal ray for variables of an unbounded LP problem |
| *RelaxLB* | Double | Feasibility relaxation values for lower bounds of columns or rows |
| *RelaxUB* | Double | Feasibility relaxation values for upper bounds of columns or rows |
| *RelaxValue* | Double | Solutions for the original model variables (columns) in the feasibility relaxation model |

Table 10.2: COPT Callback Information

| Name | Type | Description |
|---|---|---|
| *BestObj* | Double | Current best objective |
| *BestBnd* | Double | Current best objective bound |
| *HasIncumbent* | Integer | Whether an incumbent is available |
| *Incumbent* | Double | Current best feasible solution |
| *MipCandObj* | Double | Objective value for current feasible solution candidate |
| *MipCandidate* | Double | Current feasible solution candidate |
| *RelaxSolObj* | Double | Current Objective of LP-relaxation |
| *RelaxSolution* | Double | Current solution of LP-relaxation |
| *NodeStatus* | Integer | The solution status of the LP-relaxation problem at the current node |

## 10.1 Problem information

- `Obj`

  Double information.

  Objective cost of columns.

- `LB`

  Double information.

  Lower bounds of columns or rows.

- `UB`

  Double information.

  Upper bounds of columns or rows.

## 10.2 Solution and sensitivity analysis information

- `Value`

  Double information.

  Solution of columns.

- `Slack`

  Double information.

  Solution of slack variables, also known as activities of constraints. Only available for LP problem.

- `Dual`

  Double information.

  Solution of dual variables. Only available for LP problem.

- `RedCost`

  Double information.

  Reduced cost of columns. Only available for LP problem.

- `SAObjLow`

Double information.

Sensitivity analysis information of the objective coefficient.

Indicates the minimum value to which the objective coefficient of a variable can be reduced while keeping the current basis optimal.

- `SAObjUp`

  Double information.

  Sensitivity analysis information of the objective coefficient.

  Indicates the maximum value to which the objective coefficient of a variable can be increased while keeping the current basis optimal.

- `SALBLow`

  Double information.

  Sensitivity analysis information of the lower bound of the variable/constraint.

  Indicates the minimum value to which the lower bound of the variable/constraint can be reduced while keeping the current basis optimal.

- `SALBUp`

  Double information.

  Sensitivity analysis information of the lower bound of the variable/constraint.

  Indicates the maximum value to which the lower bound of the variable/constraint can be increased while keeping the current basis optimal.

- `SAUBLow`

  Double information.

  Sensitivity analysis information of the upper bound of the variable/constraint.

  Indicates the minimum value to which the upper bound of the variable/constraint can be reduced while keeping the current basis optimal.

- `SAUBUp`

  Double information.

  Sensitivity analysis information of the upper bound of the variable/constraint.

  Indicates the maximum value to which the upper bound of the variable/constraint can be increased while keeping the current basis optimal.

## 10.3 Dual Farkas and primal ray

Advanced topic. When an LP is infeasible or unbounded, the solver can return the dual Farkas or primal ray to prove it.

- `DualFarkas`

  Double information.

  The dual Farkas for constraints of an infeasible LP problem. Please enable the parameter `"ReqFarkasRay"` to ensure that the dual Farkas is available when the LP is infeasible.

  Without loss of generality, the concept of the dual Farkas can be conveniently demonstrated using an LP problem with general variable bounds and equality constraints: $Ax = 0$ and $l \leq x \leq u$. When the LP is infeasible, a dual Farkas vector $y$ can prove that the system has conflict that $\max y^T Ax < y^T b = 0$. Computing $\max y^T Ax$: with the

vector $\hat{a} = y^T A$, choosing variable bound $x_i = l_i$ when $\hat{a}_i < 0$ and $x_i = u_i$ when $\hat{a}_i > 0$ gives the maximal possible value of $y^T Ax$ for any $x$ within their bounds.

Some application relies on the alternate conflict $\min \bar{y}^T Ax > \bar{y}^T b = 0$. This can be achieved by negating the dual Farkas, i.e. $\bar{y} = -y$ returned by the solver.

In very rare cases, the solver may fail to return a valid dual Farkas. For example when the LP problem slightly infeasible by tiny amount, which We recommend to study and to repair the infeasibility using FeasRelax instead.

- `PrimalRay`

  Double information.

  The primal ray for variables of an unbounded LP problem. Please enable the parameter `"ReqFarkasRay"` to ensure that the primal ray is available when an LP is unbounded.

  For a minimization LP problem in the standard form: $\min c^T x, Ax = b$ and $x \geq 0$, a primal ray vector $r$ satisfies that $r \geq 0, Ar = 0$ and $c^T r < 0$.

## 10.4 Feasibility relaxation information

- `RelaxLB`

  Double information.

  Feasibility relaxation values for lower bounds of columns or rows.

- `RelaxUB`

  Double information.

  Feasibility relaxation values for upper bounds of columns or rows.

- `RelaxValue`

  Double information.

  Solutions for the original model variables (columns) in the feasibility relaxation model.

## 10.5 Callback information

- `BestObj`

  Double information.

  Current best objective.

- `BestBnd`

  Double information.

  Current best objective bound.

- `HasIncumbent`

  Integer information.

  Whether an incumbent is available.

- `Incumbent`

  Double information.

  Current best feasible solution.

- `MipCandObj`

Double information.

Objective value for current feasible solution candidate.

- `MipCandidate`

  Double information.

  Current feasible solution candidate.

- `RelaxSolObj`

  Double information.

  Current Objective of LP-relaxation.

- `RelaxSolution`

  Double information.

  Current solution of LP-relaxation.

- `NodeStatus`

  Integer information.

  The solution status of the LP-relaxation problem at the current node. For possible values, please refer to: *General Constants Chapter: Solution Status (Part)*, except for `NODELIMIT`, `UNSTARTED`, `INF_OR_UNB` .

## 10.6 Methods for accessing information

In different programming interfaces, the ways to access and set information are slightly different. For details, please refer to the corresponding chapters for each programming language API:

- C API: *C API Functions: Accessing information of problem*
- C# API: *C# API Reference: Information*
- Java API: *Java API Reference: Information*
- Python API: *Python API Reference: Information*

# Chapter 11

# Parameters

Parameters control the operation of the **Cardinal Optimizer**. They can be modified before the optimization begins. Each parameter has its own default value and value range. Before starting the solution, user can set the parameters to different values, so as to put forward specific requirements on the solution algorithm and solution process. Obviously, the default settings can also be maintained.

According to the task performed by the solver COPT and the optimization problem solved, it can be divided into different types of parameters.

This chapter will introduce the parameters constants provided by COPT and their meanings.

## 11.1 Limits and tolerances

Limits parameters control the termination of the solving process. If the solving algorithm exceeds the value set by a limit parameter (such as solving time, number of nodes or number of iterations), COPT terminates the solve and returns a non-optimal solution status. For solution status, please refer to *General Constants: Solution status*.

Table 11.1: Limits parameters

| Name | Type | Description |
|---|---|---|
| *TimeLimit* | Double | Time limit of the optimization |
| *SolTimeLimit* | Double | Time limit if a primal feasible solution has been found |
| *NodeLimit* | Integer | Node limit of the optimization |
| *BarIterLimit* | Integer | Iteration limit of barrier method |
| *NLPIterLimit* | Integer | Iteration limit for the nonlinear solver |
| *MipNLPIterLimit* | Integer | Iteration limit for solving NLP problem(s) within the MIP solver |
| *MemLimit* | Double | Memory limit (MB) for the optimization |

Tolerance parameters specify the violations of feasibility and optimality that the solver is allowed to accept.

Table 11.2: Tolerances parameters

| Name | Type | Description |
|---|---|---|
| *MatrixTol* | Double | Input matrix coefficient tolerance |
| *QMatrixTol* | Double | Input Q matrix coefficient tolerance |
| *FeasTol* | Double | The feasibility tolerance |
| *DualTol* | Double | The tolerance for dual solutions and reduced cost |
| *IntTol* | Double | The integrality tolerance for variables |
| *PDLPTol* | Double | The PDLP tolerance. |
| *NLPTol* | Double | The NLP tolerance |
| *RelGap* | Double | The relative gap of optimization |
| *AbsGap* | Double | The absolute gap of optimization |

- `TimeLimit`

  Double parameter.

  Time limit (in seconds) of the optimization.

  **Default:** 1e20

  **Minimal:** 0

  **Maximal:** 1e20

- `SolTimeLimit`

  Double parameter.

  Time limit (in seconds) if a primal feasible solution has been found.

  **Default:** 1e20

  **Minimal:** 0

  **Maximal:** 1e20

- `NodeLimit`

  Integer parameter.

  Node limit of the optimization.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** `INT_MAX`

- `BarIterLimit`

  Integer parameter.

  Iteration limit of barrier method.

  **Default:** 500

  **Minimal:** 0

  **Maximal:** `INT_MAX`

- `NLPIterLimit`

  Integer parameter.

  Iteration limit for the nonlinear solver.

  **Default:** 1e4

  **Minimum:** 0

  **Maximum:** `INT_MAX`

- `MipNLPIterLimit`

  Integer parameter.

  Iteration limit for solving NLP problem(s) within the MIP solver.

  **Default:** 100

  **Minimal:** -1 (no limit)

  **Maximal:** `INT_MAX`

- `MemLimit`

  Double parameter.

  Memory limit (MB) for the optimization.

  **Default:** 1e20

  **Minimal:** 0

  **Maximal:** 1e20

- `MatrixTol`

  Double parameter.

  Input matrix coefficient tolerance.

  **Default:** 1e-10

  **Minimal:** 0

  **Maximal:** 1e-7

- `QMatrixTol`

  Double parameter.

  Input Q matrix coefficient tolerance.

  **Default:** 5e-14

  **Minimal:** 0

  **Maximal:** 1e-7

- `FeasTol`

  Double parameter.

  The feasibility tolerance.

  **Default:** 1e-6

  **Minimal:** 1e-9.

  **Maximal:** 1e-4

- `DualTol`

  Double parameter.

  The tolerance for dual solutions and reduced cost.

  **Default:** 1e-6

  **Minimal:** 1e-9

  **Maximal:** 1e-4

- `IntTol`

Double parameter.

The integrality tolerance for variables.

**Default:** 1e-6

**Minimal:** 1e-9

**Maximal:** 1e-1

- `PDLPTol`

  Double parameter.

  The PDLP tolerance.

  **Default:** 1e-6

  **Minimal:** 1e-12

  **Maximal:** 1e-4

- `NLPTol`

  Double parameter.

  The NLP tolerance.

  **Default:** 1e-8

  **Minimum:** 1e-13

  **Maximum:** 1e-3

- `RelGap`

  Double parameter.

  The relative gap of optimization.

  **Default:** 1e-4

  **Minimal:** 0

  **Maximal:** `DBL_MAX`

- `AbsGap`

  Double parameter.

  The absolute gap of optimization.

  **Default:** 1e-6

  **Minimal:** 0

  **Maximal:** `DBL_MAX`

**Note**

For integer programming problems, COPT defines the optimal absolute tolerance `AbsGap` as the absolute difference between the current primal bound (primalBound) and the dual bound (dualBound):

$$\text{AbsGap} = |\text{primalBound} - \text{dualBound}|. \tag{11.1}$$

For integer programming problems, COPT defines the optimal relative tolerance `RelGap` as the absolute difference between the current primal bound (primalBound) and the dual bound (dualBound), divided by the larger of their absolute values:

$$\text{RelGap} = \frac{\text{AbsGap}}{\max(|\text{primalBound}|, |\text{dualBound}|)}. \tag{11.2}$$

For minimization problems, primalBound denotes the current primal bound, namely the upper bound corresponding to the objective value of the best known feasible solution; dualBound denotes the current dual bound, namely the lower bound on the objective value.

For brevity, if primalBound and dualBound are denoted by primal and dual, respectively, the above definitions can be written as:

$$\text{AbsGap} = |\text{primal} - \text{dual}|, \tag{11.3}$$

$$\text{RelGap} = \frac{\text{AbsGap}}{\max(|\text{primal}|, |\text{dual}|)} = \frac{|\text{primal} - \text{dual}|}{\max(|\text{primal}|, |\text{dual}|)}. \tag{11.4}$$

When both `primal` and `dual` are nonzero, this definition is equivalent to taking the smaller value of the following two relative gaps:

$$\text{RelGap} = \min\left(\frac{\text{AbsGap}}{|\text{primal}|}, \frac{\text{AbsGap}}{|\text{dual}|}\right). \tag{11.5}$$

## 11.2 Presolving and scaling

Presolving parameters control the presolve algorithms of the solver.

Table 11.3: Presolving and scaling parameters

| Name | Type | Description |
|---|---|---|
| *Presolve* | Integer | Level of presolving before solving a model |
| *Scaling* | Integer | Whether to perform scaling before solving a problem |
| *Dualize* | Integer | Whether to dualize a problem before solving it |

- `Presolve`

  Integer parameter.

  Level of presolving before solving a model.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off.

  1: Fast.

  2: Normal.

  3: Aggressive.

  4: No Limitations, continues until the model cannot be modified (may be very time-consuming).

- `Scaling`

  Integer parameter.

  Whether to perform scaling before solving a problem.

  **Default:** -1

  **Possible values:**

-1: Choose automatically.

0: No scaling.

1: Apply scaling.

- `Dualize`

  Integer parameter.

  Whether to dualize a problem before solving it.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No dualizing.

  1: Dualizing the problem.

## 11.3 Linear programming related

Linear programming parameters control the workflow of the simplex, barrier and first-order (PDLP) methods.

Table 11.4: Linear programming related parameters

| Name | Type | Description |
|---|---|---|
| *LpMethod* | Integer | Method to solve the LP problem |
| *DualPrice* | Integer | Specifies the dual simplex pricing algorithm |
| *DualPerturb* | Integer | Whether to allow the objective function perturbation when using the dual simplex method |
| *BarHomogeneous* | Integer | Whether to use homogeneous self-dual form in barrier |
| *BarOrder* | Integer | Ordering algorithm in barrier method |
| *BarStart* | Integer | Algorithm for finding initial points in barrier method |
| *Crossover* | Integer | Whether to use crossover |
| *ReqFarkasRay* | Integer | Advanced topic. Whether to compute the dual Farkas or primal ray when the LP is infeasible or unbounded |
| *ReqSensitivity* | Integer | Whether to compute sensitivity analysis when an optimal basis is available for an LP problem (when solved by the simplex method or by other methods followed by crossover) |

- `LpMethod`

  Integer parameter.

  Method to solve the LP problem.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  For Linear Programming, choose dual simplex method;

  For Mixed Integer Linear Programming, choose dual simplex or barrier method.

  1: Dual simplex.

  2: Barrier.

  3: Crossover.

4: Concurrent (Use multiple algorithms simultaneously).

5: Choose between simplex and barrier automatically (Based on features such as sparsity and/or coefficients ranges).

6: First-order method (PDLP).

**Note**

Currently, COPT GPU solver supports solving Linear Programming problems and the root relaxation of Integer Programming problems. To enable it, either the first-order method (PDLP) or barrier method needs to be used, and *GPUMode* needs to be set accordingly.

- `DualPrice`

  Integer parameter.

  Specifies the dual simplex pricing algorithm.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Using Devex pricing algorithm.

  1: Using dual steepest-edge pricing algorithm.

- `DualPerturb`

  Integer parameter.

  Whether to allow the objective function perturbation when using the dual simplex method.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No perturbation.

  1: Allow objective function perturbation.

- `BarHomogeneous`

  Integer parameter.

  Whether to use homogeneous self-dual form in barrier.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No.

  1: Yes.

  For the effect of this parameter on the barrier algorithm's determination of infeasibility or unboundedness for LP models, please see the relevant discussion in *Barrier Logging*.

- `BarOrder`

  Integer parameter.

  Ordering algorithm in barrier method.

  **Default:** -1

**Possible values:**

-1: Choose automatically.

0: Approximate Minimum Degree (AMD).

1: Nested Dissection (ND1).

2: Modified Nested Dissection (ND2).

- `BarStart`

  Integer parameter.

  Algorithm for finding initial points in barrier method.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Simple.

  1: Mehrotra.

  2: Modified Mehrotra.

- `Crossover`

  Integer parameter.

  Whether to use crossover.

  **Default:** 1

  **Possible values:**

  -1: Choose automatically.

  Only run crossover when the LP solution is not primal-dual feasible.

  0: No.

  1: Yes.

- `ReqFarkasRay`

  Integer parameter.

  Advanced topic. Whether to compute the dual Farkas or primal ray when the LP is infeasible or unbounded.

  **Default:** 0

  **Possible values:**

  0: No.

  1: Yes.

- `ReqSensitivity`

  Integer parameter.

  Whether to compute sensitivity analysis when an optimal basis is available for an LP problem (when solved by the simplex method or by other methods followed by crossover).

  **Default:** 0

  **Possible values:**

  0: No.

  1: Yes.

## 11.4 Integer Programming related

Integer programming parameters control the workflow of the integer programming algorithms.

Table 11.5: Integer programming related parameters

| Name | Type | Description |
|---|---|---|
| *CutLevel* | Integer | Level of cutting-planes generation |
| *RootCutLevel* | Integer | Level of cutting-planes generation of root node |
| *TreeCutLevel* | Integer | Level of cutting-planes generation of search tree |
| *RootCutRounds* | Integer | Rounds of cutting-planes generation of root node |
| *NodeCutRounds* | Integer | Rounds of cutting-planes generation of search tree node |
| *HeurLevel* | Integer | Level of heuristics |
| *PreRootHeurLevel* | Integer | Level of pre-root heuristics |
| *RoundingHeurLevel* | Integer | Level of rounding heuristics |
| *DivingHeurLevel* | Integer | Level of diving heuristics |
| *SubMipHeurLevel* | Integer | Level of Sub-MIP heuristics |
| *FAPHeurLevel* | Integer | Level of Fix-and-propagate heuristics |
| *StrongBranching* | Integer | Level of strong branching |
| *ConflictAnalysis* | Integer | Whether to perform conflict analysis |
| *MipStartMode* | Integer | Mode of MIP starts |
| *MipStartNodeLimit* | Integer | Limit of nodes for MIP start sub-MIPs |
| *LinearizeIndicators* | Integer | Controls whether to force the linearization of indicator constraints |
| *LinearizeSos* | Integer | Controls whether to force the linearization of SOS constraints |
| *MipRepair* | Integer | Level for repairing the MIP solution in case of numerical issues |

- `CutLevel`

  Integer parameter.

  Level of cutting-planes generation.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `RootCutLevel`

  Integer parameter.

  Level of cutting-planes generation of root node.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

3: Aggressive

- `TreeCutLevel`

  Integer parameter.

  Level of cutting-planes generation of search tree.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `RootCutRounds`

  Integer parameter.

  Rounds of cutting-planes generation of root node.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** `INT_MAX`

- `NodeCutRounds`

  Integer parameter.

  Rounds of cutting-planes generation of search tree node.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** `INT_MAX`

- `HeurLevel`

  Integer parameter.

  Level of heuristics.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `PreRootHeurLevel`

  Integer parameter.

  Level of pre-root heuristics.

  **Default:** -1

  **Possible values:**

-1: Choose automatically.

0: Off

1: Fast

2: Normal

3: Aggressive

- `RoundingHeurLevel`

  Integer parameter.

  Level of rounding heuristics.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `DivingHeurLevel`

  Integer parameter.

  Level of diving heuristics.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `SubMipHeurLevel`

  Integer parameter.

  Level of Sub-MIP heuristics.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `FAPHeurLevel`

  Integer parameter.

  Level of Fix-and-propagate heuristics.

  **Default:** -1

**Possible values:**

-1: Choose automatically.

0: Off

1: Fast

2: Normal

3: Aggressive

4: Heavy

- `StrongBranching`

  Integer parameter.

  Level of strong branching.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `ConflictAnalysis`

  Integer parameter.

  Whether to perform conflict analysis.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No

  1: Yes

- `MipStartMode`

  Integer parameter.

  Mode of MIP starts.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Do not use any MIP starts.

  1: Only load full and feasible MIP starts.

  2: Only load feasible ones (complete partial solutions by solving subMIPs).

  3: Accept all MIP starts. Intended for hierarchical multi-objective optimization.

- `MipStartNodeLimit`

  Integer parameter.

  Limit of nodes for MIP start sub-MIPs.

  **Default:** -1 (Choose automatically)

**Minimal:** -1 (Choose automatically)

**Maximal:** `INT_MAX`

- `LinearizeIndicators`

  Integer parameter.

  Controls whether to force the linearization of Indicator constraints.

  **Default:** 0

  **Possible values**:

  0: Do not force linearization of indicator constraints.

  1: Force linearization of all indicator constraints.

  (All indicator constraints will be transformed into equivalent linear constraints if enabled.)

- `LinearizeSos`

  Integer parameter.

  Controls whether to force the linearization of SOS constraints.

  **Default:** 0

  **Possible values**:

  0: Do not force linearization of SOS constraints.

  1: Force linearization of all SOS constraints.

  (All SOS constraints will be transformed into equivalent linear constraints if enabled.)

- `MipRepair`

  Integer parameter.

  Level for repairing the MIP solution in case of numerical issues.

  **Default:** -1

  **Possible values:**

  -1: Only when time left.

  0: Off.

  1: Extend time limit and attempt repair (Fast).

  2: Extend time limit and attempt repair (Normal).

  3: Extend time limit and attempt repair (Aggressive).

## 11.5 Semidefinite Programming related

Semidefinite programming parameters control the workflow of the semidefinite programming algorithms.

- `SDPMethod`

  Integer parameter.

  Method to solve the SDP problem.

  **Default:** -1

  **Possible values:**

-1: Choose automatically.

0: Primal-Dual method.

1: ADMM method.

2: Dual method.

## 11.6 Nonlinear Programming related

Nonlinear programming parameters control the workflow of nonlinear programming solvers.

Table 11.6: Overview of Nonlinear Programming Parameters

| Parameter | Type | Description |
|---|---|---|
| *NonConvex* | Integer | Handling strategy for nonconvex models |
| *NLPMuUpdate* | Integer | Barrier parameter update strategy of the nonlinear solver |
| *NLPLinScale* | Integer | Linear system scaling strategy of the nonlinear solver |

- `NonConvex`

  Integer parameter.

  Handling strategy for continuous nonconvex models.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Report nonconvexity and terminate.

  1: Search for a local optimal solution.

  2: Search for a global optimal solution.

- `NLPMuUpdate`

  Integer parameter.

  Barrier parameter update strategy of the nonlinear solver.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Monotonically decreasing.

  1: Adaptive adjustment.

- `NLPLinScale`

  Integer parameter.

  Linear system scaling strategy of the nonlinear solver.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No scaling.

  1: Always scale.

## 11.7 Multi-objective Optimization related

Multi-objective optimization parameters control the time limit for solving the whole multi-objective model and how solver parameters are applied to the model of each objective.

- `MultiObjTimeLimit`

  Double parameter.

  Time limit (in seconds) for solving the whole multi-objective model.

  **Default:** 1e20

  **Minimum:** 0

  **Maximum:** 1e20

- `MultiObjParamMode`

  Integer parameter.

  Specifies how solver parameters are applied to models in multi-objective optimization.

  **Default:** 0

  **Possible values:**

  0: Use the same parameters for all objectives.

  1: Use individual solver parameters for each objective.

## 11.8 Parallel computing related

Parallel computing parameters control the number of threads and tasks used by the solving algorithms.

Table 11.7: Parallel computing related parameters

| Name | Type | Description |
|---|---|---|
| *Threads* | Integer | Number of threads to use |
| *BarThreads* | Integer | Number of threads used by barrier. If value is -1, the thread count is determined by parameter `Threads` |
| *SimplexThreads* | Integer | Number of threads used by dual simplex. If value is -1, the thread count is determined by parameter `Threads` |
| *CrossoverThreads* | Integer | Number of threads used by crossover. If value is -1, the thread count is determined by parameter `Threads` |
| *MipTasks* | Integer | Number of MIP tasks in parallel |
| *ConcurrentLpMode* | Integer | The LP concurrent solving mode |

- `Threads`

  Integer parameter.

  Number of threads to use.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** 128

- `BarThreads`

  Integer parameter.

  Number of threads used by barrier. If value is -1, the thread count is determined by parameter `Threads`.

**Default:** -1

**Minimal:** -1

**Maximal:** 128

- `SimplexThreads`

  Integer parameter.

  Number of threads used by dual simplex. If value is -1, the thread count is determined by parameter `Threads`.

  **Default:** -1

  **Minimal:** -1

  **Maximal:** 128

- `CrossoverThreads`

  Integer parameter.

  Number of threads used by crossover. If value is -1, the thread count is determined by parameter `Threads`.

  **Default:** -1

  **Minimal:** -1

  **Maximal:** 128

- `MipTasks`

  Integer parameter.

  Number of MIP tasks in parallel.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** 256

- `ConcurrentLpMode`

  Integer parameter.

  The LP concurrent solving mode.

  Only effective when `LpMethod = 4`.

  When `LpMethod = 4` is enabled, the parameters `GPUMode` and `GPUDevice` are ignored. GPU usage and device selection are fully controlled by `ConcurrentLpMode`.

  **Default:** -1 (Choose automatically)

  **Possible values:**

  -1: Choose automatically.

  0: CPU only (simplex + barrier).

  1: CPU (simplex + barrier) + GPU first-order method PDLP (uses GPU device 0).

  2: CPU (simplex + barrier) + GPU barrier (uses GPU device 0).

  3: CPU (simplex + barrier) + GPU first-order method PDLP + GPU barrier (uses GPU device 0 and GPU device 1).

**Note**

If `ConcurrentLpMode` is set to 1 or 2 and no GPU device is detected, or if it is set to 3 and fewer than two GPU devices are detected, the solver reports an error and terminates.

## 11.9 GPU computing related

GPU computing parameters control how the GPU solving mode is used and which GPU device is selected on a machine with multiple GPUs.

- `GPUMode`

  Integer parameter.

  Specifies GPU mode.

  **Default:** -1

  **Default:** -1

  **Possible values:**

  -1: Choose automatically. The first-order method (PDLP) will attempt to use the GPU by default, while the barrier method will use the CPU by default.

  0: Force CPU mode.

  1: Attempt to use the standard GPU mode.

  2: For the barrier method, attempt to use the high-performance GPU mode when solving LP problems, which may lead to higher memory usage. For the first-order method (PDLP), this is equivalent to `GPUMode=1` (standard GPU mode).

**Notes**

1. COPT's GPU solver supports solving LP and the root relaxation of MILP problems. It is only effective when using either the first-order method (PDLP) or the barrier method.
2. COPT's GPU solver also supports solving SOCP, Q(C)P, Exponential Cone Programming, and SDP problems. It is only effective when using the barrier method.
3. On platforms such as Windows, Linux-x86, and Linux-aarch64, when `GPUMode` is set to an appropriate value, COPT will attempt to detect whether the required CUDA libraries are available and whether a supported GPU is present. If both conditions are met, COPT will run in GPU mode; otherwise, it will fall back to CPU mode. The startup messages differ slightly between the two modes.
4. On platforms such as macOS, only the CPU version of COPT is currently available. Even if `GPUMode=1` or `2` is set, COPT will still run in CPU mode. The startup messages differ slightly between the two modes.
5. Installing CUDA libraries is not required for running COPT. If GPU mode is not enabled, COPT will function normally regardless of whether the CUDA libraries are installed. For more information about installing CUDA libraries, see *FAQ - GPU Usage*.

- `GPUDevice`

  Integer parameter.

  Specifies GPU device to use (in cases where the running machine has multiple GPUs).

**Default:** -1 (Choose automatically)

**Minimal:** -1 (Choose automatically)

**Maximal:** `INT_MAX`

## 11.10 IIS computation related

IIS computation parameters control the method used to compute the IIS of an infeasible model.

- `IISMethod`

  Integer parameter.

  Method for IIS computation.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Find smaller IIS.

  1: Find IIS quickly.

## 11.11 Feasibility relaxation related

Feasibility relaxation parameters control the method used to compute the feasibility relaxation.

- `FeasRelaxMode`

  Integer parameter.

  Method for feasibility relaxation.

  **Default:** 0

  **Possible values:**

  0: Minimize sum of violations.

  1: Optimize original objective function under minimal sum of violations.

  2: Minimize number of violations.

  3: Optimize original objective function under minimal number of violations.

  4: Minimize sum of squared violations.

  5: Optimize original objective function under minimal sum of squared violations.

**Notices**

- Either `FeasRelaxMode=4` or `FeasRelaxMode=5` does not support MILP but only LP. Other methods support both two types of problems.

## 11.12 Parameter Tuning related

Parameter tuning parameters control the method, mode, termination targets, result measurement and log output level of parameter tuning.

Table 11.8: Tuner related parameters

| Name | Type | Description |
|---|---|---|
| *TuneTimeLimit* | Double | Time limit for parameter tuning |
| *TuneTargetTime* | Double | Time target for parameter tuning |
| *TuneTargetRelGap* | Double | Optimal relative tolerance target for parameter tuning |
| *TuneMethod* | Integer | Method for parameter tuning |
| *TuneMode* | Integer | Mode for parameter tuning |
| *TuneMeasure* | Integer | Parameter tuning result calculation method |
| *TunePermutes* | Integer | Permutations for each trial parameter set |
| *TuneOutputLevel* | Integer | Parameter tuning log output intensity |

- `TuneTimeLimit`

  Double parameter.

  Time limit (in seconds) for parameter tuning. If the parameter value is 0, it will be automatically set by the solver.

  **Default:** 0

  **Minimal:** 0

  **Maximal:** 1e20

- `TuneTargetTime`

  Double parameter.

  Time target for parameter tuning.

  **Default:** 1e-2

  **Minimal:** 0

  **Maximal:** `DBL_MAX`

- `TuneTargetRelGap`

  Double parameter.

  Optimal relative tolerance target for parameter tuning.

  **Default:** 1e-4

  **Minimal:** 0

  **Maximal:** `DBL_MAX`

- `TuneMethod`

  Integer parameter.

  Method for parameter tuning.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Greedy search strategy.

  1: Broader search strategy.

- `TuneMode`

  Integer parameter.

  Mode for parameter tuning.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Solving time.

  1: Optimal relative tolerance.

  2: Objective function value.

  3: The lower bound of the objective function value.

- `TuneMeasure`

  Integer parameter.

  Parameter tuning result calculation method.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Calculate the average value.

  1: Calculate the maximum value.

- `TunePermutes`

  Integer parameter.

  Permutations for each trial parameter set. If the parameter value is 0, it will be automatically set by the solver.

  **Default:** 0

  **Minimal:** 0

  **Maximal:** `INT_MAX`

- `TuneOutputLevel`

  Integer parameter.

  Parameter tuning log output level.

  **Default:** 2

  **Possible values:**

  0: No output of tuning log.

  1: Output only a summary of the improved parameters.

  2: Output a summary of each tuning attempt.

  3: Output a detailed log of each tuning attempt.

## 11.13 Callback related

Callback parameters control whether lazy constraints are part of the model when callbacks are used.

- `LazyConstraints`

  Integer parameter.

  Whether lazy constraints are part of the model.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No.

  1: Yes.

**Notes**

- This parameter only affects MIP.

## 11.14 Other parameters

Other parameters control how and in what detail the optimization log is presented, and whether the dual problem is written out.

- `Logging`

  Integer parameter.

  Whether to print optimization logs.

  **Default:** 1

  **Possible values:**

  0: No optimization logs.

  1: Print optimization logs.

- `LogLevel`

  Integer parameter.

  Controls the level of detail in the optimization logs.

  **Default:** 2

  **Possible values:**

  2: Print basic optimization logs.

  3: Print memory usage information in addition to basic optimization logs (for MIP problems).

- `LogToConsole`

  Integer parameter.

  Whether to print optimization logs to console.

  **Default:** 1

  **Possible values:**

0: No optimization logs to console.

1: Print optimization logs to console.

- `WriteDualProb`

  Integer parameter.

  Controls whether the dual of the current linear programming problem is written.

  When writing a linear programming problem in `LP` / `MPS` / `BIN` format, this parameter specifies whether the original model or the dual model is written.

  **Default:** 0

  **Possible values:**

  0: Write the original model.

  1: Write the dual model.

## 11.15 Methods for accessing and setting parameters

In different programming interfaces, the ways to access and set parameters are slightly different. For details, please refer to the corresponding chapters for each programming language API:

- C API: *C API Functions: Accessing and setting parameters*
- C++ API: *C++ API Reference: Parameters*
- C# API: *C# API Reference: Parameters*
- Java API: *Java API Reference: Parameters*
- Python API: *Python API Reference: Parameters*

# Chapter 12

# Modeling and Solving Optimization Problems

This chapter will introduce how to model and solve different types of optimization problems using the COPT solver. The content of the chapter is structured as follows:

- *Linear Programming (LP)*
- *Second-Order Cone Programming (SOCP)*
- *Exponential Cone Programming (ExpCone Programming)*
- *Semidefinite Programming (SDP)*
- *Quadratic Programming (QP)*
- *Quadratically Constrained Programming (QCP)*
- *General Nonlinear Programming (NLP)*
- *Mixed-Integer Programming (MIP)*
- *Special Constraints*

The types of problems and corresponding optimization algorithms currently supported by the COPT solver are shown in Table 1.1:

Table 12.1: Supported problem types and available algorithms

| **Problem type** | Available algorithms |
|---|---|
| Linear Programming (LP) | Simplex, Barrier (CPU/GPU), First-order Method (PDLP)(CPU/GPU) |
| Second-Order-Cone Programming (SOCP) | Barrier (CPU/GPU) |
| Exponential Cone Programming (ExpCone) | Barrier (CPU/GPU) |
| Convex Quadratic Programming (QP) | Barrier (CPU/GPU) |
| Convex Quadratically Constrained Programming (QCP) | Barrier (CPU/GPU) |
| Semidefinite Programming (SDP) | Barrier (CPU/GPU), ADMM |
| Nonconvex Quadratic Programming (QP) | Spatial Branch-and-Bound (global optimum), Barrier (local optimum) |
| Nonconvex Quadratically Constrained Programming (QCP) | Spatial Branch-and-Bound (global optimum), Barrier (local optimum) |
| General Nonlinear Programming (NLP) | Barrier (local optimum) |
| Mixed Integer Linear Programming (MILP) | Branch-and-Cut |
| Mixed Integer Second-Order-Cone Programming (MISOCP) | Branch-and-Cut |
| Mixed Integer Convex/Nonconvex Quadratic Programming (MIQP) | Branch-and-Cut |
| Mixed Integer Convex/Nonconvex Quadratically Constrained Programming (MIQCP) | Branch-and-Cut |

COPT supports three methods of inputting optimization problems: model file reader, programming interface, and third-party tools.

1. For model file input, please refer to *File Formats*.
2. The programming language interfaces supported by COPT include:
   - C
   - Python
   - C++
   - C#
   - Java
   - Fortran
3. The main third-party tools interfaces currently supported by COPT include:
   - AMPL
   - AIMMS
   - GAMS
   - Julia
   - Pyomo
   - PuLP
   - CVXPY

## 12.1 Linear Programming (LP)

Linear Programming, as the most fundamental and important branch of operations research, has a wide range of applications. The objective function and constraints of a linear programming problem are both linear.

### 12.1.1 Mathematical Model

The mathematical formulation is as follows:

$$\begin{aligned} \min \quad & \sum_{i=0}^{m-1} c_j x_j + c^f \\ \text{s.t.} \quad & l_i^c \le \sum_{j=0}^{n-1} a_{ij} x_j \le u_i^c, \qquad i = 1, 2, \cdots, m-1 \\ & l_j^v \le x_j \le u_j^v, \qquad j = 1, 2, \cdots, n-1 \end{aligned} \tag{12.1}$$

Alternatively, the model can be concisely expressed using vectors and matrices:

$$\begin{aligned} \min \quad & c^T x + c^f \\ \text{s.t.} \quad & l^c \le Ax \le u^c \\ & l^v \le x \le u^v \end{aligned} \tag{12.2}$$

The variables and parameters in the model have the following meanings:

- Problem size: $m$ represents the number of constraints, and $n$ represents the number of decision variables.
- Decision variables: $x = (x_j)_{j=0}^{n-1} \in \mathbb{R}^n$
- Variable bounds: $l^v, u^v \in \mathbb{R}^n$, where $l^v$ represents the lower bounds and $u^v$ represents the upper bounds of the variables.
- Constraint bounds: $l^c, u^c \in \mathbb{R}^m$, where $l^c$ represents the lower bounds and $u^c$ represents the upper bounds of the constraints.
- Coefficient matrix for linear constraints: $A = (a_{ij})_{m \times n} \in \mathbb{R}^{m \times n}$
- Coefficients of variables in the objective function: $c \in \mathbb{R}^n$ represents the coefficients of the variables in the objective function, and $c^f$ represents the constant term in the objective function.

### 12.1.2 Modeling

The basic steps to build and solve a Linear Programming (LP) model in COPT are as follows:

1. Create the COPT environment and model.
2. Add required data.
3. Construct the linear programming model:
   - Add decision variables.
   - Add linear constraints.
   - Set the linear objective function.
4. Set optimization parameters and solve the model.
5. Retrieve the solution results.

### Modeling: Adding Linear Constraints

The COPT provides three methods to add linear constraints:

1. Add a single linear constraint to the model.
2. Batch add a group of linear constraints.
3. Add a single linear constraint with lower and upper bounds.

When adding linear constraints to the model, the main parameters that can be specified are:

- `expr`/ `builder` : Linear constraint expression or linear constraint builder.
- `sense`: Type of constraint. For possible values, please refer to *Constants Section: Constraint Type*.
- `name`: Name of the linear constraint.

If adding a linear constraint with bounds, the following must also be specified:

- `lb`: Lower bound of the linear constraint.
- `ub`: Upper bound of the linear constraint.

The implementation methods in different programming interfaces are shown in Table 12.2:

Table 12.2: Functions for Adding Linear Constraints

| API | Add a single constraint | Add a group of constraints in batch |
|---|---|---|
| C | `COPT_AddRow` | `COPT_AddRows` |
| C++ | `Model::AddConstr()` | `Model::AddConstrs()` |
| C# | `Model.AddConstr()` | `Model.AddConstrs()` |
| Java | `Model.addConstr()` | `Model.addConstrs()` |
| Python | `Model.addConstr()` | `Model.addConstrs()` |

**Notes**

- For linear constraint operations, the function names, calling methods, and **argument names may vary slightly** across different programming interfaces, but the functionality and argument meanings are consistent.
- COPT supports defining constraints using constraint types, but we **recommend defining constraints directly using bounds**.
- In the C API, linear constraints are added using the coefficient matrix as an input argument.
- In the Python API, an additional method for adding linear constraints with bounds is provided: `Model.addBoundConstr()`.

**Linear Programming Problem Example**

$$\begin{aligned} &\text{Maximize:} \\ &\qquad 1.2x + 1.8y + 2.1z \\ &\text{Subject to:} \\ &\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\ &\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\ &\text{Bounds:} \\ &\qquad 0.1 \leq x \leq 0.6 \\ &\qquad 0.2 \leq y \leq 1.5 \\ &\qquad 0.3 \leq z \leq 2.8 \end{aligned} \tag{12.3}$$

For corresponding code implementations in different programming interfaces, please refer to: *COPT Quick Start Section*.

In the programming interfaces provided by COPT, except for the C language, the other object-oriented programming interfaces (C#, C++, Java, Python) offer classes related to linear constraints:

- Encapsulation of operations related to linear constraints:
  1. `Constraint` class: Encapsulation of operations related to linear constraints in COPT.
  2. `ConstrArray` class: Facilitates operations on a group of `Constraint` class objects.
- Encapsulation of linear constraint builders:
  1. `ConstrBuilder` class: Encapsulation of linear constraint builders in COPT.
  2. `ConstrBuilderArray` class: Facilitates operations on a group of `ConstrBuilder` class objects.
- C++ API: *Constraint Class* , *ConstrArray Class* , *ConstrBuilder Class* , *ConstrBuilderArray Class*
- C# API: *Constraint Class* , *ConstrArray Class* , *ConstrBuilder Class* , *ConstrBuilderArray Class*
- Java API: *Constraint Class* , *ConstrArray Class* , *ConstrBuilder Class* , *ConstrBuilderArray Class*
- Python API: *Constraint Class* , *ConstrArray Class* , *ConstrBuilder Class* , *ConstrBuilderArray Class*

### 12.1.3 Solving

For linear programming problems, the COPT solver provides the Simplex method and Barrier method. The specific method can be selected by setting the optimization parameter `"LpMethod"`. By configuring other related optimization parameters for linear programming, you can control the detailed workflow of the solving algorithm. For more details, please refer to *Parameter: Linear Programming Related*.

For the solving logs of linear programming problems, please refer to *Logging Section: Simplex Method* and *Interior Point Method*.

### 12.1.4 Related Attributes and Information

**Linear Programming Related Attributes**

Attributes for linear programming are shown in Table 12.3:

Table 12.3: Attributes for Linear Programming

| Name | Type | Description |
|---|---|---|
| `Cols` | Integer | Number of variables (columns) in the problem |
| `Rows` | Integer | Number of constraints (rows) in the problem |
| `Elems` | Integer | Number of non-zero elements in the coefficient matrix |
| `LpObjVal` | Double | The LP objective value |
| `SimplexIter` | Integer | Number of simplex iterations performed |
| `BarrierIter` | Integer | Number of barrier iterations performed |

Attributes for results of a linear programming problem are shown in Table 12.4:

Table 12.4: Attributes for Linear Programming Results

| Name | Type | Description |
|---|---|---|
| `LpStatus` | Integer | The LP status |
| `HasLpSol` | Integer | Whether LP solution is available |
| `HasBasis` | Integer | Whether LP basis is available |
| `LpObjVal` | Double | The LP objective value |

For the results of a linear programming problem, COPT also provides relevant information constants, as shown in Table 12.5:

Table 12.5: Information for Linear Programming

| Name | Type | Description |
|---|---|---|
| `Slack` | Double | Solution of slack variables, also known as activities of constraints. Only available for LP problem |
| `Dual` | Double | Solution of dual variables. Only available for LP problem |
| `RedCost` | Double | Reduced cost of columns. Only available for LP problem |

For different programming interfaces, please refer to *Attributes* and *Information* sections to see how to access these attributes and information.

## 12.2 Second-Order Cone Programming (SOCP)

Second-Order Cone Programming (SOCP) is an optimization problem where the objective function is linear, and the constraints include second-order cone.

### 12.2.1 Mathematical Model

Second-Order Cone (SOC):

$$\mathcal{Q}^{n+1} = \{(t, x) \in \mathbb{R} \times \mathbb{R}^n \mid t \geq \|x\|_2\} \tag{12.4}$$

**Second-Order Cone Constraint:**

When $t \in \mathbb{R}$ and $x \in \mathbb{R}^n$ are decision variables, a constraint of the form $t \geq \|x\|_2$ is called a second-order cone constraint.

The mathematical formulation is as follows:

$$\begin{aligned} \min \quad & c^T x + c^f \\ \text{s.t.} \quad & l^c \leq Ax \leq u^c \\ & l^v \leq x \leq u^v \\ & Fx + g \in \mathcal{Q} \end{aligned} \tag{12.5}$$

The variables and arguments in the model have the following meanings:

- Decision variables: $x = (x_j)_{j=0}^{n-1} \in \mathbb{R}^n$
- Decision variable bounds: $l^v, u^v \in \mathbb{R}^n$, where $l^v$ represents the lower bounds, and $u^v$ represents the upper bounds of the variables.
- Constraint boundaries: $l^c, u^c \in \mathbb{R}^m$, where $l^c$ represents the lower bounds, and $u^c$ represents the upper bounds of the linear constraints.
- Coefficient matrix of linear constraints: $A = (a_{ij})_{m \times n} \in \mathbb{R}^{m \times n}$
- Coefficients in the objective function: $c \in \mathbb{R}^n$ represents the coefficients of the variables in the objective function, and $c^f$ represents the constant term in the objective function.
- Problem size: $m$ represents the number of linear constraints, $n$ represents the number of decision variables, and $k$ represents the number of second-order cone constraints.

The following are the meanings of arguments related to the cone constraints:

- $F$: $F \in \mathbb{R}^{k \times n}$ is the coefficient matrix of the cone constraints.
- $g$: $g \in \mathbb{R}^k$ is the constant vector in the cone constraints.
- $\mathcal{Q}$: Represents the Cartesian product of $k$ sets, $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2 \times \cdots \times \mathcal{Q}_p$, where $p$ represents the number of second-order cone constraints, and each $\mathcal{Q}_i, i \in \{1, 2, \cdots, p\}$ represents a second-order cone.

### 12.2.2 Modeling

The basic steps to construct and solve a Second-Order Cone Programming (SOCP) model in COPT are as follows:

1. Create the COPT environment and model.
2. Add required data.
3. Construct the SOCP model:
   - Add decision variables.
   - Add constraints (second-order cone constraints, linear constraints).
   - Set the objective function.
4. Set solver parameters and solve.
5. Retrieve the solution results.

**Modeling: Adding Second-Order Cone Constraints**

COPT supports modeling the following two types of second-order cone constraints:

**Standard Second-Order Cone**

$$Q^n = \left\{ x \in \mathbb{R}^n \;\middle|\; x_0 \geq \sqrt{\sum_{i=1}^{n-1} x_i^2}, x_0 \geq 0 \right\} \tag{12.6}$$

Constant representation: `CONE_QUAD`

**Rotated Second-Order Cone**

$$Q_r^n = \left\{ x \in \mathbb{R}^n \;\middle|\; 2x_0 x_1 \geq \sum_{i=2}^{n-1} x_i^2, x_0 \geq 0, x_1 \geq 0 \right\} \tag{12.7}$$

Constant representation: `CONE_RQUAD`

When adding second-order cone constraints to the model, the main function arguments that can be specified by the user are:

- `ctype`: The type of second-order cone constraint.
  - `CONE_QUAD` : Standard second-order cone.
  - `CONE_RQUAD`: Rotated second-order cone.
- `cvars`: Variables that form the second-order cone constraint.
- `dim`: Dimension of the second-order cone constraint.

The implementation in different programming interfaces is shown in the table below:

Table 12.6: Functions for Adding Second-Order Cone

| API | Function |
|---|---|
| C | `COPT_AddCones` |
| C++ | `Model::AddCone()` |
| C# | `Model.addCone()` |
| Java | `Model.addCone()` |
| Python | `Model.addCone()` |

**Note**

- The function names, calling methods, and **argument names differ slightly** in different programming interfaces for operations related to modeling second-order cone constraints, but the functionality and argument meanings are consistent.
- When providing variables that form the second-order cone constraint, please input them sequentially according to their order in the constraint expression.
- In the C API, the members of the second-order cone constraints are provided in compressed row storage format. For more information on sparse matrix compressed storage in the C API, please refer to the specific example in the relevant section.
- In the Python API, an additional member method `Model.addConeByDim()` is provided for specifying the dimension of the second-order cone.

### Example of Second-Order Cone Constraints

A standard second-order cone formed by $x_4, x_1, x_2, x_3$:

$$x_4 \geq \sqrt{x_1^2 + x_2^2 + x_3^2} \tag{12.8}$$

A rotated second-order cone formed by $x_3, x_4, x_1, x_2$:

$$2x_3x_4 \geq x_1^2 + x_2^2 \tag{12.9}$$

Taking the standard second-order cone formed by $x_4, x_1, x_2, x_3$ as an example, the code implementation in different programming interfaces is as follows:

**C Interface:**

```
int ncone = 1;
int conetype[] = {COPT_CONE_QUAD};
int conebeg[] = {0};
int conecnt[] = {4};
int coneind[] = {3, 0, 1, 2};
COPT_AddCones(prob, ncone, conetype, conebeg, conecnt, coneind)
```

**C++ Interface:**

```
VarArray cvars;
cvars.PushBack(x4);
cvars.PushBack(x1);
cvars.PushBack(x2);
cvars.PushBack(x3);
model.AddCone(cvars, COPT_CONE_QUAD);
```

**C# Interface:**

```
VarArray cvars = new VarArray();
cvars.PushBack(x4);
cvars.PushBack(x1);
cvars.PushBack(x2);
cvars.PushBack(x3);
model.AddCone(cvars, Copt.Consts.CONE_QUAD);
```

**Java Interface:**

```
VarArray cvars = new VarArray();
cvars.PushBack(x4);
cvars.PushBack(x1);
cvars.PushBack(x2);
cvars.PushBack(x3);
model.addCone(cvars, copt.Consts.CONE_QUAD);
```

**Python Interface:**

```
model.addCone([x4, x1, x2, x3], COPT.CONE_QUAD)
```

In the programming interfaces provided by COPT, with the exception of the C language, object-oriented programming interfaces (C#, C++, Java, Python) provide classes related to second-order cone constraints:

- **Encapsulation of operations related to second-order cone constraints:**
  1. `Cone` class: Encapsulation of operations related to second-order cone constraints in COPT.
  2. `ConeArray` class: Conveniently allows users to operate on a group of `Cone` objects.
- **Encapsulation of second-order cone constraint builders:**
  1. `ConeBuilder` class: Encapsulation of builders for constructing second-order cone constraints in COPT.
  2. `ConeBuilderArray` class: Conveniently allows users to operate on a group of `ConeBuilder` objects.
- C++ API: *Cone Class* , *ConeArray Class* , *ConeBuilder Class* , *ConeBuilderArray Class*
- C# API: *Cone Class* , *ConeArray Class* , *ConeBuilder Class* , *ConeBuilderArray Class*
- Java API: *Cone Class* , *ConeArray Class* , *ConeBuilder Class* , *ConeBuilderArray Class*
- Python API: *Cone Class* , *ConeArray Class* , *ConeBuilder Class* , *ConeBuilderArray Class*

**Attributes Related to Second-Order Cone Constraints**

- `COPT_INTATTR_CONES` or `"Cones"`

  Integer attribute. The number of second-order cone constraints in the model.

This attribute provides the count of second-order cone constraints that have been added to the model. It can be useful for monitoring or validating the structure of the model during or after the modeling process.

## 12.3 ExpCone Programming

### 12.3.1 Mathematical Formulation

COPT supports two types of exponential cone constraints:

- `EXPCONE_PRIMAL` : Primal exponential cone

$$\mathrm{cl}(S_1) = S_1 \cup S_2 \tag{12.10}$$

$$\begin{aligned} S_1 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid s > 0,\ t \geq s \exp\left(\frac{r}{s}\right) \right\}, \\ S_2 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid s = 0,\ t \geq 0,\ r \leq 0 \right\} \end{aligned} \tag{12.11}$$

- `EXPCONE_DUAL` : Dual exponential cone

$$\mathrm{cl}(S_1) = S_1 \cup S_2 \tag{12.12}$$

$$\begin{aligned} S_1 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid r < 0,\ t \geq -r \exp\left(\frac{s}{r} - 1\right) \right\}, \\ S_2 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid r = 0,\ t \geq 0,\ s \geq 0 \right\} \end{aligned} \tag{12.13}$$

### 12.3.2 Exponential Cone Example

Take $(u_1, 1, u_3) \in K_{\exp}$ for example, which mathematical form is $u_1 \geq e^{u_3}$. The code implementation is as follows:

**C Interface:**

Please refer to the example code in the installation package `expcone_gp.c`.

**Python Interface:**

```
u1 = model.addVar(lb=-COPT.INFINITY)
u2 = model.addVar(lb=1.0, ub=1.0)
u3 = model.addVar(lb=-COPT.INFINITY)
model.addExpCone([u1, u2, u3], COPT.EXPCONE_PRIMAL)
```

**C++ Interface:**

```
Var u1 = model.AddVar(-COPT_INFINITY, +COPT_INFINITY, 0.0, COPT_CONTINUOUS, "u1");
Var u2 = model.AddVar(1.0, 1.0, 0.0, COPT_CONTINUOUS, "u2");
Var u3 = model.AddVar(-COPT_INFINITY, +COPT_INFINITY, 0.0, COPT_CONTINUOUS, "u3");

VarArray uconevars;
uconevars.PushBack(u1);
uconevars.PushBack(u2);
uconevars.PushBack(u3);
model.AddExpCone(uconevars, COPT_EXPCONE_PRIMAL);
```

**C# Interface:**

```
VarArray uconevars = new VarArray();
uconevars.PushBack(u1);
uconevars.PushBack(u2);
uconevars.PushBack(u3);
model.AddExpCone(uconevars, Copt.Consts.EXPCONE_PRIMAL);

Var u1 = model.AddVar(-Copt.Consts.INFINITY, +Copt.Consts.INFINITY, 0.0, Copt.Consts.
↪CONTINUOUS, "u1");
Var u2 = model.AddVar(1.0, 1.0, 0.0, Copt.Consts.CONTINUOUS, "u2");
Var u3 = model.AddVar(-Copt.Consts.INFINITY, +Copt.Consts.INFINITY, 0.0, Copt.Consts.
↪CONTINUOUS, "u3");
```

**Java Interface:**

```
Var u1 = model.addVar(-copt.Consts.INFINITY, +copt.Consts.INFINITY, 0.0, copt.Consts.
↪CONTINUOUS, "u1");
Var u2 = model.addVar(1.0, 1.0, 0.0, copt.Consts.CONTINUOUS, "u2");
Var u3 = model.addVar(-copt.Consts.INFINITY, +copt.Consts.INFINITY, 0.0, copt.Consts.
↪CONTINUOUS, "u3");

VarArray uconevars = new VarArray();
uconevars.pushBack(u1);
uconevars.pushBack(u2);
uconevars.pushBack(u3);
model.addExpCone(uconevars, copt.Consts.EXPCONE_PRIMAL);
```

## 12.4 Semidefinite Programming (SDP)

Semidefinite Programming (SDP) consists semidefinite variables and cone constraints.

### 12.4.1 Mathematical Model

**Positive Semidefinite Cone:**

Let $\mathcal{S}^n$ denote the set of $n$ -dimensional symmetric matrices, the positive semidefinite cone is defined as:

$$\mathcal{S}_+^n = \left\{X \in \mathcal{S}^n \mid u^T X u \geq 0, \forall u \in \mathbb{R}^n\right\} \tag{12.14}$$

**Positive Semidefinite (PSD) Variable:**

In an SDP model, the decision variable $X \in \mathcal{S}_+^n$, can also be denoted as $X \succeq 0$. This decision variable is referred to as a semidefinite variable. Besides, the linear constraints of the model include semidefinite variables.

**Semidefinite Cone Constraint**

In an SDP model, $X \succeq 0$ is commonly referred to as the **semidefinite cone constraint**, while a constraint of the form $A \bullet X = b$, which involves a semidefinite variable, is referred to as a linear constraint consisting of PSD variables. For simplicity, in the following chapter, we will refer to it as the **semidefinite constraint** and distinguish it from the semidefinite cone constraint (i.e., $X \succeq 0$).

The mathematical form of semidefinite programming is as follows:

$$\begin{aligned} \min \quad & \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} C_j \bullet X_j + c^f \\ \text{s.t.} \quad & l_i^c \leq \sum_{j=0}^{n-1} a_{ij} x_j + \sum_{j=0}^{p-1} A_j \bullet X_j \leq u_i^c, \qquad i = 0, 1, ..., m-1 \\ & l_j^v \leq \qquad\qquad x_j \qquad\qquad \leq u_j^v, \qquad j = 0, 1, ..., n-1 \\ & \qquad\qquad\qquad X_j \succeq 0, \qquad j = 0, 1, ..., p-1 \end{aligned} \tag{12.15}$$

Here, the operator $\bullet$ denotes the matrix inner product operation:

Given two matrices $A = \{a_{ij}\} \in \mathbb{R}^{m \times n}$ and $B = \{b_{ij}\} \in \mathbb{R}^{m \times n}$, the inner product of matrix $A$ and matrix $B$ is defined as:

$$A \bullet B = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_{ij} b_{ij} \tag{12.16}$$

The variables and arguments in the model have the following meanings:

- Problem size: $m$ denotes the number of constraints, $n$ denotes the number of non-semidefinite variables, and $p$ denotes the number of semidefinite variables.
- Decision variables: non-semidefinite variables $x = (x_j)_{j=0}^{n-1} \in \mathbb{R}^n$, semidefinite variables $X_j \in \mathcal{S}_+^{r_j}$ (for $j = 0, ..., p-1$).
- Non-semidefinite variable range: $l^v, u^v \in \mathbb{R}^n$, where $l^v$ denotes the lower bounds and $u^v$ denotes the upper bounds of the non-semidefinite variables.
- Constraint boundaries: $l^c, u^c \in \mathbb{R}^m$, where $l^c$ denotes the lower bounds and $u^c$ denotes the upper bounds of the constraints.
- Coefficient matrix of the linear constraints: $A = (a_{ij})_{m \times n} \in \mathbb{R}^{m \times n}$, $A_j \in \mathbb{R}^{r_j \times r_j}$.
- Coefficients in the objective function: $c \in \mathbb{R}^n$ represents the coefficients of the non-semidefinite variables, $C_j \in \mathbb{R}^{r_j \times r_j}$ represents the coefficients of the semidefinite variables, and $c^f$ represents the constant term in the objective function.

### 12.4.2 Modeling

The basic steps to construct and solve a semidefinite programming (SDP) model in COPT are as follows:

1. Create the COPT environment and model.
2. Add required data.
3. Construct the semidefinite programming model:
   - Add decision variables (semidefinite and non-semidefinite variables).
   - Add constraints (including semidefinite constraints).
   - Set the objective function.
4. Set optimization parameters and solve.
5. Retrieve the solution.

**Modeling: Adding Semidefinite Variables and Semidefinite Constraints**

In COPT, semidefinite variables are added by specifying the dimension ( `dim` ) of the semidefinite variables:

1. Add a single semidefinite variable.
2. Add multiple semidefinite variables.

COPT provides two ways to add semidefinite constraints:

1. First construct the semidefinite constraint expression by combining the semidefinite terms (semidefinite variables and their corresponding coefficient matrices) and the linear terms, and then add them to the model.
2. Directly provide the semidefinite constraint expression (or semidefinite constraint builder) as an argument input, and add it to the model.

When adding semidefinite constraints to the model, the main arguments that can be specified are:

- `expr` / `builder` : The semidefinite constraint expression or semidefinite constraint builder.
- `rhs` : The right-hand side of the semidefinite constraint.
- `sense` : The constraint type. For possible values, refer to *Constants Chapter: Constraint Type*.
- `name` : The name of the semidefinite constraint.

The implementation in different programming interfaces is shown in Table 12.7:

Table 12.7: Functions for Adding Semidefinite Variables/Constraints

| API | Add Semidefinite Variable | Add Semidefinite Constraint |
|---|---|---|
| C | `COPT_AddPSDCol` / `COPT_AddPSDCols` | `COPT_AddPSDConstr` |
| C++ | `Model::AddPsdVar()` / `Model.AddPsdVars()` | `Model::AddPsdConstr()` |
| C# | `Model.AddPsdVar()` / `Model.AddPsdVars()` | `Model.AddPsdConstr()` |
| Java | `Model.addPsdVar()` / `Model.addPsdVars()` | `Model.addPsdConstr()` |
| Python | `Model.addPsdVar()` / `Model.addPsdVars()` | `Model.addPsdConstr()` |

**Notes**

- Regarding the operations for modeling semidefinite variables and semidefinite constraints, the function names, calling methods, and **argument names differ slightly** among different programming interfaces, but the functionality and argument meanings are consistent.
- In the C API, when adding semidefinite constraints, arguments such as the indices of the nonzero linear term coefficients and the semidefinite variable indices need to be provided. For more details, refer to function `COPT_AddPSDConstr` .
- In the Python API, `Model.addConstr()` can be used to add a linear constraint, a semidefinite constraint, or an indicator constraint to the model. For more details, please refer to *Python API Functions: Model Class*.

**Examples of PSD variables and PSD constraints**

$$\begin{aligned} &A \bullet X + x_1 + x_2 = 0.6 \\ &x_1 \geq 0, x_2 \geq 0, X \succeq 0 \end{aligned} \tag{12.17}$$

where

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{12.18}$$

The code implementation in different programming interfaces is as follows:

C API:

```
{
  /* Matrix A */
  int ndim = 3;
  int nelem = 6;
  int rows[] = {0, 1, 2, 1, 2, 2};
  int cols[] = {0, 0, 0, 1, 1, 2};
  double elems[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
  retcode = COPT_AddSymMat(prob, ndim, nelem, rows, cols, elems);
  if (retcode) goto exit_cleanup;
}
/* Add PSD columns */
int nPSDCol = 1;
int colDims[] = {3};
retcode = COPT_AddPSDCols(prob, nPSDCol, colDims, NULL);
if (retcode) goto exit_cleanup;
/* Add columns */
int nCol = 2;
retcode = COPT_AddCols(prob, nCol, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL);
if (retcode) goto exit_cleanup;
/* Add PSD constraints */
{
  int nRowMatCnt = 2;
  int rowMatIdx[] = {0, 1};
  double rowMatElem[] = {1.0, 1.0};
  int nColCnt = 1;
  int psdColIdx[] = {0};
  int symMatIdx[] = {2};
  char cRowSense = COPT_EQUAL;
  double dRowBound = 0.6;
  retcode = COPT_AddPSDConstr(prob, nRowMatCnt, rowMatIdx, rowMatElem, nColCnt,
    psdColIdx, symMatIdx, cRowSense, dRowBound, 0.0, NULL);
  if (retcode) goto exit_cleanup;
}
```

C++ API:

```
SymMatrix A = model.AddOnesMat(3);
Var x1 = model.AddVar(0.0, COPT_INFINITY, 0, COPT_CONTINUOUS, "x1");
Var x2 = model.AddVar(0.0, COPT_INFINITY, 0, COPT_CONTINUOUS, "x2");
PsdVar barX = model.AddPsdVar(3, "X");
model.AddPsdConstr(A * barX + x1 + x2 == 0.6, "PSD_R");
```

C# API:

```
SymMatrix A = model.AddOnesMat(3);
Var x1 = model.AddVar(0.0, Copt.Consts.INFINITY, 0, Copt.Consts.CONTINUOUS, "x1");
Var x2 = model.AddVar(0.0, Copt.Consts.INFINITY, 0, Copt.Consts.CONTINUOUS, "x2");
PsdVar barX = model.AddPsdVar(3, "X");
model.AddPsdConstr(A * barX + x1 + x2 == 0.6, "PSD_R");
```

Java API:

```
SymMatrix A = model.addOnesMat(3);
Var x1 = model.addVar(0.0, copt.Consts.INFINITY, 0, copt.Consts.CONTINUOUS, "x1");
Var x2 = model.addVar(0.0, copt.Consts.INFINITY, 0, copt.Consts.CONTINUOUS, "x2");
PsdVar barX = model.addPsdVar(3, "X");
```

(continues on next page)

(continued from previous page)

```
PsdExpr pexpr = new PsdExpr(x1, 1.0);
pexpr.addTerm(x2, 1.0);
pexpr.addTerm(barX, A);
model.addPsdConstr(pexpr, copt.Consts.EQUAL, 0.6, "PSD_R");
```

Python API:

```
A = m.addOnesMat(3)
x1 = m.addVar(lb=0.0, ub=COPT.INFINITY, name="x1")
x2 = m.addVar(lb=0.0, ub=COPT.INFINITY, name="x2")
X = m.addPsdVars(3, "BAR_X")
psdc = model.addConstr(A * X + x1 + x2 == 0.6, name="PSD_C")
```

In the programming interfaces provided by COPT, except for the C language, other object-oriented programming interfaces (C#, C++, Java, Python) offer classes related to semidefinite constraints:

- Encapsulation of operations for constructing semidefinite expressions
  1. `PsdExpr` Class: Encapsulation of operations related to the combination of semidefinite variables when constructing semidefinite expressions in COPT.
- Encapsulation of operations related to semidefinite constraints
  1. `PsdConstraint` Class: Encapsulation of operations related to semidefinite constraints in COPT.
  2. `PsdConstrArray` Class: Facilitates user operations on a group of `PsdConstraint` class objects.
- Encapsulation of semidefinite constraint builders
  1. `PsdConstrBuilder` Class: Encapsulation for constructing semidefinite constraint builders in COPT.
  2. `PsdConstrBuilderArray` Class: Facilitates user operations on a group of `PsdConstrBuilder` class objects.
- C++ API: *PsdExpr Class* , *PsdConstrArray Class* , *PsdConstrBuilder Class* , *PsdConstrBuilderArray Class*
- C# API: *PsdExpr Class* , *PsdConstraint Class* , *PsdConstrArray Class* , *PsdConstrBuilder Class* , *PsdConstrBuilderArray Class*
- Java API: *PsdExpr Class* , *PsdConstraint Class* , *PsdConstrArray Class* , *PsdConstrBuilder Class* , *PsdConstrBuilderArray Class*
- Python API: *PsdExpr Class* , *PsdConstraint Class* , *PsdConstrArray Class* , *PsdConstrBuilder Class* , *PsdConstrBuilderArray Class*

### 12.4.3 Solving

For SDP problems, COPT provides the Barrier method and the ADMM method. You can specify the solving method by setting the optimization parameter `SDPMethod`.

For more details, please refer to *Parameter Section: Semidefinite Programming Related*.

### 12.4.4 Related Attributes

Attributes related to SDP model are as shown in Table 12.8:

Table 12.8: Attributes for Semidefinite Variables and Constraints

| Name | Type | Description |
|---|---|---|
| PSDCols | Integer | Number of PSD variables in the problem |
| PSDElems | Integer | Number of PSD terms in objective function |
| SymMats | Integer | Number of symmetric matrices in the problem |
| PSDConstrs | Integer | Number of PSD constraints |
| HasPSDObj | Integer | Whether the problem has PSD terms in objective function |

## 12.5 Quadratic Programming (QP)

### 12.5.1 Mathematical Formulation

Convex Quadratic Programming (QP) has a convex quadratic objective function and linear constraints. The mathematical formulation is as follows:

$$\begin{aligned} \min \quad & x^T Q x + c^T x + c^f \\ \text{s.t.} \quad & l_i^c \le \sum_{j=0}^{n-1} a_{ij} x_j \le u_i^c, \qquad i = 0, \cdots, m-1 \\ & l_j^v \le x_j \le u_j^v, \qquad j = 0, 1, \cdots, n-1 \end{aligned} \tag{12.19}$$

The variables and arguments in the model are defined as follows:

- Decision variables: $x = (x_j)_{j=0}^{n-1} \in \mathbb{R}^n$;
- Variable bounds: $l^v, u^v \in \mathbb{R}^n$; where $l^v$ denotes the lower bounds and $u^v$ denotes the upper bounds of the variables;
- Constraint bounds: $l^c, u^c \in \mathbb{R}^m$; where $l^c$ denotes the lower bounds and $u^c$ denotes the upper bounds of the constraints;
- Coefficient matrix of linear constraints: $A = (a_{ij})_{m \times n} \in \mathbb{R}^{m \times n}$
- Coefficients in the objective function:
  - $Q \in \mathbb{R}^{n \times n}$ represents the coefficients of the quadratic term in the objective function
  - $c \in \mathbb{R}^n$ represents the coefficients of the linear term in the objective function, and $c^f$ represents the constant term in the objective function
- Problem size: $m$ denotes the number of constraints, and $n$ denotes the number of decision variables.

## 12.6 Quadratically Constrained Programming (QCP)

Convex Quadratically Constrained Programming (QCP) consists of convex quadratic constraints.

### 12.6.1 Mathematical Model

The mathematical formulation is as follows:

$$\begin{aligned} \min \quad & \frac{1}{2}x^T Q x + c^T x + c^f \\ \text{s.t.} \quad & \frac{1}{2}x^T P_i x + a_i^T x_j \leq u_i^c, \qquad i = 0, \cdots, m-1 \\ & l^v \leq x_j \leq u^v, \qquad j = 0, 1, \cdots, n-1 \end{aligned} \tag{12.20}$$

The variables and parameters in the model are defined as follows:

- Decision variables: $x = (x_j)_{j=0}^{n-1} \in \mathbb{R}^n$;
- Variable bounds: $l^v, u^v \in \mathbb{R}^n$, where $l^v$ denotes the lower bounds and $u^v$ denotes the upper bounds of the variables;
- Constraint bounds: $u^c \in \mathbb{R}^m$, where $u^c$ denotes the upper bounds of the constraints;
- Coefficients in the quadratic constraints:
  - $P_i \in \mathbb{R}^{n \times n}$ (for $i = 0, ..., m-1$),
  - $a_i = (a_{ij})_n \in \mathbb{R}^n$;
- Coefficients in the objective function:
  - $Q \in \mathbb{R}^{n \times n}$ represents the coefficients of the quadratic term in the objective function;
  - $c \in \mathbb{R}^n$ represents the coefficients of the linear term in the objective function, and $c^f$ represents the constant term in the objective function.
- Problem size: $m$ denotes the number of constraints, and $n$ denotes the number of decision variables.

**Notes**

- The COPT solver currently supports solving convex quadratic programming and convex quadratically constrained programming problems. The matrices $Q$ and $P_i$ (for $i = 0, 1, \cdots, n$) must be **positive semidefinite**.
- When the constraint type is >= (of the form $\frac{1}{2}x^T P_i x + \sum_{j=0}^{n-1} a_{ij} x_j \geq l_i^c$), the matrix $Q$ must be **negative semidefinite**.
- As seen from the mathematical formulation above, the quadratic constraint expression includes quadratic terms, linear terms, and constant terms.

### 12.6.2 Modeling

The basic steps to construct and solve a Quadratically Constrained Programming (QCP) model in COPT are as follows:

1. Create a COPT environment and model.
2. Add model parameters.
3. Construct the QCP model:
   - Add variables.
   - Add quadratic constraints.
   - Set the quadratic objective function.
4. Set solving parameters and solve.
5. Retrieve the solution.

### Modeling: Adding Quadratic Constraints

COPT provides two ways to add quadratic constraints:

1. First, construct the quadratic constraint expression by combining the quadratic and linear terms, then add it to the model.
2. Directly provide the quadratic constraint expression (or a quadratic constraint builder) as a parameter input and add it to the model.

When adding quadratic constraints to the model, the following parameters can be specified by the user:

- `expr` / `builder`: The quadratic constraint expression or quadratic constraint builder.
- `rhs`: The right-hand side of the quadratic constraint.
- `sense`: The type of constraint, which can be `COPT_LESS_EQUAL` or `COPT_GREATER_EQUAL`.
- `name`: The name of the quadratic constraint.

The implementation in different programming interfaces is shown in Table 12.9:

Table 12.9: Functions for Adding Quadratic Constraints

| Programming Interface | Function |
|---|---|
| C | `COPT_AddQConstr` |
| C++ | `Model::AddQConstr()` |
| C# | `Model.AddQConstr()` |
| Java | `Model.addQConstr()` |
| Python | `Model.addQConstr()` |

**Note**

- The operations related to quadratic constraint modeling may vary slightly in terms of function names, calling methods, and **parameter names** in different programming interfaces, but the functionality and parameter meanings are consistent.
- In the C API, when adding a quadratic constraint, non-zero linear term coefficient indices, quadratic term indices, and other parameters must be provided. For more details, please refer to *C API Functions: Constructing and Modifying the Model* under the function `COPT_AddQConstr`.
- In the Python API, `Model.addQConstr()` can be used to add a quadratic constraint to the model. For more details, please refer to *Python API Functions: Model Class*.

### Examples of quadratic constraints

$$x_1^2 + x_2^2 + x_1 + 2x_2 <= 0 \tag{12.21}$$

The code implementation in different programming interfaces is as follows:

C API:

```
int nRowMatCnt = 2;
int rowMatIdx[] = {0, 1};
double rowMatElem[] = {1.0, 2.0};

int nQMatCnt = 2;
int qMatRow[] = {0, 1};
int qMatCol[] = {0, 1};
double qMatElem[] = {1.0, 1.0};
char cRowSense = COPT_LESS_EQUAL;
double dRowBound = 0.0;
```

(continues on next page)

(continued from previous page)

```
char *name = "q1";
errcode = COPT_AddQConstr(prob, nRowMatCnt, rowMatIdx, rowMatElem,
                          nQMatCnt, qMatRow, qMatCol, qMatElem,
                          cRowSense, dRowBound, name);
```

C++ API:

```
model.AddQConstr(x1*x1 + x2*x2 + x1 + 2*x2 <= 0, "q1");
```

C# API:

```
model.AddQConstr(x1*x1 + x2*x2 + x1 + 2*x2 <= 0, "q1");
```

Java API:

```
QuadExpr q1 = new QuadExpr(0.0);
q1.addTerm(x1, x1, 1);
q1.addTerm(x2, x2, 1);
q1.addTerm(x1, 1);
q1.addTerm(x2, 2);
model.addQConstr(q1, copt.Consts.LESS_EQUAL, 0, "q1");
```

Python API:

```
model.addQConstr(x1*x1 + x2*x2 + x1 + 2*x2 <= 0, name="q1")
```

In the programming interfaces provided by COPT, except for C language, other object-oriented programming interfaces (C#, C++, Java, Python) offer classes related to quadratic constraints:

- Encapsulation of operations for constructing quadratic expressions
  1. `QuadExpr` Class: Encapsulation of operations related to combining variables when constructing quadratic expressions in the COPT.
- Encapsulation of operations related to quadratic constraints
  1. `QConstraint` Class: Encapsulation of operations related to quadratic constraints in the COPT.
  2. `QConstrArray` Class: Facilitates user operations on a group of `QConstraint` class objects.
- Encapsulation of quadratic constraint builders
  1. `QConstrBuilder` Class: Encapsulation of builders for constructing quadratic constraints in the COPT.
  2. `QConstrBuilderArray` Class: Facilitates user operations on a group of `QConstrBuilder` class objects.
- C++ API: *QuadExpr Class* , *QConstraint Class* , *QConstrArray Class* , *QConstrBuilder Class* , *QConstrBuilderArray Class*
- C# API: *QuadExpr Class* , *QConstraint Class* , *QConstrArray Class* , *QConstrBuilder Class* , *QConstrBuilderArray Class*
- Java API: *QuadExpr Class* , *QConstraint Class* , *QConstrArray Class* , *QConstrBuilder Class* , *QConstrBuilderArray Class*
- Python API: *QuadExpr Class* , *QConstraint Class* , *QConstrArray Class* , *QConstrBuilder Class* , *QConstrBuilderArray Class*

### 12.6.3 Related Attributes

Attributes for Quadratic Programming (QP) and Quadratically Constrained Programming (QCP) models are shown in Table 12.10:

Table 12.10: Attributes for QP and QCP

| Name | Type | Description |
|---|---|---|
| `QConstrs` | Integer | Number of quadratic constraints |
| `QElems` | Integer | Number of non-zero quadratic elements in the quadratic objective function |
| `HasQObj` | Integer | Whether the problem has quadratic objective function |

## 12.7 General Nonlinear Programming (NLP)

Nonlinear Programming (NLP) refers to optimization problems in which the objective function or constraint functions contain nonlinear expressions. The Cardinal Optimizer (COPT) provides support for solving a wide range of nonlinear programming problems and is suitable for applications such as engineering optimization, numerical computation, and complex system modeling.

Currently, COPT supports nonlinear programming problems through the following approaches:

- For nonconvex quadratic programming (Nonconvex QP) and nonconvex quadratically constrained programming (Nonconvex QCQP), the problems can be solved by setting the optimization parameter `"NonConvex"`;
- For more general nonlinear programming problems, COPT supports solving the model either by explicit nonlinear expression modeling or by using a callback-based nonlinear modeling approach;
- In addition, models in `NL` format can be solved by reading the model files via the `coptampl` tool.

The following sections introduce the two main nonlinear modeling approaches supported by COPT.

### 12.7.1 Explicit Expression Nonlinear Modeling

In the explicit expression modeling approach, users construct nonlinear operators and function expressions to directly describe the mathematical form of the objective function and constraint functions. This approach is suitable for nonlinear problems with clear structures that can be represented in closed-form analytic expressions.

COPT supports the following types of nonlinear expressions:

- Basic arithmetic operations (add, minus, mult, div)
- Trigonometric functions (sin, cos, tan, etc.)
- Exponential functions (exp), power functions (pow), logarithmic functions (log), and root functions (sqrt)

In addition, COPT also supports commonly used nonlinear functions such as ceiling (ceil), floor (floor), and absolute value (abs).

For the C interface of COPT, the nonlinear expression operators are described in *Nonlinear Expression Operator Constants*;

For object-oriented interfaces, nonlinear expression operations are encapsulated in the `NL` namespace:

- C++ API: *NL Namespace*
- C# API: *NL Namespace*
- Java API: *NL Namespace*

- Python API: *nl Namespace*

The following example illustrates a nonlinear programming model that includes nonlinear expressions in both the objective function and the constraint functions.

**Objective Function**

$$\min \quad x_1 x_4 \left( \sin(x_1 + x_2) + \cos(x_2 x_3) + \tan\left(\frac{x_3}{x_4}\right)\right) + x_3 \tag{12.22}$$

**Constraints**

$$\begin{aligned} x_1 x_2 x_3 x_4 + x_1 + x_2 &\geq 35 \\ \log(x_1) + 2\log(x_2) + 3\log(x_3) + 4\log(x_4) + x_3 + x_4 &\geq 15 \\ x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_1 + x_3 &\geq 50 \end{aligned} \tag{12.23}$$

**Variable Bounds**

$$1 \leq x_1, x_2, x_3, x_4 \leq 5 \tag{12.24}$$

The complete implementation of the above model can be found in the example `nlp_ex1` under the `examples` directory of the COPT installation package.

### 12.7.2 Callback-Based Nonlinear Modeling

In addition to expression-based nonlinear modeling, COPT also provides a callback-based nonlinear modeling mechanism, which is designed for nonlinear optimization problems that are difficult to describe using explicit symbolic expressions.

Under this mode, users do not need to explicitly construct analytical expressions for the objective function and constraints. Instead, by implementing a nonlinear callback class, users provide numerical values of the objective function, constraint functions, and their first-order and second-order derivatives to the solver at given variable points.

The structural information of the nonlinear model (such as the number of variables, the number of constraints, and the sparsity patterns of derivatives) is loaded once via the `Model.loadNlData` interface. The numerical computation logic is encapsulated in the callback object and invoked by the solver during the solution process. During the iterations of the nonlinear optimization algorithm, COPT automatically calls the corresponding callback methods at different variable points to obtain the required numerical information and advance the solution process.

Currently, the callback-based nonlinear modeling mechanism is supported in the Python and C++ interfaces of COPT:

- C++ API: *NlpCallbackBase class*
- Python API: *NlpCallbackBase class*

Taking object-oriented programming interfaces as an example, the callback-based nonlinear modeling workflow typically consists of the following steps:

1. Define a custom nonlinear callback class and inherit from `NlpCallbackBase`.

   The callback class is responsible for implementing the numerical computation interfaces of the nonlinear model, including the objective function, the constraint functions, and their first- and second-order derivatives.

2. Implement the required numerical evaluation methods in the callback class.

   The computation rules of the objective function, constraint functions, and their first- and second-order derivatives are provided through the callback interfaces. Users may implement some or all of the following methods to define the numerical computation logic of the nonlinear model:

   - `NlpCallbackBase.EvalObj`: compute the objective function value;

- `NlpCallbackBase.EvalCon`: compute the constraint function values;
- `NlpCallbackBase.EvalGrad`: compute the first-order derivative (gradient) of the objective function;
- `NlpCallbackBase.EvalJac`: compute the first-order derivatives (Jacobian matrix) of the constraint functions;
- `NlpCallbackBase.EvalHess`: compute the second-order derivatives (Hessian matrix) of the Lagrangian function.

Each method takes the current variable values as input and provides the corresponding numerical results through the designated output arrays.

3. Create an instance of the custom nonlinear callback class.

   This instance encapsulates the numerical computation logic of the nonlinear model and is scheduled uniformly by the solver during the nonlinear solution process.

4. Register the nonlinear model information and the callback object by calling the `Model.loadNlData` interface.

   In this step, users provide the solver with the model scale, variable and constraint bounds, and sparsity structures of derivatives, and pass the custom callback instance as an argument.

   Meanwhile, the `evalType` parameter can be used to declare the types of numerical information that the callback object is able to provide, allowing the solver to invoke the corresponding callback methods as required during the iterations of the nonlinear optimization algorithm.

After completing the above steps, the nonlinear optimization process can be started by calling `Model.solve`. During the solution process, the solver automatically invokes the relevant callback methods at different iteration points to obtain the required numerical information and advance the nonlinear optimization algorithm.

## 12.8 Mixed-Integer Programming (MIP)

### 12.8.1 Modeling

Mixed-Integer Programming (MIP) refers to optimization problems where some of the decision variables are restricted to integer values. Currently, COPT supports integer variables combined with linear programming, second-order cone programming, quadratic programming, and quadratically constrained programming.

| MIP Problem Type | Solving Algorithm |
|---|---|
| MILP | Branch-and-Cut |
| MISOCP | Branch-and-Cut |
| MIQP | Branch-and-Cut |
| MIQCP | Branch-and-Cut |

In COPT, the supported integer variable types and their corresponding constants are as follows:

- `BINARY`

  Binary variable

- `INTEGER`

  Integer variable

When adding decision variables to the model:

- Specify the parameter `vtype` as `BINARY` to add binary variables;
- Specify the parameter `vtype` as `INTEGER` to add integer variables.

### 12.8.2 Solving

For integer programming problems, COPT provides the Branch-and-Cut algorithm, which can be specified by setting the optimization parameter `"MipMethod"`. By configuring other related optimization parameters, you can control the specific workflow of the Branch-and-Cut algorithm. For more details, please refer to *Parameter Section: Integer Programming Related*.

For the solving logs of integer programming problems, please refer to *Logging Section: Branch-and-Cut*.

### 12.8.3 Related Attributes

Table 12.11: Overview of attributes for MIP

| Name | Type | Description |
|---|---|---|
| `Bins` | Integer | Number of binary variables |
| `Ints` | Integer | Number of integer variables |
| `Indicators` | Integer | Number of indicator constraints |
| `IsMIP` | Integer | Whether the problem is a MIP |
| `NodeCnt` | Integer | Number of explored nodes |
| `HasMipSol` | Integer | Whether MIP solution is available |
| `BestObj` | Double | Best integer objective value for MIP |
| `BestBnd` | Double | Best bound for MIP |
| `BestGap` | Double | Best relative gap for MIP |

## 12.9 Special Constraints

COPT supports the construction of two types of special constraints: SOS constraints and Indicator constraints.

### 12.9.1 SOS Constraints

SOS constraints (Special Ordered Set) are a special type of constraint that restricts the values of a group of variables. Currently, COPT supports two types of SOS constraints:

1. **SOS1 Constraint**: In this type of constraint, at most one variable in the specified group can take a non-zero value.
2. **SOS2 Constraint**: In this type of constraint, at most two variables in the specified group can take non-zero values, and the variables with non-zero values must be adjacent in the order.

These two types of SOS constraints correspond to the following constants in COPT, which can be specified when adding SOS constraints to the model:

- `SOS_TYPE1`

  SOS1 Constraint

- `SOS_TYPE2`

  SOS2 Constraint

When adding SOS constraints to the model, the following arguments can be specified by the user:

- `sostype`: Specifies the type of the SOS constraint.
- `vars`: The variables involved in the SOS constraint.
- `weights`: The weights of the variables involved in the SOS constraint; an optional argument, default is `None`.

**Note**

- The variables involved in the SOS constraint can be continuous variables, binary variables, or integer variables.
- If the model includes SOS constraints, the model is an integer programming model.
- The specific operations and usage of SOS constraints, including the function names, calling methods, and **argument names, may vary slightly** in different programming interfaces, but the functionality and argument meanings are consistent.

COPT provides related functions to support operations on SOS constraints. Below are the corresponding functions for adding and retrieving SOS constraints in different programming interfaces:

Table 12.12: Adding and Retrieving SOS Constraints

| API | Add SOS Constraints | Retrieve All SOS Constraints in the Model |
|---|---|---|
| C | `COPT_AddSOSs` | `COPT_GetSOSs` |
| C++ | `Model::AddSos()` | `Model::GetSoss()` |
| C# | `Model.AddSos()` | `Model.GetSoss()` |
| Java | `Model.addSos()` | `Model.getSoss()` |
| Python | `Model.addSOS()` | `Model.getSOSs()` |

For operations related to SOS constraints, the function names, calling methods, and argument names may vary slightly in different programming interfaces, but the functionality and argument meanings are consistent. Please refer to the corresponding sections of each programming language's API reference manual for specific details:

- C API: *Building and modifying a problem*
- C++ API: *Model Class*
- C# API: *Model Class*
- Java API: *Model Class*
- Python API: *Model Class*

In the programming interfaces provided by COPT, except for the C language, other object-oriented programming interfaces (C#, C++, Java, Python) offer classes related to SOS constraints:

- Encapsulation of operations related to SOS constraints:
  1. `Sos` Class: Encapsulation of operations related to SOS constraints in COPT.
  2. `SosArray` Class: Facilitates user operations on a group of `Sos` class objects.
- Encapsulation of SOS constraint builders:
  1. `SosBuilder` Class: Encapsulation of builders for constructing SOS constraints in COPT, providing the following member methods:
  2. `SosBuilderArray` Class: Facilitates user operations on a group of `SosBuilder` class objects.

For more details on the member methods and specific descriptions of the above SOS constraint-related classes, please refer to the corresponding sections in the API reference manuals of each programming language.

- C++ API: *Sos Class* , *SosArray Class* , *SosBuilder Class* , *SosBuilderArray Class*
- C# API: *Sos Class* , *SosArray Class* , *SosBuilder Class* , *SosBuilderArray Class*
- Java API: *Sos Class* , *SosArray Class* , *SosBuilder Class* , *SosBuilderArray Class*
- Python API: *SOS Class* , *SOSArray Class* , *SOSBuilder Class* , *SOSBuilderArray Class*

### 12.9.2 Indicator Constraints

Indicator constraint is a type of logical constraint that uses a binary variable $y$ as the indicator variable to determine the logical relationship between the value of $y$ and whether the linear constraint $a^T x \leq b$ is satisfied. Currently, COPT supports three types of Indicator constraints: If-Then, Only-If, and If-and-Only-If.

- `INDICATOR_IF`

  If-Then:

  If $y = f$, then the linear constraint is satisfied;

  If $y \neq f$, then the linear constraint can be violated.

$$\begin{aligned} y = f \rightarrow a^T x \leq b \\ f \in \{0, 1\} \end{aligned} \tag{12.25}$$

- `INDICATOR_ONLYIF`

  Only-If:

  If the linear constraint $a^T x \leq b$ is satisfied, then $y = f$;

  If the linear constraint $a^T x \leq b$ is not satisfied, then $y$ can take the value of 0 or 1.

$$\begin{aligned} a^T x \leq b \rightarrow y = f \\ f \in \{0, 1\} \end{aligned} \tag{12.26}$$

- `INDICATOR_IFANDONLYIF`

  If-and-Only-If:

  The linear constraint $a^T x \leq b$ and $y = f$ are equivalent. They must both be satisfied or both be unsatisfied.

$$\begin{aligned} a^T x \leq b \leftrightarrow y = f \\ f \in \{0, 1\} \end{aligned} \tag{12.27}$$

COPT provides two methods to add Indicator constraints:

1. By calling an API function (e.g., in Python: `Model.addGenConstrIndicator()`). The key parameters for constructing an Indicator constraint are:
   - `binVar`: The binary indicator variable.
   - `binval`: The value (`True` or `False`) of the indicator variable that is conditionally related to the satisfaction of the linear constraint.
   - `builder`: The linear constraint builder.
   - `type`: The type of Indicator constraint (possible values are listed in *Indicator constraint types*).
2. By overloading operators (for If-Then and Only-If constraints):
   - `>>`: Represents the If-Then logical relationship, corresponding to `INDICATOR_IF`;
   - `<<`: Represents the Only-If logical relationship, corresponding to `INDICATOR_ONLYIF`.

Here are examples of how to implement these methods in Python API:

1. Adding an If-Then type Indicator constraint where the linear constraint $y + 2z >= 3$ is satisfied if $x$ is true.

$$x = 1 \rightarrow y + 2z \geq 3 \tag{12.28}$$

```
model.addGenConstrIndicator(x, True, y + 2*z >= 3)
```

```
model.addConstr((x==1) >> (y + 2*z >= 3))
```

2. Adding an Only-If type Indicator constraint where $x$ is false if the linear constraint $y + 2z <= 3$ is satisfied.

$$x = 0 \leftarrow y + 2z \leq 3 \tag{12.29}$$

```
model.addGenConstrIndicator(x, False, y + 2*z <= 3, type=COPT.INDICATOR_
↪ONLYIF)
```

```
model.addConstr((x==0) << (y + 2*z <= 3))
```

3. Adding an If-and-Only-If type Indicator constraint where the binary variable $x$ being true is equivalent to the linear constraint $y + 2z = 3$ being satisfied.

$$x = 1 \leftrightarrow y + 2z = 3 \tag{12.30}$$

```
model.addGenConstrIndicator(x, True, y + 2*z == 3, type=COPT.INDICATOR_
↪IFANDONLYIF)
```

**Note**

1. The general expression of the linear constraint given above, $a^T x \leq b$, can actually take the forms $\leq$, $\geq$, or $=$.
2. If the model includes Indicator constraints, it is considered an integer programming model.
3. COPT supports adding a batch of Indicator constraints to the model by calling an API function. In Python, the corresponding function is: `Model.addGenConstrIndicators()`.
4. The method of adding Indicator constraints by overloading operators only supports If-Then and Only-If constraints. If you need to add an If-and-Only-If constraint, you must use the API function and specify the `type` as `INDICATOR_IFANDONLYIF`.
5. The specific operations and usage of Indicator constraints, including function names, calling methods, and **parameter names, may vary slightly** in different programming interfaces, but the functionality and parameter meanings are consistent.

COPT provides related functions to support adding Indicator constraints and retrieving the corresponding Indicator constraint builders, as listed below:

Table 12.13: Adding and Retrieving Indicator Constraints in Different APIs

| API | Add Indicator | Retrieve Indicator |
|---|---|---|
| C | `COPT_AddIndicator` | `COPT_GetIndicator` |
| C++ | `Model::AddGenConstrIndicator()` | `Model::GetGenConstrIndicator()` |
| C# | `Model.AddSos()` | `Model.GetGenConstrIndicator()` |
| Java | `Model.addSos()` | `Model.getGenConstrIndicator()` |
| Python | `Model.addGenConstrIndicator()` | `Model.getGenConstrIndicator()` |

For operations such as building and adding Indicator constraints, the function names, calling methods, and argument names may vary slightly in different programming interfaces, but the functionality and

argument meanings are consistent. Please refer to the corresponding sections of each programming language's API reference manual for specific details:

- C API: *Constructing and Modifying the Model*
- C++ API: *Model Class*
- C# API: *Model Class*
- Java API: *Model Class*
- Python API: *Model Class*

In the programming interfaces supported by COPT, except for the C language, other object-oriented programming interfaces (C#, C++, Java, Python) offer classes related to Indicator constraints:

- Encapsulation of operations related to Indicator constraints:
  1. `GenConstr` Class: Encapsulation of operations related to Indicator constraints in COPT.
  2. `GenConstrArray` Class: Facilitates user operations on a group of `GenConstr` class objects.
- Encapsulation of Indicator constraint builders:
  1. `GenConstrBuilder` Class: Encapsulation of builders for constructing Indicator constraints in COPT.
  2. `GenConstrBuilderArray` Class: Facilitates user operations on a group of `GenConstrBuilder` class objects.

For more details on the member methods and specific descriptions of the above Indicator constraint-related classes, please refer to the corresponding sections in the API reference manuals of each programming language.

- C++ API: *GenConstr Class* , *GenConstrArray Class* , *GenConstrBuilder Class* , *GenConstrBuilderArray Class*
- C# API: *GenConstr Class* , *GenConstrArray Class* , *GenConstrBuilder Class* , *GenConstrBuilderArray Class*
- Java API: *GenConstr Class* , *GenConstrArray Class* , *GenConstrBuilder Class* , *GenConstrBuilderArray Class*
- Python API: *GenConstr Class* , *GenConstrArray Class* , *GenConstrBuilder Class* , *GenConstrBuilderArray Class*

### 12.9.3 Attributes for Special Constraints

COPT provides the following attributes to describe the number of special constraints in the model, as shown in Table 12.14.

For methods of retrieving these attributes in different programming interfaces, please refer to: *Attributes Section*.

Table 12.14: Overview of Special Constraints-Related Attributes

| Attribute | Type | Description |
|---|---|---|
| `Soss` | Integer | Number of SOS constraints |
| `Indicators` | Integer | Number of indicator constraints |
| `IISSOSs` | Integer | Number of SOS constraints in IIS |
| `IISIndicators` | Integer | Number of indicator constraints in IIS |

### 12.9.4 IIS Status of Special Constraints

Regarding the IIS (Irreducible Infeasible Set) calculation results for infeasible models, COPT provides related functions to obtain the IIS status of SOS constraints. Please refer to *Handling Infeasible Models Section: Retrieving IIS Status of Special Constraints*.

# Chapter 13

# Handling Infeasible Models

This chapter introduces two approaches supported by COPT for handling infeasible problems:

- *IIS for Infeasible Models*
- *Feasibility Relaxation*

In real-world problems, it is common to encounter infeasible models, which correspond to the solution status code `COPT.INFEASIBLE`. The main reasons for infeasibility are usually:

1. Making some mistakes when modeling or inputting data (e.g., an empty left-hand side in a constraint).
2. The problem itself is infeasible, meaning some constraints or variable bounds are conflicting.

COPT provides two methods for analyzing and handling infeasible models, which are supported for both Linear Programming (LP) and Mixed-Integer Linear Programming (MILP):

1. **Compute IIS**: Identify the key constraints and variable bounds causing infeasibility.
2. **Feasibility Relaxation (FeasRelax)**: Quantitatively compute the conflicts in constraints or variable bounds (violations) that lead to infeasibility.

> **Note**
>
> When using the barrier algorithm, if the log contains `Problem looks infeasible` or `Problem looks unbounded`, the message indicates that the standard barrier algorithm has made a preliminary assessment of the model status based on the behavior of the current iterates. For the meaning of this log message, please see the relevant discussion in *Barrier Logging*.

## 13.1 IIS for Infeasible Models

IIS (Irreducible Inconsistent Subsystem) refers to a minimal conflicting set in the model that causes infeasibility, and has the following properties:

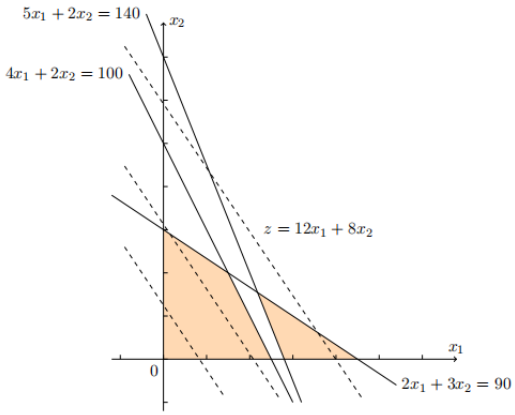
1. The subsystem is still infeasible.
2. Removing any single constraint or variable bound from the IIS will make the subsystem feasible.

**Note:** The IIS computed by COPT may not be minimal or unique. It may require several iterations of modifying constraints and recomputing the IIS before the model becomes feasible.

Below is an example of an infeasible Linear Programming model:

$$\begin{aligned} \max \quad & z = 12x_1 + 8x_2 \\ \text{s.t.} \quad & 5x_1 + 2x_2 \geq 140 \\ & 2x_1 + 3x_2 \leq 90 \\ & 4x_1 + 2x_2 \leq 100 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{13.1}$$

The regions defined by the constraints are shown below. As illustrated, these regions have no common intersection, and therefore the model is infeasible:




From the figure, the conflicting constraints can be clearly seen:

$$\begin{aligned} & c1 : 5x_1 + 2x_2 \geq 140 \\ & c3 : 4x_1 + 2x_2 \leq 100 \end{aligned} \tag{13.2}$$

After computing the IIS for the above model and writing it to a file (`.iis` format), the file content is as follows, consistent with the figure:

```
\Generated by Cardinal Operations

Maximize
   12 x[1] + 8 x[2]
Subject To
 c1: 5 x[1] + 2 x[2] >= 140
 c3: 4 x[1] + 2 x[2] <= 100
END
```

### 13.1.1 Computing IIS

COPT provides functions in various programming APIs to compute the IIS of an infeasible model, returning a set of conflicting constraints and variable bounds. The IIS can also be written to a file by specifying the file name suffix as `.iis` (e.g., `example.iis`). Related function names are shown in Table 13.1:

Table 13.1: Functions for Computing IIS of Infeasible Models

| API | Compute IIS | Write IIS to File |
|---|---|---|
| C | `COPT_ComputeIIS` | `COPT_WriteIIS` |
| C++ | `Model::ComputeIIS()` | `Model::WriteIIS()` |
| C# | `Model.ComputeIIS()` | `Model.WriteIIS()` |
| Java | `Model.computeIIS()` | `Model.writeIIS()` |
| Python | `Model.computeIIS()` | `Model.writeIIS()` |

### 13.1.2 Getting IIS status of variables and constraints

#### Functions for variable's IIS status

After computing the IIS, the IIS status of variables (lower/upper bound) can be obtained. The status indicates whether the bound is in the IIS:

- 1: The specified variable bound (lower/upper) is in the IIS
- 0: The specified variable bound (lower/upper) is not in the IIS

Supported functions for different interfaces are shown in Table 13.2:

Table 13.2: Functions for Getting Variable IIS Status

| API | Lower Bound IIS Status | Upper Bound IIS Status |
|---|---|---|
| C | `COPT_GetColLowerIIS` | `COPT_GetColUpperIIS` |
| C++ | `Model::GetVarLowerIIS()` | `Model::GetVarUpperIIS()` |
| C# | `Model.GetVarLowerIIS()` | `Model.GetVarUpperIIS()` |
| Java | `Model.getVarLowerIIS()` | `Model.getVarUpperIIS()` |
| Python | `Model.getVarLowerIIS()` | `Model.getVarUpperIIS()` |

**Notes**

- The above functions accept either a single variable (`Var` object) or a set of variables (`VarArray` or `tupledict` objects).
- Except for the C API, other interfaces provide member functions in the `Var` class to get the IIS status for a single variable. For example, in Python: `Var.getLowerIIS()` and `Var.getUpperIIS()`.
- In the Python API, you can also directly access the IIS status as member attributes of the `Var` object: `Var.iislb` for the lower bound IIS status, `Var.iisub` for the upper bound IIS status.

#### Functions for constraint's IIS status

After computing the IIS, COPT also supports querying the IIS status of constraint bounds (lower/upper). The status indicates whether the bound is in the IIS:

- 1: The specified constraint bound (lower/upper) is in the IIS
- 0: The specified constraint bound (lower/upper) is not in the IIS

Supported functions for different interfaces are shown in Table 13.3:

Table 13.3: Functions for Getting Constraint IIS Status

| API | Lower Bound IIS Status | Upper Bound IIS Status |
|---|---|---|
| C | `COPT_GetRowLowerIIS` | `COPT_GetRowUpperIIS` |
| C++ | `Model::GetConstrLowerIIS()` | `Model::GetConstrUpperIIS()` |
| C# | `Model.GetConstrLowerIIS()` | `Model.GetConstrUpperIIS()` |
| Java | `Model.getConstrLowerIIS()` | `Model.getConstrUpperIIS()` |
| Python | `Model.getConstrLowerIIS()` | `Model.getConstrUpperIIS()` |

**Notes**

- The above functions accept either a single constraint (`Constraint` object) or a set of constraints (`Constraint` or `ConstrArray` objects).
- Except for the C API, other interfaces provide member functions in the `Constraint` class to get the IIS status for a single constraint. For example, in Python: `Constraint.getLowerIIS()` and `Constraint.getUpperIIS()`.
- In the Python API, you can also directly access the IIS status as member attributes of the `Constraint` object: `Constraint.iislb` for the lower bound IIS status, `Constraint.iisub` for the upper bound IIS status.

### Functions for special constraints

COPT also provides functions, as shown in Table 13.4, for getting the IIS status of SOS and Indicator constraints:

Table 13.4: Functions for Getting IIS Status of Special Constraints

| API | SOS Constraint | Indicator Constraint |
|---|---|---|
| C | `COPT_GetSOSIIS` | `COPT_GetIndicatorIIS` |
| C++ | `Model::GetSOSIIS()` | `Model::GetIndicatorIIS()` |
| C# | `Model.GetSOSIIS()` | `Model.GetIndicatorIIS()` |
| Java | `Model.getSOSIIS()` | `Model.getIndicatorIIS()` |
| Python | `Model.getSOSIIS()` | `Model.getIndicatorIIS()` |

For details on the usage of the above functions, please refer to the API reference manual for each programming interface:

- C API: *IIS computation functions*
- C++ API: *Model class*, *Var class*, *Constraint class*
- C# API: *Model class*, *Var class*, *Constraint class*
- Java API: *Model class*, *Var class*, *Constraint class*
- Python API: *Model class*, *Var class*, *Constraint class*

### 13.1.3 IIS-related parameters, attributes and information

#### Parameters

Users can select the method for computing IIS by setting the `"IISMethod"` parameter. For parameter setting in each interface, see *Parameter section*.

- `IISMethod`

  Integer parameter.

  The method for computing IIS.

  **Default value:** -1

  **Possible values:**

  -1: Automatically selected.

  0: Prioritize quality of the IIS result.

  1: Prioritize computation efficiency.

#### Attributes

COPT provides related attributes to describe the IIS result, mainly indicating the existence of an IIS and the number of variables/constraints in the IIS. See *Attributes section* for details.

Table 13.5: Overview of IIS-Related Attributes

| Attribute Name | Type | Description |
|---|---|---|
| *HasIIS* | Integer | Indicates whether an IIS exists. |
| *IsMinIIS* | Integer | Indicates whether the computed IIS is minimal. |
| *IISCols* | Integer | Number of variable bounds in the IIS. |
| *IISRows* | Integer | Number of constraints in the IIS. |
| *IISSOSs* | Integer | Number of SOS constraints in the IIS. |
| *IISIndicators* | Integer | Number of Indicator constraints in the IIS. |

#### Example Code

Users can write the infeasible model to a file and directly compute the IIS after loading it in COPT. For example, in Python:

```
from coptpy import *

env = Envr()
model = env.createModel("example")
model.read("example.lp")
model.computeIIS()
model.writeIIS("example.iis")
```

Sample code for computing IIS in different APIs can be found in the `"examples"` folder of the COPT installation package. For COPT Python Interface, the sample path is: `"/examples/python/iis_ex1.py"`.

## 13.2 Feasibility Relaxation for infeasible models

Feasibility relaxation is the process of minimizing the conflicts in the bounds of variables and constraints in the original infeasible model. Users can use the quantitative results from feasibility relaxation to adjust constraints or variable bounds, thus making the model feasible.

### 13.2.1 Computing Feasibility Relaxation

COPT provides functions in different interfaces for computing feasibility relaxation, and also allows writing the relaxed model to a file with the suffix `.relax` (e.g., `example.relax`). Related function names are listed in Table 13.6:

Table 13.6: Functions for Computing Feasibility Relaxation

| API | Compute FeasRelax | Write Relaxed Model to File |
|---|---|---|
| C | `COPT_FeasRelax` | `COPT_WriteRelax` |
| C++ | `Model::FeasRelax()` | `Model::WriteRelax()` |
| C# | `Model.FeasRelax()` | `Model.WriteRelax()` |
| Java | `Model.feasRelax()` | `Model.writeRelax()` |
| Python | `Model.feasRelax()` | `Model.writeRelax()` |

COPT supports two approaches for computing feasibility relaxation. In interfaces other than Python, users can use different function arguments:

1. Simplified version: Relax all variables and/or constraints in the model, only requiring two parameters to specify whether to relax all variables or all constraints.
   - `ifRelaxVars` : whether to relax variables (default: `True`)
   - `ifRelaxCons` : whether to relax constraints (default: `True`)
2. Full version: Accepts more parameters (specify variables/constraints, set penalty factors for bounds).
   - `vars`: variables to relax
   - `cons`: constraints to relax
   - `colLowPen`: penalty factor for variable lower bounds
   - `colUppPen`: penalty factor for variable upper bounds
   - `rowBndPen`: penalty factor for constraint bounds
   - `rowUppPen`: penalty factor for constraint upper bounds (for double-sided constraints)

   **Note:** In most cases, set `rowUppPen` as `NULL`.

The function names and argument conventions differ slightly across APIs, but the functionality and meaning are consistent. See each interface's API manual for details:

- C API: *Feasibility relaxation functions*
- C++ API: *Model class*
- C# API: *Model class*
- Java API: *Model class*
- Python API: *Model class*

**Notes**

- For the simplified feasibility relaxation approach, the Python API additionally provides the function `Model.feasrelaxS(vrelax, crelax)`, requiring only two arguments:
  - `vrelax`: whether to relax variables (default: `True`)
  - `crelax`: whether to relax constraints (default: `True`)

### 13.2.2 Feasrelax-related parameters, attributes and information

#### Parameters

Users can select the method for feasibility relaxation by setting the `"FeasRelaxMode"` parameter. See *Parameter section* for details.

- `FeasRelaxMode`

  Integer parameter.

  The method for computing feasibility relaxation.

  **Default value:** 0

  **Possible values:**

  0: Minimize the weighted-sum of violations.

  1: Minimize the original model objective under the weighted sum of violations.

  2: Minimize the number of violations.

  3: Minimize the original model objective under the minimum number of violations.

  4: Minimize the weighted-sum of squared violations.

  5: Minimize the original model objective under the weighted-sum of squared violations.

#### Attributes

COPT provides attributes to describe the result of feasibility relaxation, as shown in Table 13.7. See *Attributes section* for details.

Table 13.7: Overview of Feasibility Relaxation Attributes

| Attribute Name | Type | Description |
|---|---|---|
| *HasFeasRelaxSol* | Integer | Indicates whether a feasible relaxation solution exists. |
| *FeasRelaxObj* | Double | Feasibility relaxation value. |

### Information

COPT provides the following information, representing the amount of relaxation for lower and upper bounds of variables (or constraints). See *Information section*.

- `RelaxLB`

  Double information.

  The relaxation amount for the lower bound of a variable (column) or constraint (row).

- `RelaxUB`

  Double information.

  The relaxation amount for the upper bound of a variable (column) or constraint (row).

### Example Code

Sample code for feasibility relaxation can be found in the `"examples"` folder of the COPT installation package, with the file name `"feasrelax_ex1.py"`. For COPT Python API, the sample path is: `"/examples/python/feasrelax_ex1.py"`.

# Chapter 14

# MIP Starts

## 14.1 Utilities of MIP Starts

### 14.1.1 Set and Load MIP Starts

For MIP problems, COPT provides methods to specify initial solution value(s) for a single variable or a set of variables and load it/them into model. The parameters that can be specified are:

- `vars` :variables
- `startvals` :variables' solution values

The functions in different APIs are shown in Table 14.1:

Table 14.1: Functions for setting MIP starts

| API | Function |
|---|---|
| C | `COPT_AddMipStart` |
| C++ | `Model::SetMipStart()` |
| C# | `Model.SetMipStart()` |
| Java | `Model.setMipStart()` |
| Python | `Model.setMipStart(vars, startvals)` |

**Note**

- Regarding the operations of MIP starts, their **function names**, **calling methods**, and **parameter names** are slightly different in different programming interfaces, but the implementation of functions and meanings of parameters are the same.
- For more details on setting initial solutions in C, please refer to function `COPT_AddMipStart` in chapter *C API Function: MIP start utilities*.
- You may want to call this method several times to input the MIP start. Please call `loadMipStart()` once when the input is done.

### 14.1.2 Read and Write MIP Starts

COPT provides functions for file read/write. It can read variable values from an MIP start file (`".mst"`) as the initial solution values of variables, and write the solving results or existing initial solution to an MIP start file (`".mst"`). The functions for reading and writing MIP start file in different APIs are shown in Table 14.2:

Table 14.2: Functions for reading and writing MIP starts

| API | read MIP starts | write MIP starts |
|---|---|---|
| C | `COPT_ReadMst` | `COPT_WriteMst` |
| C++ | `Model::ReadMst` | `Model::WriteMst` |
| C# | `Model.ReadMst()` | `Model.WriteMst()` |
| Java | `Model.readMst()` | `Model.writeMst()` |
| Python | `Model.readMst()` | `Model.writeMst()` |

## 14.2 Related Parameters

COPT provides the following parameters that control how MIP starts (initial solutions) are handled inside.

- `MipStartMode`

  Integer parameter.

  Mode of MIP starts, i.e. how MIP starts are handled.

  **Default** -1

  **Possible values:**

  -1: Automatic.

  0: Do not use any MIP starts.

  1: Only load full and feasible MIP starts.

  2: Only load feasible ones (complete partial solutions by solving subMIPs).

  3: Accept all MIP starts. Intended for hierarchical multi-objective optimization.

**Note:** If the provided initial solution is incomplete (partial), `MipStartMode=2` needs to be set, otherwise the initial solution will be rejected.

- `MipStartNodeLimit`

  Integer parameter.

  Limit of nodes for MIP start sub-MIPs.

  **Default:** -1

  **Minimal:** -1

  **Maximal:** `INT_MAX`

## 14.3 Log of MIP starts

### 14.3.1 MIP starts are accepted

1. A (better) initial solution was provided

```
Initial MIP solution # 1 with objective value 9.73987 was accepted
```

2. A partial initial solution was provided, and set `MipStartMode=2`. (complete it by solving a subMIP)

```
Loading 1 initial MIP solution
Extending partial MIP solution # 1
Extending partial MIP solution # 1 succeed (0.2s)
Initial MIP solution # 1 with objective value 9.66566 was accepted
```

### 14.3.2 MIP starts are rejected

**1. The provided initial solution was infeasible**

```
Initial MIP solution # 1 was rejected: Primal Inf 1.00e+00  Int Inf 1.78e-15
```

**2. The provided initial solution was not better than the current best one**

```
Initial MIP solution # 2 with objective value 10.3312 was rejected (not better)
```

**3. The provided initial solution was incomplete (partial), and MipStartMode=2 was not set**

```
Loading 1 initial MIP solution
Initial MIP solution # 1 was rejected: partial
```

**4. The provided initial solution was incomplete (partial), and COPT failed to find a feasible solution by solving a subMIP**

```
Loading 1 initial MIP solution
Extending partial MIP solution # 1
Extending partial MIP solution # 1 failed (infeasible)
Initial MIP solution # 1 was rejected: partial
```

# Chapter 15

# MIP Solution Pool

In general, the solver finds multiple feasible solutions in the process of solving an MIP problem with the Branch-and-cut method. COPT provides a solution pool for MIP problems, from which users can obtain solutions and the corresponding objective function values. Supported optimization problem types are: MILP, MISOCP, MIQ(C)P.

COPT provides functions with which users can get the `iSol` th solution's objective function value and solution values (of specified variables) by specifying the following parameters.

- `iSol`: Index of the solution to obtain. (0-based)
- `vars`: Variables

The functions in different APIs are shown in Table 15.1:

Table 15.1: Get solutions and objective values from solution pool

| API | Get solution | Get objective value |
|---|---|---|
| C | `COPT_GetSolution` | `COPT_GetPoolObjVal` |
| C++ | `Model::GetPoolSolution()` | `Model::GetPoolObjVal()` |
| C# | `Model.GetPoolSolution()` | `Model.GetPoolObjVal()` |
| Java | `Model.getPoolSolution()` | `Model.getPoolObjVal()` |
| Python | `Model.getPoolSolution()` | `Model.getPoolObjVal()` |

**Note:** Regarding the operations of solution pool, their **function names**, **calling methods**, and **parameter names** are slightly different in different programming interfaces, but the implementation of functions and meanings of parameters are the same.

**Attributes of Solution Pool**

- `PoolSols`

  Integer attribute

  Number of solutions in the solution pool.

# Chapter 16

# COPT Tuner

## 16.1 Introduction

The COPT Tuner is a tool designed for tuning performance automatically for all supported problem types.

- For MIP problems, it supports tuning for solving time, relative gap, best objective value and objective bound;
- For non-MIP problems, only solving time is supported.

The workflow of the COPT tuning tool is as follows:

1. First, perform benchmark calculation and allow users to customize benchmark calculation parameters;
2. Next, generate tuning parameters one by one, and find parameter combinations that improve the solution performance through parameter tuning calculations.

## 16.2 Related parameters

Firstly, the tuner will do a baseline run, possibly with fixed parameters from users, then move to the improvement run, where the tuner will generate trial parameter sets and search for parameters that can improve the performance. To summarize, the COPT Tuner provides the following capabilities:

Table 16.1: Tuner related parameters

| Name | Type | Description |
|---|---|---|
| *TuneTimeLimit* | Double | Time limit for parameter tuning |
| *TuneTargetTime* | Double | Time target for parameter tuning |
| *TuneTargetRelGap* | Double | Optimal relative tolerance target for parameter tuning |
| *TuneMethod* | Integer | Method for parameter tuning |
| *TuneMode* | Integer | Mode for parameter tuning |
| *TuneMeasure* | Integer | Parameter tuning result calculation method |
| *TunePermutes* | Integer | Permutations for each trial parameter set |
| *TuneOutputLevel* | Integer | Parameter tuning log output intensity |

## 16.3 Provided capabilities

The COPT tuning tool provides the following capabilities:

### 16.3.1 Tuning method

Controlled by the parameter `TuneMethod`, options are: greedy search and aggressive search. The greedy method tries to find better parameter settings within a limited number of trials, while the aggressive method searches for more combinations and has a much larger search space than the greedy one, and can potentially find even better parameter settings at the expense of more elapsed tuning time. Default setting is to choose automatically.

- Greedy search strategy: It is expected to optimize the calculation with fewer parameters and find better parameter settings;
- Broader search strategy: tries more parameter combinations, has a larger search space, and is more likely to find better parameter settings, but also consumes more tuning time.

The possible values and corresponding meanings of the parameter `TuneMethod` are as follows. By setting it to a different value, the search method can be selected. The default setting is automatic selection.

- -1: Automatic selection
- 0: Greedy search strategy
- 1: Broader search strategy

### 16.3.2 Tuning mode

Controlled by the parameter `TuneMode`, options are: solving time, relative gap, objective value and objective bound. For MIP problems, by default, if the baseline run is not solved to optimality within specified time limit, tuner will change tuning mode to relative gap automatically. Default setting is to choose automatically.

- 0: Solving time
- 1: Optimal relative tolerance
- 2: Objective function value
- 3: Lower bound of objective function value

**Note:** For integer programming problems, by default, if the benchmark calculation does not optimize the model within the given time limit, the tuning tool will automatically switch the tuning mode to the optimal relative tolerance.

### 16.3.3 Tuning permutations

Controlled by the parameter `TunePermutes`. Tuner allows users to run more permutations for each trial parameter set to evaluate performance variability. Default setting is to choose automatically.

### 16.3.4 Tuning measure

Controlled by the parameter `TuneMeasure`, options are: by average or maximum. When users run more permutations for each trial, tuner will compute the aggregated tuning value by this measure. Default setting is to choose automatically.

- 0: Calculate the average
- 1: Calculate the maximum value

### 16.3.5 Tuning targets

Controlled by the parameters `TuneTargetTime` and `TuneTargetRelGap`. Tuner enables users to specify a target solving time or relative gap for tuning; when the tuner finds parameters that satisfy the specified target, it will stop tuning. For solving time, default value is 0.01 seconds, while for relative gap, default value is 1e-4.

### 16.3.6 Tuning output

Controlled by the parameter `TuneOutputLevel`, options are: no output, show summary for improved trials, show summary for each trial and show detailed log for each trial. Default setting is to show summary for each trial.

- 0: Do not output tuning log
- 1: Output only a summary of the improved parameters
- 2: Output a summary of each tuning attempt
- 3: Output a detailed log of each tuning attempt

### 16.3.7 TuneTimeLimit

Controlled by the parameter `TuneTimeLimit`. This parameter is used to control the overall time limit for the improvement run of tuning. Default setting is to choose automatically.

### 16.3.8 User defined parts

- User defined parameters

  The tool enables users to set parameters for the baseline run, which will also be used as fixed parameters for each trial run. Tuner will not tune parameters in the fixed parameters.

- User defined MIP start

  The COPT tuner enables users to set MIP start for the baseline run, which will also be used for each trial run.

- User defined tuning file

  The COPT tuner enables users to read tuning parameter sets from a tuning file. If so, the tuner will try to tune from the given parameter sets; otherwise, the tuner will generate tuning parameter sets automatically. The tuning file is similar to a parameter file, with the difference that it allows multiple values for each parameter name.

  **Note:** The COPT tuning file has a similar format to the COPT parameter file, except that the tuning file allows multiple values to be specified for a single parameter.

### 16.3.9 Loading or writing tuning parameters

After the parameter tuning is completed, the number of parameter tuning results can be obtained through the attribute `TuneResults`, and the tuning results of the specified number can also be loaded into the model or written into the parameter file.

COPT can output the parameter tuning results of the specified number to the parameter file (`".par"`). The parameters that need to be specified are:

- `idx`: parameter tuning result number
- `filename`: file name

The corresponding functions in different programming interfaces are as follows:

Table 16.2: functions for writing parameter tuning results in different interfaces

| API | function |
|---|---|
| C | `COPT_WriteTuneParam` |
| C++ | `Model::WriteTuneParam()` |
| C# | `Model.WriteTuneParam()` |
| Java | `Model.writeTuneParam()` |
| Python | `Model.writeTuneParam()` |

## 16.4 Example

For example, to tune model `"foo.mps"` from command line for solving time with COPT command line tool, the commands are:

```
copt_cmd -c "read foo.mps; tune; exit"
```

To use the tuner in API such as Python, the codes are:

```
env = Envr()
m = env.createModel()
m.read("foo.mps")
m.tune()
```

# Chapter 17

# Callbacks

COPT provides the callbacks utility, which supports users in obtaining information during the MIP solving process, e.g., the current best bound, the current optimal objective value, etc.; or controlling the solving process, e.g., by adding lazy constraints and cutting planes, or terminating the solving process. The problem types supporting the use of callbacks are MILP, MISOCP, MIQ(C)P.

A callback function is a user-provided function called by COPT during the solving process. The user can register one custom callback function via their preferred API for one or multiple callback contexts. Section *Using the callback utilities in different APIs* gives a detailed introduction to how to setup a callback function. The callback function will be invoked at certain moments during the solving process, depending on the callback contexts. When invoked, the user can *access information* and *control the solving process*, respectively. The available information and operations depend on the context. Currently, COPT supports four callback contexts:

- `CBCONTEXT_INCUMBENT`: Invokes the callback after a new incumbent was found.
- `CBCONTEXT_MIPNODE`: Invokes the callback after a MIP node was processed.
- `CBCONTEXT_MIPRELAX` : Invokes the callback when an LP-relaxation was solved.
- `CBCONTEXT_MIPSOL`: Invokes the callback when a new MIP candidate solution is found.

The content of this chapter is organized as follows:

- *Obtaining information during the solving process*
- *Controlling the MIP solving process*
- *Using the callback utilities in different APIs*

**Notes**

Only **one** callback function can be registered in COPT at a time. But one callback can be registered for multiple contexts. If a user wants to call different operations for different contexts (such as adding lazy constraints under `CBCONTEXT_MIPSOL` and adding user cuts under `CBCONTEXT_MIPRELAX` ), they need to register one callback for all relevant contexts and have this callback call the respective operations based on the context in which it was called.

## 17.1 Obtaining information during the solving process

The information that can be obtained during the MIP solving process depends on the context the callback is invoked in, see the table below. Information is usually obtained by calling (an API dependent version of) `getInfo` / `GetCallbackInfo` from within the callback function, specifying the desired information via a string supplied as the function argument. For a detailed description of the available callback information arguments, please refer to *Callback information*.

The following table lists information that can be obtained in different contexts:

| Context | Callback Information |
|---|---|
| `CBCONTEXT_MIPNODE` | `NodeStatus`, `RelaxSolution`, `RelaxSolObj`, `MipCandObj`, `MipCandidate` |
| `CBCONTEXT_MIPRELAX` | `RelaxSolution`, `RelaxSolObj` |
| `CBCONTEXT_MIPSOL` | `MipCandObj`, `MipCandidate` |
| `CBCONTEXT_INCUMBENT` | |

In addition to the corresponding Callback Context and Information listed above, `BestObj`, `BestBnd`, `HasIncumbent`, and `Incumbent` can be obtained in any context.

**Notes**

1. If `HasIncumbent == False`, then `Incumbent` cannot be obtained.
2. The return value of the "NodeStatus" information is constant, representing the solving status of the current node's LP relaxation. For possible values, please refer to *General Constants Section: Solution Status (Partial)*.
3. `Incumbent`, `RelaxSolution` , and `MipCandidate` are obtained through different methods in different interfaces:
   - C API: through the function `COPT_GetCallbackInfo`, the name of the intermediate information to be obtained is provided as arguments of the function;
   - In object-oriented programming languages (C++/C#/Java/Python), the `CallbackBase` class provides specialized functions to obtain the corresponding intermediate information. E.g., in Python/C++ `CallbackBase` provides `GetIncumbent`, `GetRelaxSol`, and `GetSolution`. Other programming language interfaces are similar, please refer to the `CallbackBase` class of each API.

## 17.2 Controlling the MIP solving process

COPT provides functions to allow the user to interactively add lazy constraints or cutting planes during the solving process of the MIP branch-and-cut to control the MIP solving process. There are three main types of operations:

1. Adding lazy constraints
2. Adding cutting planes
3. Adding feasible solutions

### 17.2.1 Adding lazy constraints

Lazy constraints are constraints that are added to the model only when they are violated. For some models with a large number of constraints, adding lazy constraints only when violated can effectively reduce the size of the model during the solution process and improve the efficiency of the solving process. A popular example of this is the TSP model, see `"cb_ex1"` in the examples directory in the installation package.

COPT supports two ways of adding lazy constraints. One is to explicitly add lazy constraints to the model **before** starting the solution process. The other is to add lazy constraints **during** the solving process through a user callback. For this purpose, each API provides two sets of methods, one for adding lazy constraints to the initial model and one for adding them from a callback. In the C API, the methods can be distinguished according to whether or not the function name contains `"Callback"`, e.g., `COPT_AddLazyConstr` and `COPT_AddCallbackLazyConstr`. In object-oriented APIs, the two sets of functions correspond to the `Model` class and the `CallbackBase` class respectively. Taking python as an example:

- Before solving, a user can directly add lazy constraints to the model by calling `Model.addLazyConstr()` or `Model.addLazyConstrs()` respectively.
- During the solving process, a user can dynamically add lazy constraints (if supported by the current context, see below) from within the callback function via `CallbackBase.addLazyConstr()` or `CallbackBase.addLazyConstrs()`.

In both cases, the added lazy constraints will be stored by COPT separately from the actual model and are only added to the model, when they are violated by a solution found during the solving process.

In order to ensure correctness, COPT will check whether any lazy constraints added so far are violated by any solution found during the solving process. This will increase the solving time, especially when many non-violated lazy constraints have been added. It is recommended that the user only adds lazy constraints when necessary, e.g. when they are violated by a solution.

It should be noted that although lazy constraints can be added in the `CBCONTEXT_MIPRELAX` context, this is supported only when the `CBCONTEXT_MIPSOL` context is also enabled. Otherwise, attempts to add lazy constraints in `CBCONTEXT_MIPRELAX` will result in an error. Therefore, while it is not strictly necessary to check every LP relaxation solution for violations of lazy constraints, users must verify that every solution provided in `CBCONTEXT_MIPSOL` satisfies all lazy constraints. Failing to do so may result in incorrect solutions being accepted.

To avoid adding unnecessarily many lazy constraints, COPT has some simple redundancy checks for lazy constraints in place. Exact duplicates will be discarded. However, adding many, very similar but redundant lazy constraints will negatively affect COPT's performance. This should be avoided by the user.

**Notes**

- Registering a callback function for the `CBCONTEXT_MIPSOL` will make COPT believe that the user wants to add lazy constraints. As lazy constraints are not actually part of the model, this will lead to the deactivation of dual reductions during COPT's presolve, as dual arguments rely on the knowledge of all model rows. If the user does not intend to add lazy constraints but still wants to use the `CBCONTEXT_MIPSOL`, COPT provides the *LazyConstaints* parameter which enables the user to explicitly tell COPT whether or not lazy constraints will be added to the model. By default, this parameter is set to `-1` meaning COPT will turn off dual presolve reductions if either lazy constraints are part of the model or a callback for context `CBCONTEXT_MIPSOL` has been installed. Explicitly setting the parameter to `0` will allow dual reductions during COPT's presolve even if lazy constraints or a callback for context `CBCONTEXT_MIPSOL` are present. This is useful only in very rare cases, e.g., if the callback only prints information about solution candidates but never adds lazy constraints. As soon as lazy constraints are added, this might lead to wrong results, however. For printing information about solutions, consider using the `CBCONTEXT_INCUMBENT` context instead.

- If a user invokes a function to add lazy constraints from a callback in the `CBCONTEXT_MIPSOL` context, the current MIP candidate solution will be rejected, no matter whether the added lazy constraint(s) are actually violated or not. This enables the user to reject arbitrary solutions by adding empty lazy constraints when an undesirable solution is found. Note however, that COPT might find the same solution multiple times if no lazy constraint is provided. The LP relaxation solution will not necessarily be rejected if lazy constraints are added in `CBCONTEXT_MIPRELAX`, only when these are actually violated.
- It is invalid to call the any functions of the `Model` class for object-oriented languages (or their C equivalent) to add lazy constraints in a callback. More generally, the model cannot be changed during the solving process, except by adding lazy constraints or cutting planes.

### 17.2.2 Adding cutting planes

Cutting planes are added to the model during the solving process to strengthen the LP relaxation, e.g., cut off fractional LP solutions and improve the lower bound of the MIP problem.

COPT supports the addition of custom cutting planes to the model during the solving process. Similar to lazy constraints, cutting planes can be added to the model **before** and, via the callback, **during** the solving process. Each API provides two sets of methods, one for adding cutting planes to the initial model and one for adding them from a callback. In the C API, the methods can be distinguished according to whether or not the function name contains `"Callback"`, e.g., `COPT_AddUserCut` and `COPT_AddCallbackUserCut`. In object-oriented APIs, the two sets of functions correspond to the `Model` class and the `CallbackBase` class respectively. Taking python as an example:

- Before solving, user can directly add cutting planes to the model by calling `Model.addUserCut()` or `Model.addUserCuts()`.
- During the solving process, a user can dynamically add cutting planes (if supported by the current context, see below) from within the callback function via `CallbackBase.addUserCut()` or `CallbackBase.addUserCuts()`.

Cutting planes can **only** be added in the `CBCONTEXT_MIPRELAX` context. Here, the user is provided with the current LP relaxation solution to separate their own cutting planes.

**Notes**

- Cutting planes that do not violate the current LP relaxation solution are discarded by COPT.
- It is invalid to call any functions of the `Model` class (or their C equivalent) to add cutting planes in a callback. More generally, the model cannot be changed during the solution process, except by adding lazy constraints or cutting planes.

### 17.2.3 Adding feasible solutions

COPT supports adding feasible solutions during the MIP solving process. This enables the user to provide any feasible solution they found in parallel to the COPT solution process, e.g., in a self-implemented heuristic. Known solutions can either be supplied as starting solutions by calling `COPT_AddMipStart` (see *MIP Starts*) or from within a callback function. If a solution is know beforehand, supplying it as a MIP starting solution is preferred. For solutions found during the solving process, in the C API, a solution can be added by calling `COPT_AddCallbackSolution` inside the callback. In object-oriented APIs, the functions needed to add a solution are provided by the `CallbackBase` class and the workflow consists of calling two functions:

- Set the feasible solution: `CallbackBase.setSolution(vars, val)`
- Load a custom solution into the model: `CallbackBase.loadSolution()`

COPT will check any provided solution for feasibility and compute its objective value. The computed objective is return by `loadSolution` / `COPT_AddCallbackSolution`. If a solution is infeasible or worse than the current incumbent, it is discarded and the objective value returned by COPT is set to `1.0e+30`.

Solutions can be added in any callback context.

**Notes**

Currently, COPT only supports complete feasible solutions within callbacks.

## 17.3 Using the callback utilities in different APIs

For object oriented programming languages the basic steps of setting up a callback function are:

1. Implement a custom callback class, inheriting from the `CallbackBase` class.
2. Implement the `CallbackBase.callback()` function. This is the callback function that will be invoked by COPT. Here, the user can invoke the callback-specific functions for *obtaining information* or *controlling the solution process*.
3. Create an object of the custom callback class.
4. Register the callback in COPT through `Model.setCallback()`, and input the Callback Context as a parameter. For registering the callback function for multiple contexts, one can bitwise-or the desired contexts, e.g., `COPT.CBCONTEXT_MIPSOL | COPT.CBCONTEXT_MIPNODE`.

In the subsequent solving process, the user-supplied `CallbackBase.callback()` function will be called in each context registered. The currently invoked context can be obtained by calling the `CallbackBase` class method `where()`.

As already mentioned in the sections above, functions in `CallbackBase` class (or their corresponding C functions) can only be called in certain contexts. The following table lists for each callback context the allowed callback operations, using the Python API:

| Context | Function |
|---|---|
| CBCONTEXT_INCUMBENT | `getInfo`, `getIncumbent`, `load/setSolution` |
| CBCONTEXT_MIPNODE | `getInfo`, `getIncumbent`, `getRelaxSol`, `load/setSolution` |
| CBCONTEXT_MIPRELAX | `addUserCut(s)`, `getIncumbent`, `getInfo`, `getRelaxSol`, `load/setSolution` |
| CBCONTEXT_MIPSOL | `addLazyConstr(s)`, `getIncumbent`, `getInfo`, `getSolution`, `load/setSolution` |

**Notes**

- While `getInfo` can be called in all contexts, the information available depends on the context. See *Obtaining information during the solving process* for details.
- In other APIs `getInfo` is often split into `getIntInfo` and `getDblInfo`.
- The functions above may look slightly different for a certain API, but the presented relationships are the same.

While the above steps use Python as the reference API, the implementation in each object oriented programming language API is similar, and the user can refer to the provided sample code. For Python, `"cb_ex1.py"` is available in the examples directory in the installation package. For C, the main differences when implementing and registering a callback are as follows:

- The custom callback function can be any function using the signature `int COPT_CALL <function>(copt_prob* prob, void* cbdata, int cbctx, void* usrdata)` where `<function>` is arbitrary.
- Instead of using `Where()` for obtaining the current context, the callback context is supplied in `cbctx`.
- Callback-relevant information can be passed by defining a custom `struct` and passing it as the `usrdata` argument.

See `cb_ex1.c` in the C examples folder for a reference implementation.

The calling method and function name of the callback function in different programming interfaces are **slightly different**, but the supported functions and function meanings are the same. Please refer to the corresponding chapters of different programming interface API reference manuals for specific introductions:

- C API: *Callback utilities*
- C++ API: *CallbackBase Class*
- C# API: *CallbackBase Class*
- Java API: *CallbackBase Class*
- Python API: *CallbackBase Class*

# Chapter 18

# Matrix Modeling

The COPT Python API provides matrix modeling, supports `NumPy` multi-dimensional arrays, two-dimensional NumPy matrices, SciPy compressed sparse column matrix ( `csc_matrix` ) and compressed sparse row matrix ( `csr_matrix` ) operations and can be combined with ordinary (scalar) variables and constraints.

`NumPy` version should be 1.23 or above and `Python` minimum version requirement is 3.8. COPT mainly provides the following utilities:

1. Add multi-dimensional variables ( `MVar` ) and other related operations;
2. Construct multi-dimensional linear expressions ( `MLinExpr` ), add multi-dimensional linear constraints ( `MConstr` ) and other related operations;
3. Construct multi-dimensional quadratic expression ( `MQuadExpr` ), add multi-dimensional convex quadratic constraint ( `QConstraint` ) and other related operations.

## 18.1 Two different matrix modeling modes

The matrix modeling function provided by COPT currently supports two modes: original legacy mode and experimental mode. Among them, the legacy mode relies on Python's `NumPy` library. The experimental mode (by default) is based on the built-in matrix modeling class implementation of the COPT C++ interface. The modeling speed could be improved to a certain extent compared with the former.

In the Python interface, the two modes can be controlled and switched through `Model.matrixmodelmode` , and the default is the experimental mode. Please set according to your usage scenarios:

- `Model.matrixmodelmode = "legacy"` , depends on the `NumPy` library.
- `Model.matrixmodelmode = "experimental"` (by default), C++ built-in matrix modeling class based on COPT ( `NdArray` ) , no external dependencies.

The functions and operation types supported by the two modes are slightly different.

1. The order (storage) method of `MConstr.reshape()/MVar.reshape()` , that is, the value of the function parameter `order`:
   - Legacy mode: supports both `'C'` (by row) and `'F'` (by column)
   - Experimental mode: only supports `'C'` (by row)
2. Advanced indexing support:

Take the nqueen problem for example, only one queen can appear on each diagonal:

- Legacy mode: Compatible with `NumPy`'s advanced index, which can be implemented as follows:

```
import coptpy as cp
import numpy as np

env = cp.Envr()
model = env.createModel()
model.matrixmodelmode = "legacy"
for i in range(1, 2*n):
    # At most one queen per diagonal
    diagn = (range(max(0, i-n), min(n, i)), range(min(n, i)-1, max(0, i-n)-1, -
↪1))
    model.addConstrs(x[diagn].sum() <= 1, nameprefix="diag"+str(i))
    # At most one queen per anti-diagonal
    adiagn = (range(max(0, i-n), min(n, i)), range(max(0, n-i), min(n, 2*n-i)))
    model.addConstrs(x[adiagn].sum() <= 1, nameprefix="adiag"+str(i))
```

- Experimental mode: The above advanced indexing method is not supported and requires the built-in Python indexing method, which can be implemented as follows:

```
import coptpy as cp
import numpy as np

env = cp.Envr()
model = env.createModel()
model.matrixmodelmode = "experimental"
for i in range(-n+1, n):
    # At most one queen per diagonal
    model.addConstrs(x.diagonal(i).sum() <= 1, nameprefix="diag"+str(i))
    # At most one queen per anti-diagonal
    model.addConstrs(x[:, ::-1].diagonal(i).sum() <= 1, nameprefix="adiag
↪"+str(i))
```

3. The relevant information obtained from matrix objects (objects such as `MConstr` and `MVar`), and the return value types are different:

- Legacy mode: `numpy.ndarray`
- Experimental mode: `coptcore.NdArray`

4. The supported dimensions are different:

- Legacy mode: There is no restriction on dimensions, so higher-dimensional matrices are supported.
- Experimental mode: The highest supported dimension is three.

5. When adding multi-dimensional quadratic constraints, the support for variable dimensions participating in constraint formation and the type of return value are different:

- Legacy mode: The dimension of the variables involved in forming the quadratic constraint needs to be 1 (i.e. vector), higher dimensions are not supported, and the return value type is *QConstraint Class* object.

```
import coptpy as cp
import numpy as np

env = cp.Envr()
model = env.createModel()
model.matrixmodelmode = "legacy"
Q = np.full((3, 3), 1)
mx = model.addMVar(3, nameprefix="mx")
# mqc <coptpy.QConstraint: >
mqc = model.addQConstr(mx@Q@mx<=1)
```

- Experimental mode: The dimensions of variables participating in the quadratic constraints can be one or two-dimensional, and the return value type is *MQConstr Class* object.

```
import coptpy as cp
import numpy as np

env = cp.Envr()
model = env.createModel()
model.matrixmodelmode = "experimental"
Q = np.full((3, 3), 1)
mx = model.addMVar((3, 3), nameprefix="mx")
# mqc <coptpy.MQConstr: shape=(3, 3)>
mqc = model.addQConstr(mx@Q@mx <= 1)
```

## 18.2 Multi-dimensional Variables

1. Add multi-dimensional variable `MVar`

   `MVar` contains operations related to multi-dimensional variables. Users can use `Model.addMVar()` to add a matrix variable `MVar` of any dimension and shape to the model. In addition to the need of specifying the argument `shape` (matrix shape), the rest of the arguments are consistent with ordinary variables, including: `lb` , `ub` , `vtype` , `nameprefix` .

   - Add one-dimensional continuous multi-dimensional variables: `x = Model.addMVar(3)`
   - Add two-dimensional 3x3 binary multi-dimensional variables: `y = Model.addMVar(shape=(3,3), vtype=COPT.BINARY)`

   In addition, the `MVar` multi-dimensional variables can also be sliced, such as: `y1 = y[:,0:2]`

2. Get multi-dimensional variable related attributes:
   - Number of dimensions: `MVar.ndim`
   - The shape of the multi-dimensional variable: `MVar.shape`
   - Number of elements in the multi-dimensional variable: `MVar.size`

## 18.3 Multi-dimensional array operations and expressions

### 18.3.1 Multi-dimensional Linear Expressions

Multi-dimensional variables and their coefficients (can be `ndarray` ) form a multi-dimensional linear expression (`MLinExpr`), and the supported operations mainly include:

1. Matrix multiplication: A @ x

```
x = model.addMVar(3)
A = np.array([[1, 0, 1],[0, 0, 1]])
expr1 = A @ x
```

2. Vector inner product

```
x = model.addMVar(3)
c = np.array([1, 2, 3])
expr2 = c @ x
```

### 18.3.2 Multi-dimensional Quadratic Expression

Common multi-dimensional quadratic expressions and their corresponding mathematical forms are as follows:

- x @ Q @ x: $x^TQx$
- x @ x: $x^Tx$
- x @ Q @ x + c @ x + b: $x^TQx + c^Tx + b$

### 18.3.3 Other multi-dimensional array operations

1. Combine with regular linear variables, regular linear expressions, and constants:

```
x = model.addMVar(3)
y = model.addVar()
c = np.array([1, 2, 3])
Q = np.full((3, 3), 1)
expr3 = 2 * x @ Q @ x + c @ x + 2 * y + 1
```

2. Self-increment/self-subtraction/self-multiplication operations:

```
mx = m.addMVar((3, 3))
B = np.array([[1, 0, 1], [0, 1, 1]])
expr_add = B @ mx
expr_add += 1
expr_add *= 2
```

**Notes**

- When we directly print the multi-dimensional expression with `print(MLinExpr)`/`print(MQuadExpr)` , the `shape` of the expression will be output at the same time. When `shape=()` , it means that the expression is a scalar (single linear/quadratic expression), corresponding to `ndim=0` , `size=1` . The same is true for multi-dimensional variables `MVar` ;
- When performing matrix multiplication (A@x), the matrix multiplication algorithm needs to be satisfied, and the number of columns of A and the number of rows of x need to be the same;
- COPT supports the combination of `MLinExpr` and `LinExpr` , but it should be noted that the `MLinExpr` needs `shape=()` at this time, and the final returned expression is `MLinExpr` with `shape=()` .

## 18.4 Matrix Constraints

### 18.4.1 Matrix linear Constraints

COPT supports two ways of adding multi-dimensional linear constraints, and the format provided by the function arguments is different:

1. `Model.addMConstr()` that specifically adds multi-dimensional linear constraints, the arguments that can be specified are:
   - `A` : coefficient matrix for linear constraints
   - `x` : decision variables ( `MVar` )
   - `sense` : type of linear constraint, the possible values are: `'L'` (<=), `'G'` (>=), `'E'` (=)

- `b` : right-hand-side of linear constraints (vector with dimensions equal to the number of rows of matrix `A`)
- `name` : name prefix for linear constraints

```
x = model.addMVar(shape=3, vtype=COPT.BINARY, nameprefix='x')
A = np.array([[1, 2, 3], [3, 2, 1]])
b = np.array([2, 5])
mconstrs = model.addMConstr(A, x, 'L', b, nameprefix='c')
obj = np.array([1, 2, 1])
model.setObjective(obj @ x, COPT.MINIMIZE)
```

2. Matrix linear constraints can be regarded as a set of linear constraints, so `Model.addConstrs()` can also add multi-dimensional linear constraints:

```
x = model.addMVar(shape=3, vtype=COPT.BINARY, nameprefix='x')
A = np.array([[1, 2, 3], [3, 2, 1]])
b = np.array([2, 5])
mconstrs = model.addConstrs(A @ x <= b, nameprefix='c')
obj = np.array([1, 2, 1])
model.setObjective(obj @ x, COPT.MINIMIZE)
```

### 18.4.2 Quadratic Constraints

COPT supports two ways of constructing multi-dimensional quadratic constraints, and the format provided by the function arguments is different:

1. `Model.addMQConstr()` that specifically adds multi-dimensional quadratic constraints, the arguments that can be specified are:
   - `Q` : quadratic coefficient matrix
   - `c` : vector of linear term coefficients, or `None` if there is no linear term
   - `sense` : type of quadratic constraint, the possible values are: `'L'` (<=), `'G'` (>=), `'E'` (=)
   - `rhs` : right-hand-side of quadratic constraints
   - `xQ_L` : the left-hand variable of the quadratic coefficient matrix Q (vector whose length is consistent with the number of rows of the matrix `Q` )
   - `xQ_R` : right-hand variable of the quadratic coefficient matrix Q (vector whose length is consistent with the number of columns of the matrix `Q` )
   - `xc` : the variables for the linear term, or `None` if there is no linear term
   - `name` : name prefix for quadratic constraints

```
Q = np.diag([3, 2, 1])
x = model.addMVar(3)
c1 = model.addMQConstr(Q, None, 'L', 1.0, x, x)
```

2. `Model.addQConstr()`, directly gives the multi-dimensional quadratic expression
   - `lhs` : multi-dimensional quadratic expression
   - `sense` : constraint type
   - `rhs` : right-hand-side of quadratic constraints

```
Q = np.diag([3, 2, 1])
x = model.addMVar(3)
c2 = model.addQConstr(x@Q@x<=1.0)
```

## 18.5 Objective function composed of multi-dimensional variables

COPT supports setting linear and quadratic objective functions, and provides two ways to set objective functions. The format of function arguments is different:

1. `Model.setMObjective()` that specifically sets the objective function composed of multi-dimensional variables, the arguments that can be specified are:
   - `Q` : quadratic coefficient matrix, or `None` if the objective function is linear
   - `c` : vector of linear term coefficients, or `None` if there is no linear term
   - `constant` : the constant term of the objective function
   - `xQ_L`: the left-hand variable of the quadratic term coefficient matrix Q (vector whose length is consistent with the number of rows of the matrix `Q`), or `None` if the objective function is linear
   - `xQ_R`: the right-hand variable of the quadratic coefficient matrix Q (vector, whose length is consistent with the number of columns in the matrix `Q`), or `None` if the objective function is linear
   - `xc`: the variable for the linear term, or `None` if there is no linear term
   - `sense`: direction of optimization, possible values are: `COPT.MINIMIZE` or `COPT.MAXIMIZE`
2. `Model.setObjective()` that directly gives the expression of the objective function
   - `expr`: Objective function expression, which can be linear or quadratic
   - `sense`: optimization direction, possible values are: `COPT.MINIMIZE` or `COPT.MAXIMIZE`

```
x = model.addMVar(shape=3, vtype=COPT.BINARY, nameprefix="x")
obj = np.array([1, 2, 1])
model.setObjective(obj @ x, COPT.MINIMIZE)
```

## 18.6 Indexing and Slicing

Multidimensional array objects in COPT (such as `NdArray`, `MVar`, `MConstr`, `MLinExpr`, etc.) support a unified set of indexing and slicing rules for accessing, selecting, or updating elements or sub-objects. Supported indexing forms include integer indexing, slice indexing, and multidimensional tuple indexing.

### 18.6.1 Integer Indexing

**Synopsis**

`mobj[i]`

**Description**

Integer indexing is used to access a single element in a multidimensional array object, where `i` is an integer index.

**Return value**

Returns a scalar value representing the element at the specified index. The return type corresponds to the underlying element type of the array.

**Example**

```
x = NdArray(args=[1.0, 2.0, 3.0], shape=(3,))
v1 = x[0]    # v1 is a scalar with value 1.0
v2 = x[-1]   # v2 is a scalar with value 3.0
```

### 18.6.2 Slice Indexing

**Synopsis**

```
mobj[start:stop:step]
```

```
mobj[start:stop]
```

```
mobj[:]
```

**Description**

Slice indexing is used to select a contiguous range of elements along a given dimension. The parameters `start`, `stop`, and `step` are all optional and follow the semantics of Python's built-in slicing:

- `start`: starting index (inclusive)
- `stop`: ending index (exclusive)
- `step`: step size

**Return value**

Returns a new multidimensional array object representing the sliced sub-array. The element type is the same as that of the original object.

**Example**

```
x = NdArray(args=[1.0, 2.0, 3.0, 4.0], shape=(4,))
y = x[1:3]       # NdArray, [2.0, 3.0]
z = x[-3:-1]     # NdArray, [2.0, 3.0]
w = x[:]         # NdArray, same contents as x
```

### 18.6.3 Tuple Indexing

**Synopsis**

```
mobj[i, j]
```

```
mobj[i, :]
```

```
mobj[:,  j]
```

```
mobj[i, j:k]
```

**Description**

For multidimensional array objects, integer indexing and slice indexing can be applied simultaneously across multiple dimensions using a tuple indexing expression. Each element in the tuple corresponds to one dimension of the array and may be either an integer index or a slice.

**Return value**

- If all dimensions are indexed using integers, a scalar value is returned.
- If any dimension uses slice indexing, a multidimensional array object is returned.

**Example**

```
x = NdArray(shape=(3, 3))

v = x[0, 1]     # scalar
r = x[0, :]     # NdArray with shape (3,)
c = x[:, 1]     # NdArray with shape (3,)
```

Regarding the matrix modeling method, the COPT Python interface provides multi-dimensional variables, (linear and convex quadratic) expressions, and matrix constraint classes respectively, and contains related operations. For the methods and specific introductions included, please refer to the corresponding part of the Python API:

- Multi-dimensional variable: *MVar*
- Multi-dimensional linear expression: *MLinExpr*
- Multi-dimensional quadratic expressions: *MQuadExpr*
- Matrix constraints: *MConstr*

# Chapter 19

# Multi-objective Optimization

In real-world applications and decision-making systems, there is often more than one objective to optimize. For example, in supply chain management, one may aim to minimize inventory cost while maximizing order fulfillment rate. COPT provides multi-objective optimization functionality to properly balance the priorities or weights of multiple objectives, using either a hierarchy method or a weighted-sum method, to achieve optimal decisions under multi-objective scenarios.

## 19.1 Modeling Multiple objectives

COPT currently supports linear objective functions for multi-objective optimization. The API function `Model.setObjectiveN()` is used to define them.

When constructing multi-objective functions, the following key objective parameters must be specified:

- `MultiObjPriority`

  The priority of the objective in multi-objective optimization. Higher values indicate higher priority. This determines the order of optimization in the hierarchy method.

  **Default value** 0.0

- `MultiObjWeight`

  The weight of the objective in multi-objective optimization. This defines the coefficient of the objective in the weighted-sum method.

  **Default value** 1.0

- `MultiObjAbsTol`

  The absolute tolerance allowed for degradation for the specified objective in multi-objective optimization.

  COPT optimizes objectives sequentially based on their priority: In multi-objective MIP, if the current objective's optimal value is `z`, the solution is allowed to degrade `z` by at most `MultiObjAbsTol` in subsequent groups. In multi-objective (continuous) LP, this parameter applies to variables. To maintain solution quality of higher-priority objectives, COPT fixes some variable values from the previous optimal solution based on reduced cost. More precisely, all variables with absolute reduced costs larger than `MultiObjAbsTol` will be fixed to their value in the previous optimal solution.

  **Default value** 1e-6

- `MultiObjRelTol`

  The relative tolerance allowed for degradation for the specified objective in multi-objective optimization.

COPT optimizes objectives sequentially based on their priority: In multi-objective MIP, if the current objective's optimal value is `z`, subsequent optimization phases allow its objective value to deviate within `MultiObjRelTol * |z|`. In multi-objective (continuous) linear programming, the degradation is controlled by `MultiObjAbsTol`, and `MultiObjRelTol` is ignored.

**Default value** 0.0

**Notes**

- The above parameters apply only to multiple objectives.
- When specifying a multi-objective function with `setObjectiveN`, in addition to specifying the above parameters, the objective index, objective expression, and optimization sense can also be specified.

Following are operations on multi-objective parameters (e.g. COPT Python API):

1. Use `Model.setObjParamN(idx, paramname, newval)` to set the parameter of a specific objective function by index.
2. Use `Model.getObjParamN(idx, paramname)` to retrieve the value of a parameter for a specific objective function by index.
3. Use `Model.resetObjParamN(idx)` to reset the parameters of a specific objective function to their default values.

## 19.2 Solving Multiple objectives

When solving multi-objective models, COPT supports two methods: hierarchy method and weighted-sum method.

### 19.2.1 Hierarchy Method

The hierarchy method optimizes objectives in descending order of priority. COPT solves each objective while ensuring that the quality of higher-priority objectives is not degraded.

Users can specify priorities while calling `Model.setObjectiveN` or using `Model.setObjParamN` to assign a priority to the specified objective. The default priority of each objective is 0.0.

Assuming a model with two objectives, $obj_1$ and $obj_2$, with priorities 3 and 2 respectively, COPT will first solve the problem for $obj_1$, and then solve for $obj_2$ within the allowable degradation defined for $obj_1$.

### 19.2.2 Weighted-sum Method

For objectives with the same priority, COPT uses the weighted-sum method to combine them into a single objective using the specified weights. Weights can be set while calling `Model.setObjectiveN` or using `Model.setObjParamN`. The default weight of each objective is 1.0.

Assume a model includes two objectives at the same priority level: $\min f_1(x)$ and $\min f_2(x)$, with weights $w_1$ and $w_2$ respectively. Then, COPT constructs the combined objective below:

$$\min \; w_1 \cdot f_1(x) + w_2 \cdot f_2(x) \tag{19.1}$$

**Note**

- Please be cautious when assigning a negative weight, as it effectively changes the optimization direction.
- Avoid assigning excessively large or small weights. A large weight may cause numeric instability, while a small weight may result in an insignificant contribution, potentially ignored due to objective degradation tolerances.

COPT also supports combining the hierarchy method and weighted-sum method together. Each objective can have different/same priority and weight. Objectives are grouped and solved sequentially by priority. Within the same priority group, objectives are combined using the weighted-sum method.

Consider a multi-objective model with four objectives:

$$\begin{aligned} &\min\ \text{obj}_1 = f_1(x) \quad (\text{priority} = 2,\ \text{weight} = 1.0) \\ &\max\ \text{obj}_2 = f_2(x) \quad (\text{priority} = 1,\ \text{weight} = 2.5) \\ &\max\ \text{obj}_3 = f_3(x) \quad (\text{priority} = 1,\ \text{weight} = 1.5) \\ &\min\ \text{obj}_4 = f_4(x) \quad (\text{priority} = 0,\ \text{weight} = 1.0) \end{aligned} \tag{19.2}$$

COPT will first group the objectives by priority (2, 1, 0), and solve them in descending order. For objectives with the same priority, a weighted sum will be used.

Priority = 2:

$$\min\ f_1(x) \tag{19.3}$$

Priority = 1:

$$\max\ 2.5 \cdot f_2(x) + 1.5 \cdot f_3(x) \tag{19.4}$$

Priority = 0:

$$\min\ f_4(x) \tag{19.5}$$

### 19.2.3 Objective degradation tolerance

When using the hierarchy method for multi-objective optimization, the degree to which a higher-priority objective is allowed to degrade while lower-priority ones are optimized can be adjusted via two parameters: `MultiObjRelTol` (relative degradation tolerance) and `MultiObjAbsTol` (absolute degradation tolerance).

1. MIP Model

For MIP models, we suppose an objective min $z_1$ has an optimal value $v$, and the corresponding `MultiObjAbsTol = e`. Then, when optimizing the next lower-priority objective min $z_2$, COPT searches for the optimal solution of $z_2$ within the feasible region defined by $z_1 \leq v + e$.

For MIP models, we suppose an objective min $z_1$ has an optimal value $v$, and the corresponding `MultiObjRelTol = q`. Then, when optimizing the next lower-priority objective min $z_2$, COPT searches for the optimal solution of $z_2$ within the feasible region defined by $z_1 \leq v + q * |v|$.

Note: If both relative and absolute tolerances are set for an objective, COPT will use the larger of the two allowed degradation tolerances.

2. LP Model

For continuous LP models, in order to maintain the quality of higher-priority objectives, COPT fixes certain variables to their optimal values obtained in the previous priority group. Whether a variable is fixed will be determined by whether its reduced cost is zero within the `MultiObjAbsTol` tolerance.

Note: `MultiObjRelTol` is ignored in LP models.

## 19.3 Retrieving the results of multiple objectives

The optimization results of the multi-objective model can be queried via `Model.getAttrN(idx, attrname)` (taking COPT-Python API as an example, other interfaces work the same way).

- `idx`: The index of the objective.
- `attrname`: The attribute name of the objective.

Supported attributes for multiple objectives include: *HasQObj*, *HasNLObj*, *ObjVal*, *LpObjVal*, *BestObj*, *ObjSense*, *ObjConst*.

In addition, the overall objective value of a multi-objective optimization model can be obtained through `Model.getAttr("ObjVal")` or `Model.ObjVal`. In a multi-objective optimization model, this value represents the individual objective value of the sub-objective function with index 0 at the current solution. When evaluating multi-objective optimization results, it is recommended to obtain the objective value of each sub-objective function by index through `Model.getAttrN(idx, attrname)`.

# Chapter 20

# Logging

Logging related parameters and functions are essential for users to control the display of solving logs of COPT. This chapter provides an interpretation of loggings for different algorithms, including the following sections:

- *Parameters and Functions for Logging*
- *Basic Information Section*
- *Simplex Logging*
- *Barrier Logging*
- *Branch-and-Cut Logging*
- *First-order Method (PDLP) Logging for GPU Solver*

## 20.1 Parameters and Functions for Logging

Users can control whether to show logs and how detailed they are by setting logging-related arguments.

- `Logging`

  Integer argument.

  Whether to print optimization logs.

  **Default:** 1

  **Possible values:**

  0: Do not print optimization logs.

  1: Print optimization logs.

- `LogLevel`

  Integer argument.

  Controls the level of detail in the optimization logs.

  **Default:** 2

  **Possible values:**

  2: Print basic optimization logs.

  3: In addition to the basic logs, print memory usage information (for MIP problems).

- `LogToConsole`

Integer argument.

Whether to print optimization logs to the console.

**Default:** 1

**Possible values:**

0: Do not print logs to the console.

1: Print logs to the console.

COPT provides operations related to logs, such as setting the logging file. COPT provides functions to set the logging file, write the optimization logs into the specified file (with the file name suffix `.log` ), allowing users to save and review the logs. The functions for different programming interfaces are shown in Table 20.1:

Table 20.1: Functions for setting logging files in different programming interfaces

| Programming Interface | Functions for setting logging files |
|---|---|
| C | `COPT_SetLogFile` |
| C++ | `Model::SetLogFile()` |
| C# | `Model.SetLogFile()` |
| Java | `Model.setLogFile()` |
| Python | `Model.setLogFile()` |

**Note:** When calling these functions, users should specify the file name for saving logs using the `logfile` parameter.

## 20.2 Basic Information Section

COPT outputs basic information before starting the solving process, depending on the problem types. The following information is typically displayed:

This section prints the overview information of the model and optimization task, such as model size and presolve results, and the numerical characteristics, helping users quickly understand the problem structure and numerical status.

### 20.2.1 Solver Environment and Model Overview

At the beginning of the log, COPT prints an overview of the solver environment and the model, including the model fingerprint (`fingerprint`), COPT version, platform, and problem type.

Example:

```
Using Cardinal Optimizer v8.0.7 on Windows
Model fingerprint: 2c27ab28
Hardware has 64 cores and 128 threads. Using instruction set X86_AVX512_E1 (14)
Minimizing a MIP problem
```

Here, `Model fingerprint` is a unique identifier of the current model. Hardware information includes CPU cores (`cores`) and threads (`threads`). COPT automatically detects the problem type and optimization sense, and then selects appropriate algorithms to solve it, for example:

`Minimizing an LP problem`, `Minimizing a MIP problem`, `Minimizing an SDP problem`, etc.

### 20.2.2 Model Size and Presolve

First, COPT prints the size of the original model, for example:

```
The original problem has:
    404 rows, 1200 columns and 2598 non-zero elements
    200 binaries and 1000 integers
```

For LP problems, the size includes:

- Constraints (`rows`)
- Variables (`columns`)
- Nonzeros in the constraint matrix (`non-zero elements`)

For MIP problems, it additionally includes integer-variable statistics:

- Binary variables (`binaries`)
- General integer variables (`integers`)

**Note**

- For MIP problems, the variable count (`columns`) includes all variables in the model: continuous, integer, and binary.
- In theory, binaries are a special case of integer variables, but here they are reported separately.

For nonlinear problems, the log also reports corresponding nonlinear structures, for example:

- Semidefinite variables (`PSD columns`)
- Quadratic terms in the objective (`quadratic objective elements`)
- Quadratic constraints (`quadratic constraints`)
- Second-Order cones (`SOC rows`)
- Exponential cones (`exponential cones`)

With default settings, before the optimization starts, COPT presolves to transform or reduce the original model to improve quality; the algorithm then works on the presolved model. The presolve part prints the model sizes. Example:

```
Presolving the problem
The presolved problem has:
    373 rows, 1169 columns and 2505 non-zero elements
    369 binaries and 800 integers
```

### 20.2.3 Problem Numerical Characteristics

This part prints key numerical characteristics of the model to help assess numerical issue (e.g., whether coefficient ranges span too many orders of magnitude). The items are:

- `Range of matrix coefficients`: Range of the constraint matrix coefficients
- `Range of rhs coefficients`: Range of right-hand-side coefficients
- `Range of bound coefficients`: Range of variable bounds
- `Range of cost coefficients`: Range of objective function coefficients
- `Density of cost`: Density of the objective coefficients (ratio of nonzeros)

Example:

```
Problem info:
    Range of matrix coefficients:    [5e-01,4e+00]
    Range of rhs coefficients:       [7e+02,2e+04]
    Range of bound coefficients:     [8e+02,3e+03]
    Range of cost coefficients:      [6e-02,3e-01]
```

## 20.3 Simplex Logging

Based on different stages of the optimization process, the logs for Simplex Method can be divided into two parts:

1. Simplex iteration process
2. Summary of optimization results

This section uses the logs for example problem afiro.mps to interpret the information in the Simplex Method logging.

### 20.3.1 Simplex Iteration Process

This part of the logs provides relevant information about the iteration process using the Simplex method.

```
Starting the simplex solver using up to 8 threads

Method   Iteration           Objective  Primal.NInf   Dual.NInf        Time
Dual             0   -4.8553789460e+02            3           0       0.00s
Dual             3   -4.6476735494e+02            0           0       0.00s
Postsolving
Dual             3   -4.6475314286e+02            0           0       0.00s
```

Here, the first line indicates that the current optimization algorithm is the Simplex method, and it uses 8 threads (`threads`) for computation.

The subsequent lines represent the simplex iteration process with 6 columns:

- `Method`: The optimization algorithm used, where `"Dual"` represents the dual simplex method.
- `Iteration`: The number of iterations.
- `Objective`: The objective function value.
- `Primal.NInf`: The number of primal infeasibilities in the primal problem.
- `Dual.NInf`: The number of dual infeasibilities in the dual problem.
- `Time`: The time taken for solving (in seconds).

### 20.3.2 Solution Summary

This part of the logs summarizes the results and the iteration process of the Simplex method after completing the solving process.

```
Solving finished
Status: Optimal  Objective: -4.6475314286e+02  Iterations: 3  Time: 0.00s
```

The included information consists of:

- Solving status (`Status`): If the model has an optimal solution, it is `Optimal`.

- Objective function value (`Objective`): If the model has an optimal solution, `Objective` displays the optimal objective function value.
- Total number of iterations (`Iterations`).
- Total solving time (`Time`).

If the model is infeasible, the corresponding log output is as follows:

```
Solving finished
Status: Infeasible  Objective: -  Iterations: 2  Time: 0.00s
```

## 20.4 Barrier Logging

By solution phase, the Barrier log consists of two parts:

1. Barrier solution Process
2. Solution summary

**Note:** By setting the optimization parameter `"LpMethod = 2"`, you can choose the Barrier method as the algorithm for solving linear programming problems.

Again, we take afiro.mps as an example to explain the Barrier log for LP.

### 20.4.1 Barrier Iteration Process

First, the log prints numerical information related to Barrier:

```
Starting barrier solver using 64 threads

Problem info:
Dualized in presolve:            No
Range of matrix coefficients:    [4e-01,4e+00]
Range of rhs coefficients:       [8e+01,3e+02]
Range of bound coefficients:     [4e+01,1e+02]
Range of cost coefficients:      [2e-01,2e+00]

Factor info:
Number of free columns:          0
Number of dense columns:         0
Number of matrix entries:        2.800e+01
Number of factor entries:        2.800e+01
Number of factor flops:          1.140e+02
```

The first line shows the algorithm (Barrier) and thread count (`threads`). Then:

- `Problem info` includes whether the model is dualized in presolve, and the ranges of matrix, RHS, bounds, and costs.
- `Factor info` includes the number of free columns, dense columns, and factorization-related statistics.

Next comes the Barrier iteration table, which reports iteration (`Iter`), objectives, and time:

- `Iter`: Iteration number
- `Primal.Obj`: Primal objective
- `Dual.Obj`: Dual objective
- `Compl`: Complementarity violation

- `Primal.Inf`: Primal infeasibility
- `Dual.Inf`: Dual infeasibility
- `Time`: Elapsed time (s)

```
Iter       Primal.Obj         Dual.Obj      Compl  Primal.Inf  Dual.Inf    Time
   0  +2.07010046e+01  -4.97632246e+02   5.89e+03    4.50e+02  2.65e+00   0.02s
   1  -1.18912241e+02  -5.91808560e+02   7.58e+02    3.36e+01  1.61e-01   0.02s
   2  -3.98096520e+02  -4.77597371e+02   2.28e+02    9.32e+00  7.45e-02   0.02s
   3  -4.55223227e+02  -4.68222895e+02   1.86e+01    3.60e-01  2.63e-03   0.02s
   4  -4.64587726e+02  -4.64803786e+02   2.52e-01    7.80e-03  7.93e-06   0.02s
   5  -4.64753143e+02  -4.64753143e+02   3.11e-07    7.80e-09  1.56e-11   0.02s
```

### 20.4.2 Barrier Summary

Key items include solution status, primal/dual optimal objectives, etc.

```
Barrier status:                 OPTIMAL
Primal objective:               -4.64753143e+02
Dual objective:                 -4.64753143e+02
Duality gap (abs/rel):          2.61e-07 / 5.63e-10
Primal infeasibility (abs/rel): 7.80e-09 / 2.60e-11
Dual infeasibility (abs/rel):   1.56e-11 / 6.99e-12
```

**Crossover**

```
Starting crossover using up to 8 threads

    1 primal pushes remaining      0.03s
    0 primal pushes remaining      0.03s
    1 dual pushes remaining        0.03s
    0 dual pushes remaining        0.03s

Method   Iteration           Objective  Primal.NInf   Dual.NInf        Time
Dual             1   -4.6475314286e+02            0           0       0.03s
Postsolving
Dual             1   -4.6475314286e+02            0           0       0.03s
```

### 20.4.3 Solution Summary

Printed after solving finishes; it summarizes the model solution and the (final) crossover cleanup.

```
Solving finished
Status: Optimal  Objective: -4.6475314286e+02  Iterations: 1  Time: 0.03s
```

Same as the Simplex log, it includes:

- `Status`: If the model is solved to optimality, the status will be `Optimal`
- `Objective`: If solved to optimality, the optimal objective value
- `Iterations`: Total iterations
- `Time`: Total solving time

**Note**

During the COPT barrier optimization process, the log may contain messages such as `Problem looks infeasible` or `Problem looks unbounded`. These messages indicate that the standard barrier algorithm (the non-homogeneous algorithm) has made a preliminary assessment of the model status based on the behavior of the current iterates, rather than rigorously verifying infeasibility or unboundedness by residual computations.

For LP models, with the default setting `BarHomogeneous = -1`, COPT automatically restarts the homogeneous algorithm when the standard barrier algorithm determines that the model `looks infeasible` or `looks unbounded`. If `BarHomogeneous` is explicitly set to `0`, this confirmation step is disabled. In this case, set `BarHomogeneous` to `-1` or `1` to obtain a more reliable determination of infeasibility or unboundedness.

For QP models, the homogeneous algorithm does not provide the same theoretical verification guarantees as in the LP case. Therefore, COPT does not restart the homogeneous algorithm to further verify the above assessment. If the QP log contains `Problem looks infeasible` or `Problem looks unbounded`, users who need to rigorously confirm the model status should perform further analysis through model inspection, numerical diagnostics, or other appropriate approaches.

## 20.5 Branch-and-Cut Logging

By solution phase, the Branch-and-Cut log consists of two parts:

1. Search Process
2. Solution Summary

We use the `cutstock.mps.gz` example (available under `/examples/data` in the COPT installation) to explain the MIP log.

### 20.5.1 Branch-and-Cut Search Process

This section of the log provides information about the Branch-and-Cut search process.

```
Starting the MIP solver with 8 threads and 32 tasks

    Nodes    Active  LPit/n  IntInf     BestBound  BestSolution    Gap   Time
        0         1      --       0  3.100000e+01            --    Inf  0.05s
H       0         1      --       0  3.100000e+01  6.800000e+01  54.4%  0.70s
H       0         1      --       0  3.100000e+01  6.600000e+01  53.0%  0.70s
H       0         1      --       0  3.100000e+01  6.500000e+01  52.3%  0.71s
        0         1      --      86  5.591304e+01  6.500000e+01  14.0%  0.72s
H       0         1      --      86  5.591304e+01  6.200000e+01  9.82%  0.74s
        1         2     0.0      86  5.591304e+01  6.200000e+01  9.82%  0.76s
H       1         1    2129       6  6.000000e+01  6.000000e+01  0.00%  0.87s
        2         0    1064       6  6.000000e+01  6.000000e+01  0.00%  0.87s
        2         0    1064       6  6.000000e+01  6.000000e+01  0.00%  0.87s
```

**Note:** For brevity, only a portion of the log is shown for explanation.

By meaning, the columns can be grouped as follows; we explain each group below:

- Node search information (columns 1-4)
- Feasible solution interval information (columns 5-7)
- Solving time information (column 8)

**Node search information (columns 1-4):**

- `Nodes`: Number of nodes searched
- `Active`: Number of leaf nodes yet to be searched
- `LPit/n`: Average simplex iterations per node
- `IntInf`: Number of integer variables that are fractional in the current LP relaxation

**Feasible solution interval information (columns 5-7):**

- `BestBound`: Current best bound on the objective
- `BestSolution`: Current incumbent objective value
- `Gap`: Relative gap between bound and incumbent. If it drops below the `RelGap` threshold, solving process stops

**Solving time information (column 8):**

- `Time`: Total solving time

**Notes**

- The prefix before the first column (`H/*`) indicates a new incumbent solution:
  - `H`: Found by a heuristic
  - `*`: Found by branching (subproblem) search
- Sometimes `Nodes` remains 0 for a while, which means COPT is still processing the root node—typically generating cuts or running heuristics to obtain a good incumbent, in order to reduce subsequent search.

### 20.5.2 Solution Summary

Printed after solving finishes, it summarizes the final MIP status and the search effort, including the model solution and search workload.

```
Best solution   : 60.000000000
Best bound      : 60.000000000
Best gap        : 0.0000%
Solve time      : 0.87
Solve node      : 2
MIP status      : solved
Solution status : integer optimal (relative gap limit 0.0001)

Violations    :     absolute     relative
bounds        :            0            0
rows          :            0            0
integrality   :            0
```

Solution summary includes:

- Best objective (`Best solution`)
- Best bound (`Best bound`)
- Best gap (`Best gap`)
- Solution status (`Solution status`)

Search workload includes:

- Solve time (`Solve time`)

- Number of explored nodes (`Solve node`)

The `Violations` block reports the satisfaction of bounds and constraints at the solution, including:

- Violations in variable bounds (`bounds`) and constraints (`rows`)
- Integrality violations (`integrality`)

## 20.6 First-order Method (PDLP) Logging for GPU Solver

For Linear Programming Problems, if the PDLP method is selected (setting the parameter `LpMethod=6` ), GPU solver can be enabled. (The machine needs the compatible GPU and the necessary CUDA library needs to be configured).

The logging for GPU solver can be divided into the following sections, which are slightly different from the CPU solver, with the main distinction in the second section:

1. GPU hardware information of the machine
2. First-order method PDLP iteration process
3. Crossover section
4. Summary of optimization results

Taking the `"thk_63"` instance from public LP benchmark as an example, the following is the solving log for the GPU solver.

### 20.6.1 GPU Hardware Information

This section outputs information about the current machine's GPU information and CUDA version.

```
Hardware has 1 supported GPU device with CUDA 12.3
  GPU 0: NVIDIA GeForce RTX 4090 (CUDA capability 8.9)
```

**Notes**

1. The `"CUDA 12.3"` mentioned in the log refers to the highest version of the CUDA Toolkit supported by the currently installed CUDA driver.
2. COPT's GPU solver requires a minimum CUDA Toolkit version of 12.0.1.

### 20.6.2 First-order Method(PDLP) Iteration Process

This section outputs the solving iteration process of the PDLP method, and the summary after the completion, including the iteration count of PDLP, and the optimal objective values, dual gaps for primal and dual problems.

```
Starting PDLP solver on GPU 0

Iterations      Primal.Obj        Dual.Obj         Gap  Primal.Inf  Dual.Inf    Time
        0  +6.00000000e+00  +6.00000000e+00  +0.00e+00    7.87e+00  0.00e+00  21.63s
     4000  +1.95436674e+03  -1.77166004e+03  +3.73e+03    2.09e-02  0.00e+00  33.37s
     8000  +1.90433201e+03  +1.55817851e+03  +3.46e+02    1.51e-02  0.00e+00  44.75s
    12000  +1.87801607e+03  +1.85689627e+03  +2.11e+01    1.74e-02  0.00e+00  56.27s
    16000  +1.86810632e+03  +1.86897715e+03  +8.71e-01    4.92e-03  0.00e+00  67.72s
    20000  +1.87022842e+03  +1.86994685e+03  +2.82e-01    3.42e-03  0.00e+00  79.18s
    23640  +1.87099459e+03  +1.87099144e+03  +3.15e-03    4.69e-05  0.00e+00  89.68s
```

(continues on next page)

(continued from previous page)

```
PDLP status:                      OPTIMAL
PDLP iterations:                  23640
Primal objective:                 1.87099459e+03
Dual objective:                   1.87099144e+03
Primal infeasibility (abs/rel):  4.69e-05 / 6.20e-07
Dual infeasibility (abs/rel):    0.00e+00 / 0.00e+00
Duality gap (abs/rel):           3.15e-03 / 8.43e-07

Experimental: using crossover to find a basic solution after PDLP
Please set parameter Crossover to 0 if the basic solution is not required
Please set parameter PDLPTol to a smaller value if the crossover cleanup takes too␣
↪long

Starting crossover using up to 8 threads

50320 primal pushes remaining     92.09s
12495 primal pushes remaining     97.71s
 4124 primal pushes remaining       102s
  202 primal pushes remaining       104s
    0 primal pushes remaining       104s
1480858 dual pushes remaining       104s
 589582 dual pushes remaining       106s
      0 dual pushes remaining       107s

Method   Iteration           Objective  Primal.NInf   Dual.NInf        Time
Dual        986011    1.8710000000e+03            0           0     107.97s
Postsolving
Dual        986011    1.8710000000e+03            0           0     110.56s
Unfolding
Dual       1036742    1.8710000000e+03            0           0     157.25s

Solving finished
Status: Optimal  Objective: 1.8710000000e+03  Iterations: 1036742  Time: 157.69s
```

**Note:** After solving with a First-order method (PDLP), if the solution reaches optimality (`Status: Optimal`), the default behavior is to perform the Crossover process to the basic solution. This process can also be disabled by setting the parameter *Crossover* to 0.

# Chapter 21

# File formats

## 21.1 File format list

The file formats currently supported by COPT are listed below:

Table 21.1: Supported file formats

| Fileformat | File extension |
|---|---|
| MPS model file | `.mps`, `.mps.gz` |
| LP model file | `.lp`, `.lp.gz` |
| SDPA model file | `.dat-s`, `.dat-s.gz` |
| CBF model file | `.cbf`, `.cbf.gz` |
| model file in COPT binary format | `.bin`, `.bin.gz` |
| basis file | `.bas` |
| solution file | `.sol` |
| IIS file | `.iis` |
| FeasRelax file | `.relax` |
| MIP initial solution file | `.mst` |
| parameter file | `.par` |
| parameter tuning file | `.tune` |
| branching-order file | `.ord` |

## 21.2 File I/O operations

By calling the relevant functions, users can input external model files to COPT for reading. At the same time, they can also save the output of the modeling and optimization results of COPT and output the files.

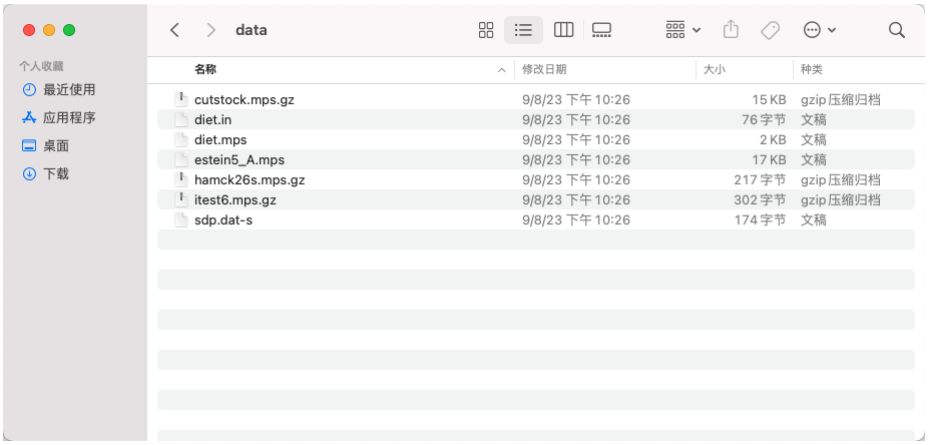
Let's take reading/writing the model in MPS format in the current directory as an example (similar operations are performed for other files). The implementation methods in different interfaces are as follows:

Table 21.2: Functions for input and output files

| API | Input | Output |
|---|---|---|
| COPT cmd | `read example.mps` | `write example.mps` |
| C | `COPT_ReadMps` | `COPT_WriteMps` |
| Python | `Model.read()` / `Model.readMps()` | `Model.write()` / `Model.writeMps()` |
| C++ | `Model::read()` / `Model::readMps()` | `Model::write()` / `Model::writeMps()` |
| C# | `Model.Read()` / `Model.ReadMps()` | `Model.Write()` / `Model.WriteMps()` |
| Java | `Model.read()` / `Model.readMps()` | `Model.write()` / `Model.writeMps()` |

## 21.3 Model file introduction

Users can find the model file examples that come with COPT in the `"examples/data"` directory of the installation package. Here we introduce the specific contents of two common model file formats: MPS and LP.




### MPS format

MPS is a universal model file standard format. Different types of optimization problems can be output and stored in mps format, which is widely used in optimization software.

The following is an example of a model file in MPS format:

```
NAME          COPTPROB
OBJSENSE
    MAX
ROWS
N  __OBJ___
L  R0000000
G  R0000001
COLUMNS
    x         __OBJ___  1.2
    x         R0000000  1.5
    x         R0000001  0.80000000000000004
    y         __OBJ___  1.8
    y         R0000000  1.2
    y         R0000001  0.59999999999999998
    z         __OBJ___  2.1000000000000001
    z         R0000000  1.8
```

(continues on next page)

(continued from previous page)

```
    z         R0000001  0.90000000000000002
RHS
    RHS       R0000000  2.6000000000000001
    RHS       R0000001  1.2
BOUNDS
LO BOUND     x         0.10000000000000001
UP BOUND     x         0.59999999999999998
LO BOUND     y         0.20000000000000001
UP BOUND     y         1.5
LO BOUND     z         0.29999999999999999
UP BOUND     z         2.7999999999999998
ENDATA
```

This MPS format example mainly includes several parts: NAME, OBJSENSE, ROWS, COLUMNS, RHS, and BOUNDS.

1. NAME: The name of the model
2. OBJSENSE: Optimization direction of the objective function
3. ROWS: Constraints and their directions in the model (L means <= constraints, G means >= constraints, N means no boundaries)
4. COLUMNS: Variables and their coefficients in the model
5. RHS: The value of the right-hand term of the constraint
6. BOUNDS: Bounds of variables (LO means lower bound, UP means upper bound, FR means no bounds)

**Notes**

1. In the ROWS section, the first line `__OBJ___` represents the objective function.
2. In the COLS part, the form `x __OBJ___ 1.2` indicates that the coefficient of variable x in the objective function is 1.2.
3. In MPS format, integer variables will be identified by the following fields:
   - First integer variable: `MARKER 'MARKER' 'INTORG'`
   - Last integer variable: `MARKER 'MARKER' 'INTEND'`

### LP format

The LP format is closer to the algebraic form. It is more readable than MPS, and can easily correspond to its original mathematical model.

The following is an example of a model file in LP format:

```
\Generated by Cardinal Operations

Maximize
1.2 x + 1.8 y + 2.1 z
Subject To
1.5 x + 1.2 y + 1.8 z <= 2.6
0.8 x + 0.6 y + 0.9 z >= 1.2
Bounds
0.1 <= x <= 0.6
0.2 <= y <= 1.5
0.3 <= z <= 2.8
END
```

This LP format mainly includes several parts: objective function (Maximize), constraints (Subject To), and variable scope (Bounds).

**Notes**

1. In some LP format file, we can see variable names in the form `x#1`, which marks x as the first variable. When the user does not specify a variable name, this is the name automatically generated when outputting the lp model file.
2. If there is a binary type in the variable, it will be identified by the `Binaries` field.

# Chapter 22

# FAQs

## 22.1 Installation and Licensing Configuration Related

- **Q:** What is the reason for the error `invalid username` when configuring the license?

  **A:** This error indicates that the username was incorrectly filled in when applying, and you can re-fill the form with the correct username to apply. For information on how to obtain username under different operating systems, please refer to the COPT application page, please remark in the application reason with **"The username is incorrectly filled in, reapply"** . we will issue new license for the correct username.

- **Q:** After downloading COPT, an antivirus software installed on the computer reports a virus and automatically isolates it.

  **A:** The COPT software downloaded from the COPT official download link is the official version, which has not been developed with any suspicious virus behavior. It can be determined that the anti-virus software is falsely reporting. Please temporarily close the anti-virus software before downloading it.

- **Q:** After validating the license (executing `copt_licgen -v` ), it reports an error: `Missing Files` or `Invalid Signature` .

  **A:** This type of error indicates that the license file configuration fails. Please refer to *Installation Instructions: Configuring License File* to check whether the steps for configuring the license file are correctly followed. Common reasons are as follows:

  1. The license file in the current working directory is not compatible with the version of COPT (for example: the license is version 4.0, while the COPT is version 5.0), please check the `VERSION` in `"license.dat"` to confirm whether the major version is the same, if not, please go to the COPT application page to re-apply, and we will issue you the latest license.
  2. In Windows system, if the COPT software is installed in the system disk (usually `C`) in a non-user directory (eg: the default installation path `"C:\Program Files\copt70"`) , you need to **open the command line window with administrator privileges** and execute the license acquisition command `copt_licgen` again.

- **Q:** I have already installed an old version of the COPT Python interface ( `coptpy` ), how do I upgrade to the new version?

  **A:** Please refer to *Python Interface Quick Start: Upgrade to the newer version* for detailed steps.

### 22.1.1 MacOS System

- **Q:** When calling `coptpy` on MacOS, an error is reported: `ImportError: from .coptpywrap import * symbol not found in flat namespace`.

  **A:** This error may occur before COPT 6.5.12, since the architecture of `coptpy` and Anaconda do not match. For example, `coptpy` is the M chip version (arm64), and Anaconda is x86 version, you could install Anaconda that supports arm64 architecture to solve this problem.

  However, from COPT 6.5.12, for MacOS systems, we provide universal package, so this problem could be resolved by upgrading COPT to the latest version.

- **Q:** When the license is configured in the MacOS system, the `copt_licgen` command is executed in the terminal, and an error is reported: `command not found: copt_licgen` .

  **A:** This error is because the relevant environment variables of COPT are not configured. In the MacOS system, you need to configure the environment variables after installing COPT. Please refer to *Installation Instructions: MacOS System* chapter to obtain detailed installation instructions.

- **Q:** When manually configuring environment variables, I copy content directly from the document to `.zshrc` file or `.bash_profile` file, causing the configuration to fail.

  **A:** Due to the document encoding problem, the above environment variables cannot be directly copied to the corresponding file, and the contents of the environment variables need to be manually entered.

### 22.1.2 Windows System

- **Q:** In Windows system, when executing `copt_licgen` to generate the license file, an error is reported that the license file cannot be written to the hard disk, and the error message is: `error opening file` .

  **A:** If the COPT software is installed in the system disk (usually `C`) in a non-user directory (eg: the default installation path `"C:\Program Files\copt80"`), y you need to **open the command line window with administrator privileges** and execute the license acquisition command `copt_licgen`, in order to normally write the license file to the `C` drive. Administrator privileges are not required to execute permission acquisition commands under user directories such as `"C:\Users\copt80"`.

- **Q:** In Windows system, when installing COPT Python interface via command `pip install coptpy`, an error is displayed: `could not find a version, no matching distribution` , what is the reason?

  **A:** Please do not use Python installed through Microsoft Store, it is recommended to download from Anaconda distribution or Python official distribution Download Python.

- **Q:** In Windows system, when installing COPT Python interface through COPT installation package ( `python setup.py install` ), an error `could not create build: access denied` is reported.

  **A:** If COPT is installed in the system disk (usually `C`) in a non-user directory (eg: the default installation path `"C:\Program Files\copt70"`), you need to first **Open the command line window with administrator privileges** and execute the command `python setup.py install`.

## 22.2 Modeling and Solving Functions Related

- **Q:** When creating a COPT solution environment, two lines of version information will be output. If I want to turn off this information, how should I do it?
- **A:** You can turn it off by setting `"nobanner"` to `"1"` in `EnvrConfig` before creating the solution environment. Taking the Python API as an example, the specific operations are as follows:

```
envconfig = coptpy.EnvrConfig()
envconfig.set("nobanner", "1")
env = coptpy.Envr(envconfig)
model = env.createModel()
```

- **Q:** How to deal with the situation where the model is infeasible?

  **A:** COPT provides functions to calculate IIS and feasible relaxation to analyze the reasons for model infeasibility: Computing IIS will obtain the minimum set of infeasible constraints and variables, and feasibility relaxation attempts to make the model feasible with minimal changes.

- **Q:** Are there default value ranges for variables created in COPT?

  **A:** Yes, the default lower bound of variables in COPT is 0, and the upper bound is `COPT.INFINITY`. Users can specify the lower bound of variables through the function argument *lb*, and the upper bound of variables through *ub*.

- **Q:** What is the reason for the error `ValueError: cannot create object arrays from iterator.` when adding matrix variables using the matrix modeling method of COPT?

  **A:** The matrix modeling function supported by COPT Python has a minimum version requirement, the minimum version requirement for `NumPy` is 1.23, and the minimum version requirement for `Python` is 3.8 . `NumPy` can be upgraded to the latest version by `pip install --upgrade numpy`.

- **Q:** When adding constraints using Python interface, if the modeling efficiency is slow, are there any ways to improve the modeling process?

  **A:** The Python interface of COPT supports building linear expression, quadratic expression and PSD expression in natural way. For linear and quadratic expression, it is recommended to use quicksum() to build expression objects. For linear and PSD expression, it is recommended to use psdquicksum() to build expression objects. Both of them implement inplace summation, which is much faster than standard plus operator.

## 22.3 GPU Usage Related

- **Q:** Are there any requirements for the CUDA library version when enabling COPT's GPU solver?

  **A:** COPT requires a minimum version of 12.0.1 for the CUDA library.

- **Q:** Are there any requirements for the GPU architecture when enabling GPU solver?

  **A:** The GPU architecture must be at least Maxwell or a more recent version. (Maxwell is a GPU architecture introduced by NVIDIA in 2014 as an upgrade to the earlier Kepler architecture.) Besides, COPT supports the Blackwell architecture for Windows and Linux platform.

- **Q:** What are the common error messages and possible reasons when the machine cannot use COPT GPU solver?

  **A:** Common error messages and possible reasons are as follows:

  1.Solving log indicates `"CUDA libraries could not be loaded: cuBLAS cuSPARSE"` which suggests that the listed necessary CUDA libraries are missing. You could try checking and setting the environment variable `"LD_LIBRARY_PATH"` to point to the directory where CUDA is installed. (Please follow the instructions provided after CUDA installation. Environment variables are automatically configured during installation on Windows systems. For Linux systems, manual

configuration of environment variables is typically required. The directory will be like: `"/usr/local/cuda/lib64"` ).

2.Solving error message `"Fail to solve problem"` is usually due to the lower version of CUDA Driver. Please upgrade the COPT Driver (for Linux systems: 525.60.13 or above; for Windows systems: 527.41 or above) to resolve this issue.

3.Solving error message `"sparse matrix format CUSPARSE_FORMAT_CSC is not supported"` is usually caused by the lower version of CUDA Toolkit (typically occurring between CUDA V11.2 and V11.6). Please upgrade CUDA to version 12.0.1 or above to resolve this issue.

- **Q:** On a client machine with multiple GPUs, when setting the parameter `GPUDevice` to use a specific GPU number, why does it still detect only GPU with number 0 during solving?

  **A:** Please check if the environment variable `"CUDA_VISIBLE_DEVICES"` has been manually set to specify the visible GPU devices for CUDA. Try not to set this environment variable so that COPT can detect all available GPUs on the current machine.

- **Q:** Why do I encounter errors when using COPT's GPU solver via Windows Subsystem for Linux (WSL) despite installing CUDA libraries that meet the version requirements (V12.0.1 or higher)?

  **A:** Please check if the CUDA Driver version meets the requirements. WSL usually skips the Driver installation when installing CUDA and directly uses the Driver already installed in Windows. Please manually upgrade the CUDA Driver version and then restart WSL to resolve this issue.

- **Q:** When solving with the GPU Barrier method, an error message `"GPU memory issue"` is reported. How should I address this?

  **A:** This error is typically caused by insufficient GPU memory. Please try setting the parameter `GPUMode=1`. If the issue persists, consider solving the model using the CPU (`GPUMode=0`).

- **Q:** When solving with the GPU Barrier method, a message says `"Performance may degenerate for GPUMode=2 on this problem, consider trying GPUMode=1 instead"`. What does this mean?

  **A:** This message indicates that the problem is still solvable, but using `GPUMode=2` (high-performance mode) may result in longer solving time. You are advised to set `GPUMode=1` to solve the problem in the standard mode.

## 22.4 CPU and Memory Binding Related

- **Q:** What is a NUMA node? How can I obtain the number of NUMA nodes in the operating system?

  **A:** NUMA (Non-Uniform Memory Access) is a memory architecture designed for multi-processor systems. A NUMA node is the basic unit in this architecture, consisting of a group of CPU cores and the local memory directly attached to them. If the current machine has $N$ physical CPUs, the return value of `env.getNumaNodeCount()` will be $N$.

- **Q:** How can I solve a model using a specified NUMA node (CPU)?

  **A:** If `env.getNumaNodeCnt()` returns a value greater than 1, the machine contains multiple NUMA nodes. Since cross-node memory access is typically more expensive than access within the same node, you may call `env.bindNumaCpu(numaNode=n)` to bind the solver to NUMA node $n$ in order to reduce cross-node access overhead. The `numaNode` argument starts from 0.

- **Q:** How can I bind memory to a specified NUMA node?

  **A:** Similar to CPU binding, you may call `env.bindNumaMem(numaNode=0)` to restrict memory allocation to the local memory of the specified NUMA node, thereby avoiding cross-node memory access.

- **Q:** How does `Envr.setCpuAffinity()` specify the CPU cores used for solving? How should the argument `hexMask` be interpreted?

**A:** `hexMask` is a CPU mask represented as a hexadecimal string and used to specify the set of CPU cores available to the current process. After converting the mask to binary, bits are interpreted from right to left, corresponding to CPU core indices 0, 1, 2, and so on. A bit set to 1 indicates that the corresponding core is enabled for solving. For example, on a machine with 32 CPU cores, the call `env.setCpuAffinity("F1")` uses the mask `"F1"`, which corresponds to the binary `1111 0001`. Thus, CPU cores 0 and 4–7 will participate in solving.

- **Q:** How can I check which CPU cores are used by the current solving process?

  **A:** You may call `env.getCpuAffinity()` to retrieve the CPU affinity setting of the current solving process. It returns a list of core indices, such as `[0, 4, 5, 6, 7]`. If no CPU affinity is set, an empty list is returned, meaning the operating system determines the CPU scheduling.

- **Q:** What is the effective scope of CPU affinity set via `Envr.setCpuAffinity()`?

  **A:** This function sets the CPU affinity for the current solving process. Once set, the affinity remains effective for the entire lifetime of the process and will not revert when `env.close()` is called.

# Chapter 23

# C API Reference

The **Cardinal Optimizer** provides a C API library for advanced usage. This section documents all the COPT constants, API functions, parameters and attributes listed in `copt.h`.

## 23.1 Constants

There are three types of constants.

1. Constructing models, such as optimization directions, constraint senses or variable types.
2. Accessing solution results, such as API return code, basis status and LP status.
3. Monitoring optimization progress, such as callback context.

### 23.1.1 Optimization directions

For different optimization scenarios, it may be required to either maximize or minimize the objective function. There are two optimization directions:

- `COPT_MINIMIZE`

  For minimizing the objective function.

- `COPT_MAXIMIZE`

  For maximizing the objective function.

The optimization direction is automatically set when reading in a problem from file. It can also be set explicitly using `COPT_SetObjSense`.

### 23.1.2 Infinity

In COPT, the infinite bound is represented by a large value, whose default value is also available as a constant:

- `COPT_INFINITY`

  The default value (`1e30`) of the infinite bound.

### 23.1.3 Undefined Value

In COPT, the undefined value is represented by another large value. For example, the default solution value of MIP start is set to a constant:

- `COPT_UNDEFINED`

  Undefined value(`1e40`).

### 23.1.4 Constraint senses

**NOTE:** Using constraint senses is supported by COPT but not recommended. We recommend defining constraints using explicit lower and upper bounds.

Traditionally, for optimization models, constraints are defined using **senses**. The most common constraint senses are:

- `COPT_LESS_EQUAL`

  For constraint in the form of $g(x) \leq b$

- `COPT_GREATER_EQUAL`

  For constraint in the form of $g(x) \geq b$

- `COPT_EQUAL`

  For constraint in the form of $g(x) = b$

In additional, there are two less used constraint senses:

- `COPT_FREE`

  For unconstrained expression

- `COPT_RANGE`

  For constraints with both lower and upper bounds in the form of $l \leq g(x) \leq u$.

  Please refer to documentation of `COPT_LoadProb` regarding how to use `COPT_RANGE` to define a constraints with both lower and upper bounds.

### 23.1.5 Variable types

Variable types are used for defining whether a variable is continuous or integral.

- `COPT_CONTINUOUS`

  Non-integer continuous variables

- `COPT_BINARY`

  Binary variables

- `COPT_INTEGER`

  Integer variables

### 23.1.6 SOS-constraint types

SOS constraint (Special Ordered Set) is a kind of special constraint that places restrictions on the values that a set of variables can take.

COPT currently support two types of SOS constraints, one is SOS1 constraint, where at most one variable in the constraint is allowed to take a non-zero value, the other is SOS2 constraint, where at most two variables in the constraint are allowed to take non-zero value, and those non-zero variables must be contiguous. Variables in SOS constraints are allowed to be continuous, binary and integer.

- `COPT_SOS_TYPE1`

  SOS1 constraint

- `COPT_SOS_TYPE2`

  SOS2 constraint

### 23.1.7 Indicator constraint

Indicator constraint is a kind of logical constraints in COPT, used to describe the relationship between the value of the binary variable $y$ and whether the linear constraint $a^T x \leq b$ is satisfied. Currently, COPT supports three types of indicator constraints:

- `COPT_INDICATOR_IF`

  If-Then:

  If $y = f$ , then the linear constraint $a^T x \leq b$ is satisfied.

  If $y \neq f$ , then the linear constraint $a^T x \leq b$ is invalid (may be violated).

$$\begin{aligned} & y = f \rightarrow a^T x \leq b \\ & f \in \{0, 1\} \end{aligned} \tag{23.1}$$

- `COPT_INDICATOR_ONLYIF`

  Only-If:

  If the linear constraint $a^T x \leq b$ is satisfied, then $y = f$ .

  If the linear constraint $a^T x \leq b$ is not satisfied, then $y$ can be 0 or 1.

$$\begin{aligned} & a^T x \leq b \rightarrow y = f \\ & f \in \{0, 1\} \end{aligned} \tag{23.2}$$

- `COPT_INDICATOR_IFANDONLYIF`

  If-and-Only-If:

  The linear constraint $a^T x \leq b$ and $y = f$ must be satisfied simultaneously or not satisfied simultaneously.

$$\begin{aligned} & a^T x \leq b \leftrightarrow y = f \\ & f \in \{0, 1\} \end{aligned} \tag{23.3}$$

### 23.1.8 SOC constraint type

The Second-Order-Cone (SOC) constraint is a special type of quadratic constraints. COPT supports two types of SOC constraints:

- `COPT_CONE_QUAD` : Regular Second-Order-Cone

$$Q^n = \left\{ x \in \mathbb{R}^n \;\middle|\; x_0 \geq \sqrt{\sum_{i=1}^{n-1} x_i^2}, x_0 \geq 0 \right\} \tag{23.4}$$

- `COPT_CONE_RQUAD` : Rotated Second-Order-Cone

$$Q_r^n = \left\{ x \in \mathbb{R}^n \;\middle|\; 2x_0x_1 \geq \sum_{i=2}^{n-1} x_i^2, x_0 \geq 0, x_1 \geq 0 \right\} \tag{23.5}$$

### 23.1.9 Exponential cone constraint type

COPT supports two types of exponential cone constraints:

- `COPT_EXPCONE_PRIMAL` : Primal exponential cone

$$\mathrm{cl}(S_1) = S_1 \cup S_2 \tag{23.6}$$

$$\begin{aligned} S_1 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid s > 0,\ t \geq s \exp\left(\frac{r}{s}\right) \right\}, \\ S_2 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid s = 0,\ t \geq 0,\ r \leq 0 \right\} \end{aligned} \tag{23.7}$$

- `COPT_EXPCONE_DUAL` : Dual exponential cone

$$\mathrm{cl}(S_1) = S_1 \cup S_2 \tag{23.8}$$

$$\begin{aligned} S_1 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid r < 0,\ t \geq -r \exp\left(\frac{s}{r} - 1\right) \right\}, \\ S_2 &= \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3 \mid r = 0,\ t \geq 0,\ s \geq 0 \right\} \end{aligned} \tag{23.9}$$

### 23.1.10 Quadratic objective function

Besides linear objective function, COPT also supports general convex quadratic objective function.

The mathematical form is:

$$x^T Q x + c^T x \tag{23.10}$$

Where, $x$ is an array of variables, $Q$ is the quadratic part of the quadratic objective function and $c$ is the linear part.

### 23.1.11 Quadratic constraint

Besides the special type of quadratic constraint, Second-Order-Cone (SOC) constraint, COPT also supports general convex quadratic constraint.

The mathematical form is:

$$x^T Q x + q^T x \leq b \tag{23.11}$$

Where, $x$ is an array of variables, $Q$ is the quadratic part of the quadratic constraint and $c$ is the linear part.

### 23.1.12 Basis status

For an LP problem with $n$ variables and $m$ constraints, the constraints are treated as slack variables *internally*, resulting in $n + m$ variables. When solving an LP problem using the simplex method, the simplex method fixes $n$ variables at one of their bounds, and then computes solutions for the other $m$ variables. The $m$ variables with computed solution are called *basic* variables, and the other $n$ variables are called *non-basic* variables. The simplex progress and its final solution can be defined using the basis status of all the variables and constraints.

The basis status supported by COPT are:

- `COPT_BASIS_LOWER`

  The variable is non-basic at its lower bound.

- `COPT_BASIS_BASIC`

  The variable is basic.

- `COPT_BASIS_UPPER`

  The variable is non-basic at its upper bound.

- `COPT_BASIS_SUPERBASIC`

  The variable is non-basic but not any of its bounds.

- `COPT_BASIS_FIXED`

  The variable is non-basic and fixed at its bound.

### 23.1.13 Optimization status

The solution status of an optimization problem is called optimization status, which can be obtained using integer attribute `COPT_INTATTR_STATUS`.

Possible optimization status values are:

- `COPT_STATUS_UNSTARTED`

  The optimization is not started yet.

- `COPT_STATUS_OPTIMAL`

  The problem is solved to optimality.

- `COPT_STATUS_INFEASIBLE`

  The problem is infeasible.

- `COPT_STATUS_UNBOUNDED`

  The problem is unbounded.

- `COPT_STATUS_INF_OR_UNB`

The problem is infeasible or unbounded.

- `COPT_STATUS_NUMERICAL`

  Numerical trouble encountered.

- `COPT_STATUS_NODELIMIT`

  The optimization is stopped because of node limit.

- `COPT_STATUS_IMPRECISE`

  The solution is imprecise.

- `COPT_STATUS_TIMEOUT`

  The optimization is stopped because of time limit.

- `COPT_STATUS_UNFINISHED`

  The optimization is stopped but the solver cannot provide a solution because of numerical difficulties.

- `COPT_STATUS_INTERRUPTED`

  The optimization is stopped by user interrupt.

- `COPT_STATUS_ITERLIMIT`

  The optimization is stopped because of iteration limit.

- `COPT_STATUS_LOCAL_OPTIMAL`

  A locally optimal solution satisfying KKT conditions is found for the non-convex problem.

- `COPT_STATUS_LOCAL_INFEASIBLE`

  The non-convex problem is locally infeasible.

### 23.1.14 LP solution status

The solution status of an LP problem is called LP status, which can be obtained using integer attribute `COPT_INTATTR_LPSTATUS`.

Possible LP status values are:

- `COPT_LPSTATUS_UNSTARTED`

  The LP optimization is not started yet.

- `COPT_LPSTATUS_OPTIMAL`

  The LP problem is solved to optimality.

- `COPT_LPSTATUS_INFEASIBLE`

  The LP problem is infeasible.

- `COPT_LPSTATUS_UNBOUNDED`

  The LP problem is unbounded.

- `COPT_LPSTATUS_NUMERICAL`

  Numerical trouble encountered.

- `COPT_LPSTATUS_TIMEOUT`

  The LP optimization is stopped because of time limit.

- `COPT_LPSTATUS_UNFINISHED`

  The LP optimization is stopped but the solver cannot provide a solution because of numerical difficulties.

- COPT_LPSTATUS_IMPRECISE

  The solution is imprecise.

- COPT_LPSTATUS_INTERRUPTED

  The LP optimization is stopped by user interrupt.

- COPT_LPSTATUS_ITERLIMIT

  The LP optimization is stopped because of iteration limit.

### 23.1.15 MIP solution status

The solution status of an MIP problem is called MIP status, which can be obtained using integer attribute COPT_INTATTR_MIPSTATUS.

Possible MIP status values are:

- COPT_MIPSTATUS_UNSTARTED

  The MIP optimization is not started yet.

- COPT_MIPSTATUS_OPTIMAL

  The MIP problem is solved to optimality.

- COPT_MIPSTATUS_INFEASIBLE

  The MIP problem is infeasible.

- COPT_MIPSTATUS_UNBOUNDED

  The MIP problem is unbounded.

- COPT_MIPSTATUS_INF_OR_UNB

  The MIP problem is infeasible or unbounded.

- COPT_MIPSTATUS_NODELIMIT

  The MIP optimization is stopped because of node limit.

- COPT_MIPSTATUS_TIMEOUT

  The MIP optimization is stopped because of time limit.

- COPT_MIPSTATUS_UNFINISHED

  The MIP optimization is stopped but the solver cannot provide a solution because of numerical difficulties.

- COPT_MIPSTATUS_INTERRUPTED

  The MIP optimization is stopped by user interrupt.

**Note**

COPT_INTATTR_LPSTATUS and COPT_INTATTR_MIPSTATUS are deprecated. It is recommended to use COPT_INTATTR_STATUS instead, which provides a unified interface for querying the solution status across all supported problem types.

### 23.1.16 Callback context

- `COPT_CBCONTEXT_INCUMBENT`

  Invokes the callback after a new incumbent was found.

- `COPT_CBCONTEXT_MIPNODE`

  Invokes the callback after a MIP node was processed.

- `COPT_CBCONTEXT_MIPRELAX`

  Invokes the callback when an LP-relaxation was solved.

- `COPT_CBCONTEXT_MIPSOL`

  Invokes the callback when a new MIP candidate solution is found.

### 23.1.17 Nonlinear Expression Operators

- `COPT_NL_PLUS`

  Addition operator.

- `COPT_NL_MINUS`

  Subtraction operator.

- `COPT_NL_MULT`

  Multiplication operator.

- `COPT_NL_DIV`

  Division operator.

- `COPT_NL_POW`

  Power operator.

- `COPT_NL_SQRT`

  Square root operator.

- `COPT_NL_EXP`

  Exponential function operator.

- `COPT_NL_LOG`

  Natural logarithm operator.

- `COPT_NL_LOG10`

  Base-10 logarithm operator.

- `COPT_NL_NEG`

  Unary negation operator.

- `COPT_NL_ABS`

  Absolute value operator.

- `COPT_NL_FLOOR`

  Floor function operator.

- `COPT_NL_CEIL`

  Ceiling function operator.

- `COPT_NL_SIN`

Sine function operator.

- `COPT_NL_COS`

  Cosine function operator.

- `COPT_NL_TAN`

  Tangent function operator.

- `COPT_NL_SINH`

  Hyperbolic sine function operator.

- `COPT_NL_COSH`

  Hyperbolic cosine function operator.

- `COPT_NL_TANH`

  Hyperbolic tangent function operator.

- `COPT_NL_ASIN`

  Inverse sine (arcsin) function operator.

- `COPT_NL_ACOS`

  Inverse cosine (arccos) function operator.

- `COPT_NL_ATAN`

  Inverse tangent (arctan) function operator.

- `COPT_NL_ASINH`

  Inverse hyperbolic sine (arsinh) function operator.

- `COPT_NL_ACOSH`

  Inverse hyperbolic cosine (arcosh) function operator.

- `COPT_NL_ATANH`

  Inverse hyperbolic tangent (artanh) function operator.

- `COPT_NL_ATAN2`

  Two-argument inverse tangent function operator.

- `COPT_NL_SUM`

  Summation operator.

- `COPT_NL_GET`

  Constant retrieval operator.

### 23.1.18 API function return code

When an API function finishes, it returns an integer **return code**, which indicates whether the API call was finished okay or failed. In case of failure, it specifies the reason of the failure.

Possible COPT API function return codes are:

- `COPT_RETCODE_OK`

  The API call finished successfully.

- `COPT_RETCODE_MEMORY`

  The API call failed because of memory allocation failure.

- `COPT_RETCODE_FILE`

The API call failed because of file input or output failure.

- `COPT_RETCODE_INVALID`

  The API call failed because of invalid data.

- `COPT_RETCODE_LICENSE`

  The API call failed because of license validation failure. In this case, further information can be obtained by calling `COPT_GetLicenseMsg`.

- `COPT_RETCODE_INTERNAL`

  The API call failed because of internal error.

- `COPT_RETCODE_THREAD`

  The API call failed because of thread error.

- `COPT_RETCODE_SERVER`

  The API call failed because of remote server error.

- `COPT_RETCODE_NONCONVEX`

  The API call failed because of problem is nonconvex.

- `COPT_RETCODE_MEMORY_GPU`

  The API call failed because of GPU memory allocation failure.

### 23.1.19 Client configuration

For floating and cluster clients, users are allowed to set client configuration parameters, currently available settings are:

- `COPT_CLIENT_CLUSTER`

  IP address of cluster server.

- `COPT_CLIENT_FLOATING`

  IP address of token server.

- `COPT_CLIENT_PASSWORD`

  Password of cluster server.

- `COPT_CLIENT_PORT`

  Connection port of token server.

- `COPT_CLIENT_WAITTIME`

  Wait time of client.

### 23.1.20 Other constants

- `COPT_BUFFSIZE`

  Defines the recommended buffer size when obtaining a C-style string message from COPT library. This can be used with, for example, `COPT_GetBanner`, `COPT_GetRetcodeMsg` etc.

## 23.2 Attributes

### 23.2.1 Problem attribute

- `COPT_INTATTR_COLS` or `"Cols"`

  Integer attribute.

  Number of variables (columns) in the problem.

- `COPT_INTATTR_BINS` or `"Bins"`

  Integer attribute.

  Number of binary variables.

- `COPT_INTATTR_INTS` or `"Ints"`

  Integer attribute.

  Number of integer variables.

- `COPT_INTATTR_PSDCOLS` or `"PSDCols"`

  Integer attribute.

  Number of PSD variables in the problem.

- `COPT_INTATTR_ROWS` or `"Rows"`

  Integer attribute.

  Number of constraints (rows) in the problem.

- `COPT_INTATTR_INDICATORS` or `"Indicators"`

  Integer attribute.

  Number of indicator constraints.

- `COPT_INTATTR_SOSS` or `"Soss"`

  Integer attribute.

  Number of SOS constraints.

- `COPT_INTATTR_CONES` or `"Cones"`

  Integer attribute.

  Number of Second-Order-Cone constraints.

- `COPT_INTATTR_EXPCONES` or `"ExpCones"`

  Integer attribute.

  Number of exponential cone constraints.

- `COPT_INTATTR_AFFINECONES` or `"AffineCones"`

  Integer attribute.

  Number of affine cone constraints.

- `COPT_INTATTR_QCONSTRS` or `"QConstrs"`

  Integer attribute.

  Number of quadratic constraints.

- `COPT_INTATTR_PSDCONSTRS` or `"PSDConstrs"`

  Integer attribute.

  Number of PSD constraints.

- `COPT_INTATTR_LMICONSTRS` or `"LMIConstrs"`

  Integer attribute.

  Number of LMI constraints.

- `COPT_INTATTR_NLCONSTRS` or `"NLConstrs"`

  Integer attribute.

  The number of nonlinear expression constraints in the model.

- `COPT_INTATTR_ELEMS` or `"Elems"`

  Integer attribute.

  Number of non-zero elements in the coefficient matrix.

- `COPT_INTATTR_QELEMS` or `"QElems"`

  Integer attribute.

  Number of non-zero quadratic elements in the quadratic objective function.

- `COPT_INTATTR_PSDELEMS` or `"PSDElems"`

  Integer attribute.

  Number of PSD terms in objective function.

- `COPT_INTATTR_NLELEMS` or `"NLElems"`

  Integer attribute.

  The number of nonlinear expression terms in the objective of the model.

- `COPT_INTATTR_SYMMATS` or `"SymMats"`

  Integer attribute.

  Number of symmetric matrices in the problem.

- `COPT_INTATTR_MULTIOBJS` or `"MultiObjs"`

  Integer attribute.

  The number of objectives in a multi-objective model.

- `COPT_INTATTR_OBJSENSE` or `"ObjSense"`

  Integer attribute.

  The optimization direction.

- `COPT_DBLATTR_OBJCONST` or `"ObjConst"`

  Double attribute.

  The constant part of the objective function.

- `COPT_INTATTR_HASQOBJ` or `"HasQObj"`

  Integer attribute.

  Whether the problem has quadratic terms in its objective.

- `COPT_INTATTR_HASPSDOBJ` or `"HasPSDObj"`

  Integer attribute.

  Whether the problem has PSD terms in its objective.

- `COPT_INTATTR_HASNLOBJ` or `"HasNLObj"`

  Integer attribute.

  Whether the problem has nonlinear terms in its objective.

- `COPT_INTATTR_ISMIP` or `"IsMIP"`

  Integer attribute.

  Whether the problem is a MIP.

- `COPT_DBLATTR_MINELEM` or `"MinElem"`

  Double attribute.

  Minimum absolute value of the non-zero elements of the coefficient matrix in the linear constraints.

- `COPT_DBLATTR_MAXELEM` or `"MaxElem"`

  Double attribute.

  Maximum absolute value of the non-zero elements of the coefficient matrix in the linear constraints.

- `COPT_DBLATTR_MINBOUND` or `"MinBound"`

  Double attribute.

  Minimum absolute value of the non-zero variable bounds.

  Bounds equal to zero and infinite bounds are not taken into account. A bound is treated as infinite when its absolute value reaches the infinite-bound value used by COPT, which is 1e30 by default.

- `COPT_DBLATTR_MAXBOUND` or `"MaxBound"`

  Double attribute.

  Maximum absolute value of the non-zero variable bounds.

  Please refer to `COPT_DBLATTR_MINBOUND` for details.

- `COPT_DBLATTR_MINRHS` or `"MinRHS"`

  Double attribute.

  Minimum absolute value of the non-zero bounds (right-hand sides) of the linear constraints.

  In COPT, a linear constraint is defined by a lower and an upper bound rather than by a single right-hand side, and both bounds are taken into account. A ranged constraint therefore contributes two values, while a constraint with a single sense contributes only its finite bound. Bounds equal to zero, and infinite bounds, are not taken into account.

- `COPT_DBLATTR_MAXRHS` or `"MaxRHS"`

  Double attribute.

  Maximum absolute value of the non-zero bounds (right-hand sides) of the linear constraints.

  Please refer to `COPT_DBLATTR_MINRHS` for details.

- `COPT_DBLATTR_MINCOST` or `"MinCost"`

  Double attribute.

  Minimum absolute value of the non-zero linear objective costs.

  Objective costs equal to zero are not taken into account.

- `COPT_DBLATTR_MAXCOST` or `"MaxCost"`

  Double attribute.

  Maximum absolute value of the non-zero linear objective costs.

  Please refer to `COPT_DBLATTR_MINCOST` for details.

- `COPT_DBLATTR_MINQELEM` or `"MinQElem"`

  Double attribute.

  Minimum absolute value of the non-zero quadratic elements of the quadratic constraints.

  This attribute describes the quadratic constraints. For the quadratic objective function, whose elements are counted by `COPT_INTATTR_QELEMS`, see `COPT_DBLATTR_MINQCOST` and `COPT_DBLATTR_MAXQCOST`.

- `COPT_DBLATTR_MAXQELEM` or `"MaxQElem"`

  Double attribute.

  Maximum absolute value of the non-zero quadratic elements of the quadratic constraints.

  Please refer to `COPT_DBLATTR_MINQELEM` for details.

- `COPT_DBLATTR_MINQLELEM` or `"MinQLElem"`

  Double attribute.

  Minimum absolute value of the non-zero linear elements of the quadratic constraints.

- `COPT_DBLATTR_MAXQLELEM` or `"MaxQLElem"`

  Double attribute.

  Maximum absolute value of the non-zero linear elements of the quadratic constraints.

- `COPT_DBLATTR_MINQRHS` or `"MinQRHS"`

  Double attribute.

  Minimum absolute value of the non-zero right-hand sides of the quadratic constraints.

  Unlike a linear constraint, a quadratic constraint is defined by a single right-hand side together with a sense. Right-hand sides equal to zero, and infinite right-hand sides, are not taken into account.

- `COPT_DBLATTR_MAXQRHS` or `"MaxQRHS"`

  Double attribute.

  Maximum absolute value of the non-zero right-hand sides of the quadratic constraints.

  Please refer to `COPT_DBLATTR_MINQRHS` for details.

- `COPT_DBLATTR_MINQCOST` or `"MinQCost"`

  Double attribute.

  Minimum absolute value of the non-zero quadratic objective costs.

  Coefficients are reported as supplied to the model, without applying any additional factor.

- `COPT_DBLATTR_MAXQCOST` or `"MaxQCost"`

  Double attribute.

  Maximum absolute value of the non-zero quadratic objective costs.

  Please refer to `COPT_DBLATTR_MINQCOST` for details.

**Note**

The attributes from `COPT_DBLATTR_MINELEM` to `COPT_DBLATTR_MAXQCOST` describe the problem as it was last loaded into the solver. Because model modifications are buffered, they return 0.0 before the model is first loaded, and their previous values after any later modification. Load the model before querying them, for example by solving or updating it.

A value of 0.0 is also returned when the model contains no elements of the corresponding kind: all quadratic attributes do so for a model with neither quadratic constraints nor a quadratic objective function. Use `COPT_INTATTR_QCONSTRS` and `COPT_INTATTR_HASQOBJ` to tell the two cases apart.

These attributes are retrieved from the original problem, and will therefore generally differ from the ranges of coefficients printed in the solver log, which are those of the problem actually passed to the solver, after presolving, scaling and reformulation.

### 23.2.2 Solution attribute

- `COPT_INTATTR_STATUS` or `"Status"`

  Integer attribute.

  Optimization status. Please refer to *Constants: Optimization status* for possible values.

- `COPT_INTATTR_LPSTATUS` or `"LpStatus"`

  Integer attribute.

  The LP status. Please refer to *Constants: LP solution status* for possible values.

  Deprecated. Use `COPT_INTATTR_STATUS` instead.

- `COPT_INTATTR_MIPSTATUS` or `"MipStatus"`

  Integer attribute.

  The MIP status. Please refer to *Constants: MIP solution status* for possible values.

  Deprecated. Use `COPT_INTATTR_STATUS` instead.

- `COPT_INTATTR_HASSOL` or `"HasSol"`

  Integer attribute.

  Whether a solution is available.

- `COPT_INTATTR_HASLPSOL` or `"HasLpSol"`

  Integer attribute.

  Whether LP solution is available.

  Deprecated. Use `COPT_INTATTR_HASSOL` instead.

- `COPT_INTATTR_HASMIPSOL` or `"HasMipSol"`

  Integer attribute.

  Whether MIP solution is available.

  Deprecated. Use `COPT_INTATTR_HASSOL` instead.

- `COPT_INTATTR_HASBASIS` or `"HasBasis"`

  Integer attribute.

  Whether LP basis is available.

- `COPT_INTATTR_HASSENSITIVITY` or `"HasSensitivity"`

  Integer attribute.

  Whether sensitivity analysis results are available for LP problem.

- `COPT_INTATTR_HASDUALFARKAS` or `"HasDualFarkas"`

  Integer attribute.

  Whether the dual Farkas of an infeasible LP problem is available.

- `COPT_INTATTR_HASPRIMALRAY` or `"HasPrimalRay"`

  Integer attribute.

  Whether the primal ray of an unbounded LP problem is available.

- `COPT_INTATTR_HASFEASRELAXSOL` or `"HasFeasRelaxSol"`

  Integer attribute.

  Whether feasibility LP-relaxation solution is available.

- `COPT_INTATTR_HASIIS` or `"HasIIS"`

  Integer attribute.

  Whether IIS is available.

- `COPT_INTATTR_ISMINIIS` or `"IsMinIIS"`

  Integer attribute.

  Whether the computed IIS is minimal.

- `COPT_INTATTR_IISCOLS` or `"IISCols"`

  Integer attribute.

  Number of bounds of columns in IIS.

- `COPT_INTATTR_IISROWS` or `"IISRows"`

  Integer attribute.

  Number of rows in IIS.

- `COPT_INTATTR_IISSOSS` or `"IISSOSs"`

  Integer attribute.

  Number of SOS constraints in IIS.

- `COPT_INTATTR_IISINDICATORS` or `"IISIndicators"`

  Integer attribute.

  Number of indicator constraints in IIS.

- `COPT_INTATTR_SIMPLEXITER` or `"SimplexIter"`

  Integer attribute.

  Number of simplex iterations performed.

- `COPT_INTATTR_BARRIERITER` or `"BarrierIter"`

  Integer attribute.

  Number of barrier iterations performed.

- `COPT_INTATTR_PDLPITER` or `"PDLPIter"`

  Integer attribute.

  Number of first-order method (PDLP) iterations performed.

- `COPT_INTATTR_NODECNT` or `"NodeCnt"`

  Integer attribute.

  Number of explored nodes.

- `COPT_INTATTR_POOLSOLS` or `"PoolSols"`

  Integer attribute.

  Number of solutions in solution pool.

- `COPT_INTATTR_TUNERESULTS` or `"TuneResults"`

  Integer attribute.

  Number of parameter tuning results.

- `COPT_DBLATTR_OBJVAL` or `"ObjVal"`

  Double attribute.

  Current best objective value. When `COPT_INTATTR_STATUS` is `COPT_STATUS_OPTIMAL`, this is the optimal objective value.

- `COPT_DBLATTR_OBJBOUND` or `"ObjBound"`

  Double attribute.

  Current best bound on the optimal objective value.

- `COPT_DBLATTR_LPOBJVAL` or `"LpObjVal"`

  Double attribute.

  The LP objective value.

  Deprecated. Use `COPT_DBLATTR_OBJVAL` instead.

- `COPT_DBLATTR_BESTOBJ` or `"BestObj"`

  Double attribute.

  Best integer objective value for MIP.

  Deprecated. Use `COPT_DBLATTR_OBJVAL` instead.

- `COPT_DBLATTR_BESTBND` or `"BestBnd"`

  Double attribute.

  Best bound for MIP.

  Deprecated. Use `COPT_DBLATTR_OBJBOUND` instead.

- `COPT_DBLATTR_BESTGAP` or `"BestGap"`

  Double attribute.

  Best relative gap for MIP.

- `COPT_DBLATTR_FEASRELAXOBJ` or `"FeasRelaxObj"`

  Double attribute.

  Feasibility relaxation objective value.

- `COPT_DBLATTR_SOLVINGTIME` or `"SolvingTime"`

  Double attribute.

  The time spent for the optimization (in seconds).

- `COPT_DBLATTR_MEMPEAK` or `"MemPeak"`

  Double attribute.

  Peak memory usage (MB) for the optimization.

**Note**

It is recommended to use `COPT_INTATTR_STATUS`, `COPT_INTATTR_HASSOL`, `COPT_DBLATTR_OBJVAL` and `COPT_DBLATTR_OBJBOUND` instead of the attributes marked as deprecated above, as they provide a unified interface for querying the solution status, checking solution availability, and retrieving the

optimal objective value across all supported problem types, including linear programming, mixed-integer programming, and convex/non-convex optimization.

## 23.3 Information

### 23.3.1 Problem information

- `COPT_DBLINFO_OBJ` or `"Obj"`

  Double information.

  Objective cost of columns.

- `COPT_DBLINFO_LB` or `"LB"`

  Double information.

  Lower bounds of columns or rows.

- `COPT_DBLINFO_UB` or `"UB"`

  Double information.

  Upper bounds of columns or rows.

### 23.3.2 Solution and sensitivity analysis information

- `COPT_DBLINFO_VALUE` or `"Value"`

  Double information.

  Solution of columns.

- `COPT_DBLINFO_SLACK` or `"Slack"`

  Double information.

  Solution of slack variables, also known as activities of constraints. Only available for LP problem.

- `COPT_DBLINFO_DUAL` or `"Dual"`

  Double information.

  Solution of dual variables. Only available for LP problem.

- `COPT_DBLINFO_REDCOST` or `"RedCost"`

  Double information.

  Reduced cost of columns. Only available for LP problem.

- `COPT_DBLINFO_SAOBJLOW` or `"SAObjLow"`

  Double information.

  Sensitivity analysis information of the objective coefficient.

  Indicates the minimum value to which the objective coefficient of a variable can be reduced while keeping the current basis optimal.

- `COPT_DBLINFO_SAOBJUP` or `"SAObjUp"`

Double information.

Sensitivity analysis information of the objective coefficient.

Indicates the maximum value to which the objective coefficient of a variable can be increased while keeping the current basis optimal.

- `COPT_DBLINFO_SALBLOW` or `"SALBLow"`

  Double information.

  Sensitivity analysis information of the lower bound of the variable/constraint.

  Indicates the minimum value to which the lower bound of the variable/constraint can be reduced while keeping the current basis optimal.

- `COPT_DBLINFO_SALBUP` or `"SALBUp"`

  Double information.

  Sensitivity analysis information of the lower bound of the variable/constraint.

  Indicates the maximum value to which the lower bound of the variable/constraint can be increased while keeping the current basis optimal.

- `COPT_DBLINFO_SAUBLOW` or `"SAUBLow"`

  Double information.

  Sensitivity analysis information of the upper bound of the variable/constraint.

  Indicates the minimum value to which the upper bound of the variable/constraint can be reduced while keeping the current basis optimal.

- `COPT_DBLINFO_SAUBUP` or `"SAUBUp"`

  Double information.

  Sensitivity analysis information of the upper bound of the variable/constraint.

  Indicates the maximum value to which the upper bound of the variable/constraint can be increased while keeping the current basis optimal.

### 23.3.3 Dual Farkas and primal ray

Advanced topic. When an LP is infeasible or unbounded, the solver can return the dual Farkas or primal ray to prove it.

- `COPT_DBLINFO_DUALFARKAS` or `"DualFarkas"`

  Double information.

  The dual Farkas for constraints of an infeasible LP problem. Please enable the parameter `"ReqFarkasRay"` to ensure that the dual Farkas is available when the LP is infeasible.

  Without loss of generality, the concept of the dual Farkas can be conveniently demonstrated using an LP problem with general variable bounds and equality constraints: $Ax = 0$ and $l \leq x \leq u$. When the LP is infeasible, a dual Farkas vector $y$ can prove that the system has conflict that $\max y^T Ax < y^T b = 0$. Computing $\max y^T Ax$: with the vector $\hat{a} = y^T A$, choosing variable bound $x_i = l_i$ when $\hat{a}_i < 0$ and $x_i = u_i$ when $\hat{a}_i > 0$ gives the maximal possible value of $y^T Ax$ for any $x$ within their bounds.

  Some application relies on the alternate conflict $\min \bar{y}^T Ax > \bar{y}^T b = 0$. This can be achieved by negating the dual Farkas, i.e. $\bar{y} = -y$ returned by the solver.

  In very rare cases, the solver may fail to return a valid dual Farkas. For example when the LP problem slightly infeasible by tiny amount, which We recommend to study and to repair the infeasibility using FeasRelax instead.

- `COPT_DBLINFO_PRIMALRAY` or `"PrimalRay"`

Double information.

The primal ray for variables of an unbounded LP problem. Please enable the parameter `"ReqFarkasRay"` to ensure that the primal ray is available when an LP is unbounded.

For a minimization LP problem in the standard form: $\min c^T x, Ax = b$ and $x \geq 0$, a primal ray vector $r$ satisfies that $r \geq 0, Ar = 0$ and $c^T r < 0$.

### 23.3.4 Feasibility relaxation information

- `COPT_DBLINFO_RELAXLB` or `"RelaxLB"`

  Double information.

  Feasibility relaxation values for lower bounds of columns or rows.

- `COPT_DBLINFO_RELAXUB` or `"RelaxUB"`

  Double information.

  Feasibility relaxation values for upper bounds of columns or rows.

- `COPT_DBLINFO_RELAXVALUE` or `"RelaxValue"`

  Double information.

  Solutions for the original model variables (columns) in the feasibility relaxation model.

## 23.4 Callback information

- `COPT_CBINFO_BESTOBJ` or `"BestObj"`

  Double information.

  Current best objective.

- `COPT_CBINFO_BESTBND` or `"BestBnd"`

  Double information.

  Current best objective bound.

- `COPT_CBINFO_HASINCUMBENT` or `"HasIncumbent"`

  Integer information.

  Whether an incumbent is available.

- `COPT_CBINFO_INCUMBENT` or `"Incumbent"`

  Double information.

  Current best feasible solution.

- `COPT_CBINFO_MIPCANDIDATE` or `"MipCandidate"`

  Double information.

  Current feasible solution candidate.

- `COPT_CBINFO_MIPCANDOBJ` or `"MipCandObj"`

  Double information.

  Objective value for current feasible solution candidate.

- `COPT_CBINFO_RELAXSOLUTION` or `"RelaxSolution"`

  Double information.

  Current solution of LP-relaxation.

- `COPT_CBINFO_RELAXSOLOBJ` or `"RelaxSolObj"`

  Double information.

  Current objective of LP-relaxation.

- `COPT_CBINFO_NODESTATUS` or `"NodeStatus"`

  Integer information.

  The solution status of the LP-relaxation problem at the current node.

  For possible values, please refer to: *General Constants Chapter: Solution Status (Part)*, except for `NODELIMIT`, `UNSTARTED`, `INF_OR_UNB` .

## 23.5 Parameters

### 23.5.1 Limits and tolerances

- `COPT_DBLPARAM_TIMELIMIT` or `"TimeLimit"`

  Double parameter.

  Time limit (in seconds) of the optimization.

  **Default:** 1e20

  **Minimal:** 0

  **Maximal:** 1e20

- `COPT_DBLPARAM_SOLTIMELIMIT` or `"SolTimeLimit"`

  Double parameter.

  Time limit (in seconds) if a primal feasible solution has been found.

  **Default:** 1e20

  **Minimal:** 0

  **Maximal:** 1e20

- `COPT_INTPARAM_NODELIMIT` or `"NodeLimit"`

  Integer parameter.

  Node limit of the optimization.

  **Default:** -1

  **Minimal:** -1

  **Maximal:** `INT_MAX`

- `COPT_INTPARAM_BARITERLIMIT` or `"BarIterLimit"`

  Integer parameter.

  Iteration limit of barrier method.

  **Default:** 500

  **Minimal:** 0

  **Maximal:** `INT_MAX`

- `COPT_INTPARAM_NLPITERLIMIT` or `"NLPIterLimit"`

Integer parameter.

Iteration limit for the nonlinear solver.

**Default:** 1e4

**Minimum value**: 0

**Maximum value**: `INT_MAX`

- `COPT_INTPARAM_MIPNLPITERLIMIT` or `"MipNLPIterLimit"`

  Integer parameter.

  Iteration limit for solving NLP problem(s) within the MIP solver.

  **Default:** 100

  **Minimal:** -1 (no limit)

  **Maximal:** `INT_MAX`

- `COPT_DBLPARAM_MEMLIMIT` or `"MemLimit"`

  Double parameter.

  Memory limit (MB) for the optimization.

  **Default:** 1e20

  **Minimal:** 0

  **Maximal:** 1e20

- `COPT_DBLPARAM_MATRIXTOL` or `"MatrixTol"`

  Double parameter.

  Input matrix coefficient tolerance.

  **Default:** 1e-10

  **Minimal:** 0

  **Maximal:** 1e-7

- `COPT_DBLPARAM_QMATRIXTOL` or `"QMatrixTol"`

  Double parameter.

  Input Q matrix coefficient tolerance.

  **Default:** 5e-14

  **Minimal:** 0

  **Maximal:** 1e-7

- `COPT_DBLPARAM_FEASTOL` or `"FeasTol"`

  Double parameter.

  The feasibility tolerance.

  **Default:** 1e-6

  **Minimal:** 1e-9.

  **Maximal:** 1e-4

- `COPT_DBLPARAM_DUALTOL` or `"DualTol"`

  Double parameter.

  The tolerance for dual solutions and reduced cost.

  **Default:** 1e-6

**Minimal:** 1e-9

**Maximal:** 1e-4

- `COPT_DBLPARAM_INTTOL` or `"IntTol"`

  Double parameter.

  The integrality tolerance for variables.

  **Default:** 1e-6

  **Minimal:** 1e-9

  **Maximal:** 1e-1

- `COPT_DBLPARAM_PDLPTOL` or `"PDLPTol"`

  Double parameter.

  The PDLP tolerance.

  **Default:** 1e-6

  **Minimal:** 1e-12

  **Maximal:** 1e-4

- `COPT_DBLPARAM_NLPTOL` or `"NLPTol"`

  Double parameter.

  The NLP tolerance.

  **Default:** 1e-8

  **Minimum value**: 1e-13

  **Maximum value**: 1e-3

- `COPT_DBLPARAM_RELGAP` or `"RelGap"`

  Double parameter.

  The relative gap of optimization.

  **Default:** 1e-4

  **Minimal:** 0

  **Maximal:** `DBL_MAX`

- `COPT_DBLPARAM_ABSGAP` or `"AbsGap"`

  Double parameter.

  The absolute gap of optimization.

  **Default:** 1e-6

  **Minimal:** 0

  **Maximal:** `DBL_MAX`

### 23.5.2 Presolving and scaling

- `COPT_INTPARAM_PRESOLVE` or `"Presolve"`

  Integer parameter.

  Level of presolving before solving a model.

  **Default:** -1

  **Possible values:**

  -1: Automatic

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

  4: No Limitations, continues until the model cannot be modified (may be very time-consuming).

- `COPT_INTPARAM_SCALING` or `"Scaling"`

  Integer parameter.

  Whether to perform scaling before solving a problem.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No scaling.

  1: Apply scaling.

- `COPT_INTPARAM_DUALIZE` or `"Dualize"`

  Integer parameter.

  Whether to dualize a problem before solving it.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No dualizing.

  1: Dualizing the problem.

### 23.5.3 Linear Programming related

- `COPT_INTPARAM_LPMETHOD` or `"LpMethod"`

  Integer parameter.

  Method to solve the LP problem.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

For Linear Programming, choose dual simplex method;

For Mixed Integer Linear Programming, choose dual simplex or barrier method.

1: Dual simplex.

2: Barrier.

3: Crossover.

4: Concurrent (Use multiple algorithms simultaneously).

5: Choose between simplex and barrier automatically (Based on features such as sparsity and/or coefficients ranges).

6: First-order method (PDLP).

**Note**

Currently, COPT's GPU mode only supports solving Linear Programming problems using the first-order method (PDLP). To enable it, you need to set `LpMethod=6` first.

- `COPT_INTPARAM_DUALPRICE` or `"DualPrice"`

  Integer parameter.

  Specifies the dual simplex pricing algorithm.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Using Devex pricing algorithm.

  1: Using dual steepest-edge pricing algorithm.

- `COPT_INTPARAM_DUALPERTURB` or `"DualPerturb"`

  Integer parameter.

  Whether to allow the objective function perturbation when using the dual simplex method.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No perturbation.

  1: Allow objective function perturbation.

- `COPT_INTPARAM_BARHOMOGENEOUS` or `"BarHomogeneous"`

  Integer parameter.

  Whether to use homogeneous self-dual form in barrier.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No.

  1: Yes.

- `COPT_INTPARAM_BARORDER` or `"BarOrder"`

Integer parameter.

Barrier ordering algorithm.

**Default:** -1

**Possible values:**

-1: Choose automatically.

0: Approximate Minimum Degree (AMD).

1: Nested Dissection (ND1).

2: Modified Nested Dissection (ND2).

- `COPT_INTPARAM_BARSTART` or `"BarStart"`

  Integer parameter.

  Algorithm for finding initial points in barrier method.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Simple.

  1: Mehrotra.

  2: Modified Mehrotra.

- `COPT_INTPARAM_CROSSOVER` or `"Crossover"`

  Integer parameter.

  Whether to use crossover.

  **Default:** 1

  **Possible values:**

  -1: Choose automatically.

  Only run crossover when the LP solution is not primal-dual feasible.

  0: No.

  1: Yes.

- `COPT_INTPARAM_REQFARKASRAY` or `"ReqFarkasRay"`

  Integer parameter.

  Advanced topic. Whether to compute the dual Farkas or primal ray when the LP is infeasible or unbounded.

  **Default:** 0

  **Possible values:**

  0: No.

  1: Yes.

- `COPT_INTPARAM_REQSENSITIVITY` or `"ReqSensitivity"`

  Integer parameter.

  Whether to compute sensitivity analysis when an optimal basis is available for an LP problem (when solved by the simplex method or by other methods followed by crossover).

  **Default:** 0

  **Possible values:**

0: No.

1: Yes.

### 23.5.4 Semidefinite Programming related

- `COPT_INTPARAM_SDPMETHOD` or `"SDPMethod"`

  Integer parameter.

  Method to solve the SDP problem.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Primal-Dual method.

  1: Alternating direction method of multipliers (ADMM).

  2: Dual method.

### 23.5.5 Integer Programming related

- `COPT_INTPARAM_CUTLEVEL` or `"CutLevel"`

  Integer parameter.

  Level of cutting-planes generation.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_ROOTCUTLEVEL` or `"RootCutLevel"`

  Integer parameter.

  Level of cutting-planes generation of root node.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_TREECUTLEVEL` or `"TreeCutLevel"`

Integer parameter.

Level of cutting-planes generation of search tree.

**Default:** -1

**Possible values:**

-1: Choose automatically.

0: Off

1: Fast

2: Normal

3: Aggressive

- `COPT_INTPARAM_ROOTCUTROUNDS` or `"RootCutRounds"`

  Integer parameter.

  Rounds of cutting-planes generation of root node.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1

  **Maximal:** `INT_MAX`

- `COPT_INTPARAM_NODECUTROUNDS` or `"NodeCutRounds"`

  Integer parameter.

  Rounds of cutting-planes generation of search tree node.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1

  **Maximal:** `INT_MAX`

- `COPT_INTPARAM_HEURLEVEL` or `"HeurLevel"`

  Integer parameter.

  Level of heuristics.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_PREROOTHEURLEVEL` or `"PreRootHeurLevel"`

  Integer parameter.

  Level of pre-root heuristics.

  **Default:** -1

  **Possible value:**

  -1: Choose automatically

  0: Off

  1: Fast

2: Normal

3: Aggressive

- `COPT_INTPARAM_ROUNDINGHEURLEVEL` or `"RoundingHeurLevel"`

  Integer parameter.

  Level of rounding heuristics.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_DIVINGHEURLEVEL` or `"DivingHeurLevel"`

  Integer parameter.

  Level of diving heuristics.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_SUBMIPHEURLEVEL` or `"SubMipHeurLevel"`

  Integer parameter.

  Level of Sub-MIP heuristics.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_FAPHEURLEVEL` or `"FAPHeurLevel"`

  Integer parameter.

  Level of Fix-and-propagate heuristics.

  **Default:** -1

  **Possible values:**

-1: Choose automatically

0: Off

1: Fast

2: Normal

3: Aggressive

4: Heavy

- `COPT_INTPARAM_STRONGBRANCHING` or `"StrongBranching"`

  Integer parameter.

  Level of strong branching.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_CONFLICTANALYSIS` or `"ConflictAnalysis"`

  Integer parameter.

  Whether to perform conflict analysis.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No

  1: Yes

- `COPT_INTPARAM_MIPSTARTMODE` or `"MipStartMode"`

  Integer parameter.

  Mode of MIP starts.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Do not use any MIP starts.

  1: Only load full and feasible MIP starts.

  2: Only load feasible ones (complete partial solutions by solving subMIPs).

  3: Accept all MIP starts. Intended for hierarchical multi-objective optimization.

- `COPT_INTPARAM_MIPSTARTNODELIMIT` or `"MipStartNodeLimit"`

  Integer parameter.

  Limit of nodes for MIP start sub-MIPs.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

**Maximal:** INT_MAX

- COPT_INTPARAM_LINEARIZEINDICATORS or "LinearizeIndicators"

  Integer parameter.

  Controls whether to force the linearization of Indicator constraints.

  **Default:** 0

  **Possible values**

  0: Do not force linearization of indicator constraints.

  1: Force linearization of all indicator constraints.

  (All indicator constraints will be transformed into equivalent linear constraints if enabled.)

- COPT_INTPARAM_LINEARIZESOS or "LinearizeSos"

  Integer parameter.

  Controls whether to force the linearization of SOS constraints.

  **Default:** 0

  **Possible values**

  0: Do not force linearization of SOS constraints.

  1: Force linearization of all SOS constraints.

  (All SOS constraints will be transformed into equivalent linear constraints if enabled.)

- COPT_INTPARAM_MIPREPAIR or "MipRepair"

  Integer parameter.

  Level for repairing the MIP solution in case of numerical issues.

  **Default:** -1

  **Possible values:**

  -1: Only when time left.

  0: Off.

  1: Extend time limit and attempt repair (Fast).

  2: Extend time limit and attempt repair (Normal).

  3: Extend time limit and attempt repair (Aggressive).

### 23.5.6 Nonlinear Programming related

- COPT_INTPARAM_NONCONVEX or "NonConvex"

  Integer parameter.

  Handling strategy for nonconvex models.

  **Default:** -1

  **Possible values**:

  -1: Choose automatically.

  0: Report model as nonconvex and terminate.

  1: Search for a local optimal solution.

- `COPT_INTPARAM_NLPMUUPDATE` or `"NLPMuUpdate"`

  Integer parameter.

  Barrier parameter update strategy for the nonlinear solver.

  **Default:** -1

  **Possible values**:

  -1: Choose automatically.

  0: Monotonic decrease.

  1: Adaptive adjustment.

- `COPT_INTPARAM_NLPLINSCALE` or `"NLPLinScale"`

  Integer parameter.

  Scaling strategy for the linear system in the nonlinear solver.

  **Default:** -1

  **Possible values**:

  -1: Choose automatically.

  0: No scaling.

  1: Always scale.

### 23.5.7 Parallel computing related

- `COPT_INTPARAM_THREADS` or `"Threads"`

  Integer parameter.

  Number of threads to use.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** 128

- `COPT_INTPARAM_BARTHREADS` or `"BarThreads"`

  Integer parameter.

  Number of threads used by barrier. If value is -1, the thread count is determined by parameter `Threads`.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** 128

- `COPT_INTPARAM_SIMPLEXTHREADS` or `"SimplexThreads"`

  Integer parameter.

  Number of threads used by dual simplex. If value is -1, the thread count is determined by parameter `Threads`.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** 128

- `COPT_INTPARAM_CROSSOVERTHREADS` or `"CrossoverThreads"`

Integer parameter.

Number of threads used by crossover. If value is -1, the thread count is determined by parameter `Threads`.

**Default:** -1 (Choose automatically)

**Minimal:** -1 (Choose automatically)

**Maximal:** 128

- `COPT_INTPARAM_MIPTASKS` or `"MipTasks"`

  Integer parameter.

  Number of MIP tasks in parallel.

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** 256

- `COPT_INTPARAM_CONCURRENTLPMODE` or `"ConcurrentLpMode"`

  Integer parameter.

  The LP concurrent solving mode. Only effective when `LpMethod = 4`.

  When `LpMethod = 4` is enabled, the parameters `GPUMode` and `GPUDevice` are ignored. GPU usage and device selection are fully controlled by `ConcurrentLpMode`.

  **Default:** -1 (Choose automatically)

  **Possible values:**

  -1: Choose automatically.

  0: CPU only (simplex + barrier).

  1: CPU (simplex + barrier) + GPU first-order method PDLP (uses GPU device 0).

  2: CPU (simplex + barrier) + GPU barrier (uses GPU device 0).

  3: CPU (simplex + barrier) + GPU first-order method PDLP + GPU barrier (uses GPU device 0 and GPU device 1).

**Note**

If `ConcurrentLpMode` is set to 1 or 2 and no GPU device is detected, or if it is set to 3 and fewer than two GPU devices are detected, the solver reports an error and terminates.

### 23.5.8 IIS computation related

- `COPT_INTPARAM_IISMETHOD` or `"IISMethod"`

  Integer parameter.

  Method for IIS computation.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Find smaller IIS.

  1: Find IIS quickly.

### 23.5.9 Feasibility relaxation related

- `COPT_INTPARAM_FEASRELAXMODE` or `"FeasRelaxMode"`

  Integer parameter.

  Method for feasibility relaxation.

  **Default:** 0

  **Possible values:**

  0: Minimize sum of violations.

  1: Optimize original objective function under minimal sum of violations.

  2: Minimize number of violations.

  3: Optimize original objective function under minimal number of violations.

  4: Minimize sum of squared violations.

  5: Optimize original objective function under minimal sum of squared violations.

### 23.5.10 Tuner related

- `COPT_DBLPARAM_TUNETIMELIMIT` or `"TuneTimeLimit"`

  Double parameter.

  Time limit (in seconds) for parameter tuning. If the parameter value is 0, it will be automatically set by the solver.

  **Default:** 0

  **Minimal:** 0

  **Maximal:** 1e20

- `COPT_DBLPARAM_TUNETARGETTIME` or `"TuneTargetTime"`

  Double parameter.

  Time target for parameter tuning.

  **Default:** 1e-2

  **Minimal:** 0

  **Maximal:** `DBL_MAX`

- `COPT_DBLPARAM_TUNETARGETRELGAP` or `"TuneTargetRelGap"`

  Double parameter.

  Optimal relative tolerance target for parameter tuning.

  **Default:** 1e-4

  **Minimal:** 0

  **Maximal:** `DBL_MAX`

- `COPT_INTPARAM_TUNEMETHOD` or `"TuneMethod"`

  Integer parameter.

  Method for parameter tuning.

  **Default:** -1

  **Possible values:**

-1: Choose automatically.

0: Greedy search strategy.

1: Broader search strategy.

- `COPT_INTPARAM_TUNEMODE` or `"TuneMode"`

  Integer parameter.

  Mode for parameter tuning.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Solving time.

  1: Optimal relative tolerance.

  2: Objective function value.

  3: The lower bound of the objective function value.

- `COPT_INTPARAM_TUNEMEASURE` or `"TuneMeasure"`

  Integer parameter.

  Parameter tuning result calculation method.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: Calculate the average value.

  1: Calculate the maximum value.

- `COPT_INTPARAM_TUNEPERMUTES` or `"TunePermutes"`

  Integer parameter.

  Permutations for each trial parameter set. If the parameter value is 0, it will be automatically set by the solver.

  **Default:** 0

  **Minimal:** 0

  **Maximal:** `INT_MAX`

- `COPT_INTPARAM_TUNEOUTPUTLEVEL` or `"TuneOutputLevel"`

  Integer parameter.

  Parameter tuning log output level.

  **Default:** 2

  **Possible values:**

  0: Not displayed.

  1: Display only improved parameter results.

  2: Displays a summary of the results for each set of parameters.

  3: Display each group of parameter results in detail.

### 23.5.11 Callback related

- `COPT_INTPARAM_LAZYCONSTRAINTS` or `"LazyConstraints"`

  Integer parameter.

  Whether lazy constraints are part of the model.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically.

  0: No.

  1: Yes.

**Notes**

- This parameter only affects MIP.

### 23.5.12 GPU computing related

- `COPT_INTPARAM_GPUMODE` or `"GPUMode"`

  Integer parameter.

  Specifies GPU mode.

  **Default:** -1

  **Possible values:**

  -1: Choose automatically. The first-order method (PDLP) will attempt to use the GPU by default, while the barrier method will use the CPU by default.

  0: Force CPU mode.

  1: Attempt to use the standard GPU mode.

  2: For the barrier method, attempt to use the high-performance GPU mode when solving LP problems, which may lead to higher memory usage. For the first-order method (PDLP), this is equivalent to `GPUMode=1` (standard GPU mode).

- `COPT_INTPARAM_GPUDEVICE` or `"GPUDevice"`

  Integer parameter.

  Specifies GPU device to use (in cases where the running machine has multiple GPUs).

  **Default:** -1 (Choose automatically)

  **Minimal:** -1 (Choose automatically)

  **Maximal:** `INT_MAX`

### 23.5.13 Multi-objective Optimization

- `COPT_DBLPARAM_MULTIOBJTIMELIMIT` or `"MultiObjTimeLimit"`

  Double parameter.

  Time limit (in seconds) for solving a multi-objective model.

  **Default** 1e20

  **Minimum** 0

  **Maximum** 1e20

- `COPT_INTPARAM_MULTIOBJPARAMMODE` or `"MultiObjParamMode"`

  Integer parameter.

  Parameter usage mode for the model of each objective in multi-objective optimization.

  **Default** 0

  **Possible values**

  0: Use the global solver parameters for all multi-objective models

  1: Use the solver parameters specific to each multi-objective model.

### 23.5.14 Other parameters

- `COPT_INTPARAM_LOGGING` or `"Logging"`

  Integer parameter.

  Whether to print optimization logs.

  **Default:** 1

  **Possible values:**

  0: No optimization logs.

  1: Print optimization logs.

- `COPT_INTPARAM_LOGLEVEL` or `"LogLevel"`

  Integer parameter.

  Controls the level of detail in the optimization logs.

  **Default:** 2

  **Possible values:**

  2: Print basic optimization logs.

  3: Print memory usage information in addition to basic optimization logs (for MIP problems).

- `COPT_INTPARAM_LOGTOCONSOLE` or `"LogToConsole"`

  Integer parameter.

  Whether to print optimization logs to console.

  **Default:** 1

  **Possible values:**

  0: No optimization logs to console.

  1: Print optimization logs to console.

- `COPT_INTPARAM_WRITEDUALPROB` or `"WriteDualProb"`

Integer parameter.

Controls whether the dual of the current linear programming problem is written.

When writing a linear programming problem in `LP` / `MPS` / `BIN` format, this parameter specifies whether the original model or the dual model is written.

**Default:** 0

**Possible values:**

0: Write the original model.

1: Write the dual model.

## 23.6 API Functions

The documentations for COPT API functions are grouped by their purposes.

All the return values of COPT API functions are integers, and possible return values are documented in the constants section.

### 23.6.1 Creating the environment and problem

#### COPT_CreateEnvConfig

**Synopsis**

```
int COPT_CreateEnvConfig(copt_env_config **p_config)
```

**Description**

Create a COPT client configuration.

**Arguments**

`p_config`

Output pointer to COPT client configuration.

#### COPT_DeleteEnvConfig

**Synopsis**

```
int COPT_DeleteEnvConfig(copt_env_config **p_config)
```

**Description**

Delete COPT client configuration.

**Arguments**

`p_config`

Input pointer to COPT client configuration.

### COPT_SetEnvConfig

**Synopsis**

```
int COPT_SetEnvConfig(copt_env_config *config, const char *name,
const char *value)
```

**Description**

Set COPT client configuration.

**Arguments**

`config`

COPT client configuration.

`name`

Name of configuration parameter.

`value`

Value of configuration parameter.

### COPT_CreateEnv

**Synopsis**

```
int COPT_CreateEnv(copt_env **p_env)
```

**Description**

Creates a COPT environment.

Calling this function is the first step when using the COPT library. It validates the license, and when the license is okay, the resulting environment variable will allow for creating COPT problems. When the license validation fails, more information can be obtained using `COPT_GetLicenseMsg` to help identify the issue.

**Arguments**

`p_env`

The output pointer to a variable holding COPT environment.

### COPT_CreateEnvWithPath

**Synopsis**

```
int COPT_CreateEnvWithPath(const char *licDir, copt_env **p_env)
```

**Description**

Creates a COPT environment, directory of license files is specified by argument `licDir`.

Calling this function is the first step when using the COPT library. It validates the license, and when the license is okay, the resulting environment variable will allow for creating COPT problems. When the license validation fails, more information can be obtained using `COPT_GetLicenseMsg` to help identify the issue.

**Arguments**

`licDir`

Directory of license files.

`p_env`

Output pointer to a variable holding COPT environment.

### COPT_CreateEnvWithConfig

#### Synopsis

```
int COPT_CreateEnvWithConfig(copt_env_config *config, copt_env
**p_env)
```

#### Description

Creates a COPT environment, client configuration is specified by argument `config`.

Calling this function is the first step when using the COPT library. It validates the client configuration, and when the license is okay, the resulting environment variable will allow for creating COPT problems. When the license validation fails, more information can be obtained using `COPT_GetLicenseMsg` to help identify the issue.

#### Arguments

`config`

Client configuration.

`p_env`

Output pointer to a variable holding COPT environment.

### COPT_CloseEnv

#### Synopsis

```
int COPT_CloseEnv(copt_env **p_env)
```

#### Description

Closes the remote connection of the COPT environment. The token obtained earlier becomes invalid, and so does the remote license of all problems created in that environment.

This function does not release the environment itself. To release the environment, call `COPT_DeleteEnv`.

#### Arguments

`p_env`

Input pointer to a variable holding COPT environment.

### COPT_DeleteEnv

#### Synopsis

```
int COPT_DeleteEnv(copt_env **p_env)
```

#### Description

Deletes the COPT environment created by `COPT_CreateEnv`.

#### Arguments

`p_env`

Input pointer to a variable holding COPT environment.

### COPT_GetLicenseMsg

**Synopsis**

```
int COPT_GetLicenseMsg(copt_env *env, char *buff, int buffSize)
```

**Description**

Returns a C-style string regarding the license validation information. Please refer to this function when `COPT_CreateEnv` fails.

**Arguments**

`env`

The COPT environment.

`buff`

A buffer for holding the resulting string.

`buffSize`

The size of the above buffer.

### COPT_CreateProb

**Synopsis**

```
int COPT_CreateProb(copt_env *env, copt_prob **p_prob)
```

**Description**

Creates an empty COPT problem.

**Arguments**

`env`

The COPT environment.

`p_prob`

Output pointer to a variable holding the COPT problem.

### COPT_CreateCopy

**Synopsis**

```
int COPT_CreateCopy(copt_prob *src_prob, copt_prob **p_dst_prob)
```

**Description**

Create a deep-copy of a COPT problem.

**Note:** The parameter settings will be copied too. To solve the copied problem with different parameters, users should reset its parameters to default by calling `COPT_ResetParam` first, and then set parameters as needed.

**Arguments**

`src_prob`

The pointer to a variable hoding the COPT problem to be copied.

`p_dst_prob`

Output pointer to a variable holding the copied COPT problem.

#### COPT_ClearProb

**Synopsis**

```
int COPT_ClearProb(copt_prob *prob)
```

**Description**

Clear COPT problem data (excluding callback function).

**Arguments**

`prob`

COPT problem.

#### COPT_DeleteProb

**Synopsis**

```
int COPT_DeleteProb(copt_prob **p_prob)
```

**Description**

Deletes the COPT problem created using `COPT_CreateProb`

**Arguments**

`p_prob`

Input pointer to a variable holding the COPT problem.

#### COPT_Update

**Synopsis**

```
int COPT_Update(copt_prob *prob)
```

**Description**

Update the model, including numerical ranges, removed variables and constraints.

**Arguments**

`prob`

COPT problem.

### 23.6.2 Building and modifying a problem

#### COPT_LoadProb

**Synopsis**

```
int COPT_LoadProb(
    copt_prob *prob,
    int nCol,
    int nRow,
    int iObjSense,
    double dObjConst,
    const double *obj,
    const int *colMatBeg,
```

```
    const int *colMatCnt,
    const int *colMatIdx,
    const double *colMatElem,
    const char *colType,
    const double *colLower,
    const double *colUpper,
    const char *rowSense,
    const double *rowBound,
    const double *rowUpper,
    char const *const *colNames,
    char const *const *rowNames)
```

**Description**

Loads a problem defined by arrays.

**Arguments**

`prob`

The COPT problem.

`nCol`

Number of variables (coefficient matrix columns).

`nRow`

Number of constraints (coefficient matrix rows).

`iObjSense`

The optimization sense, either `COPT_MAXIMIZE` or `COPT_MINIMIZE`.

`dObjConst`

The constant part of the objective function.

`obj`

Objective coefficients of variables.

`colMatBeg`, `colMatCnt`, `colMatIdx` and `colMatElem`

Defines the coefficient matrix in compressed column storage (CCS) format. Please see **other information** for an example of the CCS format.

If `colMatCnt` is `NULL`, `colMatBeg` will need to have length of `nCol+1`, and the begin and end pointers to the i-th matrix column coefficients are defined using `colMatBeg[i]` and `colMatBeg[i+1]`.

If `colMatCnt` is provided, the begin and end pointers to the i-th column coefficients are defined using `colMatBeg[i]` and `colMatBeg[i] + colMatCnt[i]`.

`colType`

Types of variables.

If `colType` is `NULL`, all variables will be continuous.

`colLower` and `colUpper`

Lower and upper bounds of variables.

If `colLower` is `NULL`, lower bounds will be 0.

If `colUpper` is `NULL`, upper bounds will be infinity.

`rowSense`

Senses of constraint.

Please refer to the list of all senses constants for all the supported types.

If `rowSense` is `NULL`, then `rowBound` and `rowUpper` will be treated as lower and upper bounds for constraints. This is the recommended method for defining constraints.

If `rowSense` is provided, then `rowBound` and `rowUpper` will be treated as RHS and **range** for constraints. In this case, `rowUpper` is only required when there are `COPT_RANGE` constraints, where the

lower bound is `rowBound[i] - fabs(rowUpper[i])`

upper bound is `rowBound[i]`

`rowBound`

Lower bounds or RHS of constraints.

`rowUpper`

Upper bounds or **range** of constraints.

`colNames` and `rowNames`

Names of variables and constraints. Can be `NULL`.

**Other information**

The compressed column storage (CCS) is a common format for storing sparse matrix. We demonstrate how to store the example matrix with 4 columns and 3 rows in the CCS format.

$$A = \begin{bmatrix} 1.1 & 1.2 & & \\ & 2.2 & 2.3 & \\ & & 3.3 & 3.4 \end{bmatrix} \tag{23.12}$$

```
// Compressed column storage using colMatBeg
colMatBeg[5]  = {  0,   1,   3,   5,   6};
colMatIdx[6]  = {  0,   0,   1,   1,   2,   2};
colMatElem[6] = {1.1, 1.2, 2.2, 2.3, 3.3, 3.4};

// Compressed column storage using both colMatBeg and colMatCnt.
// The * in the example represents unused spaces.
colMatBeg[4]  = {  0,   1,   5,   7};
colMatCnt[4]  = {  1,   2,   2,   1};
colMatIdx[6]  = {  0,   0,   1,   1,   2,   *,   *,   2};
colMatElem[6] = {1.1, 1.2, 2.2, 2.3, 3.3,   *,   *, 3.4};
```

### COPT_AddCol

**Synopsis**

```
int COPT_AddCol(
    copt_prob *prob,
    double dColObj,
    int nColMatCnt,
    const int *colMatIdx,
    const double *colMatElem,
    char cColType,
    double dColLower,
    double dColUpper,
    const char *colName)
```

**Description**

Adds one variable (column) to the problem.

**Arguments**

`prob`

The COPT problem.

`dColObj`

The objective coefficient of the variable.

`nColMatCnt`

Number of non-zero elements in the column.

`colMatIdx`

Row index of non-zero elements in the column.

`colMatElem`

Values of non-zero elements in the column.

`cColType`

The type of the variable.

`dColLower` and `dColUpper`

The lower and upper bounds of the variable.

`colName`

The name of the variable. Can be `NULL`.

### COPT_AddPSDCol

**Synopsis**

```
int COPT_AddPSDCol(copt_prob *prob, int colDim, const char *name)
```

**Description**

Add a PSD variable to the problem.

**Arguments**

`prob`

The COPT problem.

`colDim`

Dimension of new PSD variable.

`name`

Name of new PSD variable. Can be `NULL`.

### COPT_AddRow

**Synopsis**

```
int COPT_AddRow(
    copt_prob *prob,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char *rowName)
```

**Description**

Adds one constraint (row) to the problem.

**Arguments**

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero elements in the row.

`rowMatIdx`

Column index of non-zero elements in the row.

`rowMatElem`

Values of non-zero elements in the row.

`cRowSense`

The sense of the row.

Please refer to the list of all senses constants for all the supported types.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` will be treated as lower and upper bounds for the constraint. This is the recommended method for defining constraints.

If `cRowSense` is provided, then `dRowBound` and `dRowUpper` will be treated as RHS and **range** for the constraint. In this case, `dRowUpper` is only required when `cRowSense = COPT_RANGE`, where

lower bound is `dRowBound - dRowUpper`

upper bound is `dRowBound`

`dRowBound`

Lower bound or RHS of the constraint.

`dRowUpper`

Upper bound or **range** of the constraint.

`rowName`

The name of the constraint. Can be `NULL`.

### COPT_AddCols

**Synopsis**

```
int COPT_AddCols(
    copt_prob *prob,
    int nAddCol,
    const double *colObj,
    const int *colMatBeg,
    const int *colMatCnt,
    const int *colMatIdx,
    const double *colMatElem,
    const char *colType,
    const double *colLower,
    const double *colUpper,
    char const *const *colNames)
```

**Description**

Adds `nAddCol` variables (columns) to the problem.

**Arguments**

`prob`

The COPT problem.

`nAddCol`

Number of new variables.

`colObj`

Objective coefficients of new variables.

`colMatBeg`, `colMatCnt`, `colMatIdx` and `colMatElem`

Defines the coefficient matrix in compressed column storage (CCS) format. Please see **other information** of `COPT_LoadProb` for an example of the CCS format.

`colType`

Types of new variables.

`colLower` and `colUpper`

Lower and upper bounds of new variables.

If `colLower` is `NULL`, lower bounds will be 0.

If `colUpper` is `NULL`, upper bounds will be `COPT_INFINITY`.

`colNames`

Names of new variables. Can be `NULL`.

### COPT_AddPSDCols

**Synopsis**

```
int COPT_AddPSDCols(copt_prob *prob, int nAddCol, const int*
colDims, char const *const *names)
```

**Description**

Add `nAddCol` PSD variables to the problem.

**Arguments**

`prob`

The COPT problem.

`nAddCol`

Number of new PSD variables.

`colDims`

Dimensions of new PSD variables.

`names`

Names of new PSD variables. Can be `NULL`.

### COPT_AddRows

**Synopsis**

```
int COPT_AddRows(
    copt_prob *prob,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowBound,
```

```
    const double *rowUpper,
    char const *const *rowNames)
```

**Description**

Adds `nAddRow` constraints (rows) to the problem.

**Arguments**

`prob`

The COPT problem.

`nAddRow`

Number of new constraints.

`rowMatBeg, rowMatCnt, rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new constraints.

Please refer to the list of all senses constants for all the supported types.

If `rowSense` is `NULL`, then `rowBound` and `rowUpper` will be treated as lower and upper bounds for constraints. This is the recommended method for defining constraints.

If `rowSense` is provided, then `rowBound` and `rowUpper` will be treated as RHS and **range** for constraints. In this case, `rowUpper` is only required when there are `COPT_RANGE` constraints, where the

lower bound is `rowBound[i] - fabs(rowUpper[i])`

upper bound is `rowBound[i]`

`rowBound`

Lower bounds or RHS of new constraints.

`rowUpper`

Upper bounds or **range** of new constraints.

`rowNames`

Names of new constraints. Can be `NULL`.

### COPT_AddSOSs

**Synopsis**

```
int COPT_AddSOSs(
    copt_prob *prob,
    int nAddSOS,
    const int *sosType,
    const int *sosMatBeg,
    const int *sosMatCnt,
    const int *sosMatIdx,
    const double *sosMatWt)
```

**Description**

Add `nAddSOS` SOS constraints to the problem. If `sosMatWt` is `NULL`, then COPT will generate it internally.

**Note:** if a problem contains SOS constraints, the problem is a MIP.

**Arguments**

`prob`

The COPT problem.

`nAddSOS`

Number of new SOS constraints.

`sosType`

Types of SOS constraints.

`sosMatBeg, sosMatCnt, sosMatIdx` and `sosMatWt`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`sosMatWt`

Weights of variables in SOS constraints. Can be `NULL`.

### COPT_AddCones

**Synopsis**

```
int COPT_AddCones(
    copt_prob *prob,
    int nAddCone,
    const int *coneType,
    const int *coneBeg,
    const int *coneCnt,
    const int *coneIdx)
```

**Description**

Add `nAddCone` Second-Order-Cone (SOC) constraints.

**Arguments**

`prob`

The COPT problem.

`nAddCone`

Number of new SOC constraints.

`coneType`

Types of SOC constraints.

`coneBeg, coneCnt, coneIdx`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

### COPT_AddExpCones

#### Synopsis

```
int COPT_AddExpCones(
    copt_prob *prob,
    int nAddCone,
    const int *coneType,
    const int *coneIdx)
```

#### Description

Add `nAddCone` exponential cone constraints.

#### Arguments

`prob`

The COPT problem.

`nAddCone`

Number of new exponential cone constraints.

`coneType`

Types of exponential cone constraints.

`coneIdx`

Array of subscripts for the variables that constitute the exponential cone constraints.

### COPT_AddAffineCone

#### Synopsis

```
int COPT_AddAffineCone(
    copt_prob *prob,
    int coneType,
    int nConeDim,
    int nAlphaDim,
    const double *alphaElem,
    const int *psdBeg,
    const int *psdCnt,
    const int *psdColIdx,
    const int *psdMatIdx,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const double *rowConst,
    const char *name)
```

#### Description

Add the affine cone constraints.

**Arguments**

`prob`

The COPT model.

`coneType`

The type of the affine cone. Please refer to *SOC constraint types* and *Exponential Cone type* for possible values.

`nConeDim`

The dimension of the affine cone.

`nAlphaDim`

Reserved parameter, currently not in use.

`alphaElem`

Reserved parameter, currently not in use.

`psdBeg, psdCnt, psdColIdx, psdMatIdx`

Represents the PSD terms in the affine cone.

`psdBeg` indicates the starting position of the PSD terms in each affine cone term, `psdCnt` specifies the number of PSD terms, `psdColIdx` indicates the index of the PSD variable, and `psdMatIdx` refers to the index of the symmetric matrix.

`rowMatBeg, rowMatCnt, rowMatIdx, rowMatElem`

Represents the linear terms in the affine cone.

The coefficient matrix is provided in CRS-format. For detailed examples, please refer to **Additional Information** in `COPT_LoadProb`.

`rowConst`

The constant terms in the affine cone.

`name`

The name of the affine cone.

## COPT_AddQConstr

**Synopsis**

```
int COPT_AddQConstr(
    copt_prob *prob,
    int nRowMatCnt,
    const int *rowMatIdx,
    const int *rowMatElem,
    int nQMatCnt,
    const int *qMatRow,
    const int *qMatCol,
    const double *qMatElem,
    char cRowSense,
```

```
    double dRowBound, const char *name)
```

**Description**

Add a general quadratic constraint.

**Note** Only convex quadratic constraint is currently supported.

**Arguments**

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero linear terms of the quadratic constraint (row).

`rowMatIdx`

Column index of non-zero linear terms of the quadratic constraint (row).

`rowMatElem`

Values of non-zero linear terms of the quadratic constraint (row).

`nQMatCnt`

Number of non-zero quadratic terms of the quadratic constraint (row).

`qMatRow`

Row index of non-zero quadratic terms of the quadratic constraint (row).

`qMatCol`

Column index of non-zero quadratic terms of the quadratic constraint (row).

`qMatElem`

Values of non-zero quadratic terms of the quadratic constraint (row).

`cRowSense`

The sense of the quadratic constraint (row).

`dRowBound`

Right hand side of the quadratic constraint (row).

`name`

Name of the quadratic constraint (row).

### COPT_AddPSDConstr

**Synopsis**

```
int COPT_AddPSDConstr(
    copt_prob *prob,
    int nRowMatCnt,
    const int *rowMatIdx,
    const int *rowMatElem,
    int nColCnt,
    const int *psdColIdx,
    const int *symMatIdx,
```

```
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char *name)
```

**Description**

Add a PSD constraint.

**Arguments**

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero linear terms of the PSD constraint.

`rowMatIdx`

Column index of non-zero linear terms of the PSD constraint.

`rowMatElem`

Values of non-zero linear terms of the PSD constraint.

`nColCnt`

Number of PSD terms of the PSD constraint.

`psdColIdx`

PSD variable index of PSD terms of the PSD constraint.

`symMatIdx`

Symmetric matrix index of PSD terms of the PSD constraint.

`cRowSense`

Senses of new PSD constraint.

Please refer to the list of all senses constants for all the supported types.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` will be treated as lower and upper bounds for the constraint. This is the recommended method for defining constraints.

If `cRowSense` is provided, then `dRowBound` and `dRowUpper` will be treated as RHS and **range** for the constraint. In this case, `dRowUpper` is only required when `cRowSense = COPT_RANGE`, where

lower bound is `dRowBound - dRowUpper`

upper bound is `dRowBound`

`dRowBound`

Lower bound or RHS of the PSD constraint.

`dRowUpper`

Upper bound or **range** of the PSD constraint.

`name`

Name of the PSD constraint. Can be `NULL`.

### COPT_AddLMIConstr

**Synopsis**

```
int COPT_AddLMIConstr(
    copt_prob *prob,
    int nDim,
    int nLMIMatCnt,
    const int *colIdx,
    const int *symMatIdx,
    int constMatIdx,
    const char *name)
```

**Description**

Add a LMI constraint to the problem.

**Arguments**

`prob`

The COPT problem.

`nDim`

Dimension of symmetric matrix in the LMI constraint.

`nLMIMatCnt`

Number of coefficient matrix entries in the LMI constraint.

`colIdx`

Index of scalar variable in the LMI constraint.

`symMatIdx`

Index of symmetric coefficient matrix in the LMI constraint.

`constMatIdx`

Index of constant-term symmetric matrix in the LMI constraint.

`name`

Name of LMI constraint. Can be `NULL`.

### COPT_AddIndicator

**Synopsis**

```
int COPT_AddIndicator(
    copt_prob *prob,
    int binColIdx,
    int binColVal,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense, double dRowBound)
```

**Description**

Add an indicator constraint to the problem.

**Arguments**

`prob`

The COPT problem.

`binColIdx`

Index of indicator variable (column).

`binColVal`

Value of indicator variable (column).

`nRowMatCnt`

Number of non-zero elements in the linear constraint (row).

`rowMatIdx`

Column index of non-zero elements in the linear constraint (row).

`rowMatElem`

Values of non-zero elements in the linear constraint (row).

`cRowSense`

The sense of the linear constraint (row). Options are: `COPT_EQUAL` , `COPT_LESS_EQUAL` and `COPT_GREATER_EQUAL` .

`dRowBound`

Right hand side of the linear constraint (row).

### COPT_AddIndicators

**Synopsis**

```
int COPT_AddIndicators(
    copt_prob *prob,
    int nInd,
    int *indType,
    int *binColIdx,
    int *binColVal,
    const int *rowMatBeg,
    const int *RowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double *dRowBound,
    char const *const indNames);
```

**Description**

Add `nInd` indicator constraints to the problem.

**Arguments**

`prob`

The COPT problem.

`nInd`

Number of indicator constraints.

`indType`

Type for indicator constraints. Please refer to *Indicator constraint types* for possible values.

`binColIdx`

Index of indicator variable (column).

`binColVal`

Value of indicator variable (column).

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format for linear constraint in the indicator constraints.

Please see **other information** of `COPT_LoadProb` for an example of the CRS format.

`cRowSense`

The sense of the linear constraint (row). Options are: `COPT_EQUAL` , `COPT_LESS_EQUAL` and `COPT_GREATER_EQUAL` .

`dRowBound`

Right hand side of the linear constraint (row).

`indNames`

Names for indicator constraints.

### COPT_AddSymMat

#### Synopsis

```
int COPT_AddSymMat(copt_prob *prob, int ndim, int nelem, int *rows,
int *cols, double *elems)
```

#### Description

Add a symmetric matrix to the problem. (Expect lower triangle part)

#### Arguments

`prob`

The COPT problem.

`ndim`

Dimension of symmetric matrix.

`nelem`

Number of non-zeros of symmetric matrix.

`rows`

Row index of symmetric matrix.

`cols`

Column index of symmetric matrix.

`elems`

Nonzero elements of symmetric matrix.

### COPT_AddNLConstr

**Synopsis**

```
int COPT_AddNLConstr(
    copt_prob *prob,
    int nToken,
    int nTokenElem,
    const int *token,
    const double *tokenElem,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char *name)
```

**Description**

Add a nonlinear expression constraint. Currently, only prefix notation is supported.

**Arguments**

`prob`

The COPT problem.

`nToken`

Number of tokens in the expression.

`nTokenElem`

Number of constants in the expression.

`token`

Array of expression tokens.

`tokenElem`

Array of constants in the expression.

`nRowMatCnt`

Number of linear terms in the constraint.

`rowMatIdx`

Indices of linear terms in the constraint.

`rowMatElem`

Coefficients of linear terms in the constraint.

`cRowSense`

Constraint type.

Refer to the constants section for the supported constraint types in COPT.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` are treated as the lower and upper bounds of the constraint. This is the recommended way to define constraints.

If `cRowSense` is a meaningful value, then `dRowBound` and `dRowUpper` are treated as the RHS and **range**. In this case, `dRowUpper` is used only when the constraint type is `COPT_RANGE`. The constraint bounds in this scenario are:

Lower bound: `dRowBound - dRowUpper`

Upper bound: `dRowBound`

`dRowBound`

Lower bound or RHS of the constraint.

`dRowUpper`

Upper bound or **range** of the constraint.

`name`

Name of the constraint. Can be set to `NULL`.

### COPT_AddNLConstrs

#### Synopsis

```
int COPT_AddNLConstrs(
    copt_prob *prob,
    int nConstrs,
    const int *tokenBeg,
    const int *tokenCnt,
    const int *tokenElemBeg,
    const int *tokenElemCnt,
    const int *token,
    const double *tokenElem,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowBound,
    const double *rowUpper,
    char const *const *rowNames)
```

#### Description

Add a set of nonlinear expression constraints. Currently, only prefix notation is supported.

#### Arguments

`prob`

The COPT problem.

`nConstrs`

Number of constraints.

`tokenBeg`

Array of starting positions of tokens in expressions.

`tokenCnt`

Array of counts of tokens in expressions.

`tokenElemBeg`

Array of starting positions of constants in expressions.

`tokenElemCnt`

Array of counts of constants in expressions.

`token`

Array of tokens in expressions.

`tokenElem`

Array of constants in expressions.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Provide the coefficient matrix in a row-compressed storage format. For specific examples of sparse matrix compression storage format, refer to the **Additional Information** section in `COPT_LoadProb`.

`rowSense`

Constraint types.

Refer to the constants section for supported constraint types in COPT.

If `rowSense` is not provided, `rowBound` and `rowUpper` are treated as the lower and upper bounds of constraints. This is the recommended way to define constraints.

If `rowSense` is provided, `rowBound` and `rowUpper` are treated as the RHS and **range**. In this case, the `rowUpper` array is only used when there is a `COPT_RANGE` constraint type. The constraint bounds in this scenario are:

Lower bound: `rowBound[i] - fabs(rowUpper[i])`

Upper bound: `rowBound[i]`

`rowBound`

Lower bound or RHS of the new constraints.

`rowUpper`

Upper bound or **range** of the new constraints.

`rowNames`

Names of the new constraints. Can be set to `NULL`.

### COPT_DelCols

**Synopsis**

```
int COPT_DelCols(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` variables (columns) from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

Number of variables to be deleted.

`list`

A list of index of variables to be deleted.

### COPT_DelPSDCols

**Synopsis**

```
int COPT_DelPSDCols(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` PSD variables from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

Number of PSD variables to be deleted.

`list`

A list of index of PSD variables to be deleted.

### COPT_DelRows

**Synopsis**

```
int COPT_DelRows(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` constraints (rows) from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of constraints to be deleted.

`list`

The list of index of constraints to be deleted.

### COPT_DelSOSs

**Synopsis**

```
int COPT_DelSOSs(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` SOS constraints from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of SOS constraints to be deleted.

`list`

The list of index of SOS constraints to be deleted.

### COPT_DelCones

**Synopsis**

```
int COPT_DelCones(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` Second-Order-Cone (SOC) constraints from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of SOC constraints to be deleted.

`list`

The list of index of SOC constraints to be deleted.

### COPT_DelExpCones

**Synopsis**

```
int COPT_DelExpCones(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` exponential cone constraints from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of exponential cone constraints to be deleted.

`list`

The list of index of exponential cone constraints to be deleted.

### COPT_DelAffineCones

**Synopsis**

```
int COPT_DelAffineCones(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` affine cone constraints from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of affine cone constraints to be deleted.

`list`

The list of index of affine cone constraints to be deleted.

### COPT_DelQConstrs

**Synopsis**

```
int COPT_DelQConstrs(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` quadratic constraints from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of quadratic constraints to be deleted.

`list`

The list of index of quadratic constraints to be deleted.

### COPT_DelPSDConstrs

**Synopsis**

```
int COPT_DelPSDConstrs(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` PSD constraints from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of PSD constraints to be deleted.

`list`

The list of index of PSD constraints to be deleted.

### COPT_DelLMIConstrs

**Synopsis**

```
int COPT_DelLMIConstrs(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` LMI constraints from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of LMI constraints to be deleted.

`list`

The list of index of LMI constraints to be deleted.

### COPT_DelIndicators

**Synopsis**

```
int COPT_DelIndicators(copt_prob *prob, int num, const int *list)
```

**Description**

Deletes `num` indicator constraints from the problem.

**Arguments**

`prob`

The COPT problem.

`num`

The number of indicator constraints to be deleted.

`list`

The list of index of indicator constraints to be deleted.

### COPT_DelNLConstrs

**Synopsis**

```
int COPT_DelNLConstrs(copt_prob *prob, int num, const int *list)
```

**Description**

Delete `num` nonlinear expression constraints.

**Arguments**

`prob`

The COPT problem.

`num`

Number of nonlinear expression constraints to delete.

`list`

List of indices of nonlinear expression constraints to delete.

### COPT_DelQuadObj

**Synopsis**

```
int COPT_DelQuadObj(copt_prob *prob)
```

**Description**

Deletes the quadratic terms from the quadratic objective function.

**Arguments**

`prob`

The COPT problem.

### COPT_DelNLObj

**Synopsis**

```
int COPT_DelNLObj(copt_prob *prob)
```

**Description**

Delete the nonlinear expression terms in the objective function.

**Arguments**

`prob`

The COPT problem.

### COPT_DelPSDObj

**Synopsis**

```
int COPT_DelPSDObj(copt_prob *prob)
```

**Description**

Deletes the PSD terms from objective function.

**Arguments**

`prob`

The COPT problem.

### COPT_SetElem

**Synopsis**

```
int COPT_SetElem(copt_prob *prob, int iCol, int iRow, double
newElem)
```

**Description**

Set coefficient of specified row and column.

**Note:** If `newElem` is smaller than or equal to parameter `MatrixTol`, the coefficient will be set as zero.

**Arguments**

`prob`

The COPT problem.

`iCol`

Column index.

`iRow`

Row index.

`newElem`

New coefficient.

### COPT_SetElems

**Synopsis**

```
int COPT_SetElems(copt_prob *prob, int nelem, const int *cols, const
int *rows, const double *elems)
```

**Description**

Set the coefficients of the specified columns and rows in batches.

**Note** The index pairs of columns and rows cannot appear repeatedly.

**Arguments**

`prob`

The COPT problem.

`nelem`

The number of new coefficients to be set.

`cols`

Column indexes.

`rows`

Row indexes.

`elems`

The values of the new coefficients to be set.

### COPT_SetPSDElem

**Synopsis**

```
int COPT_SetPSDElem(copt_prob *prob, int iCol, int iRow, int newIdx)
```

**Description**

Set symmetric matrix index for given PSD term of PSD constraint.

**Arguments**

`prob`

The COPT problem.

`iCol`

PSD variable index.

`iRow`

PSD constraint index.

`newIdx`

New symmetric matrix index.

### COPT_SetLMIElem

**Synopsis**

```
int COPT_SetLMIElem(copt_prob *prob, int iCol, int iRow, int newIdx)
```

**Description**

Set symmetric matrix index for given term of LMI constraint.

**Arguments**

`prob`

The COPT problem.

`iCol`

Scalar variable index.

`iRow`

LMI constraint index.

`newIdx`

New coefficient symmetric matrix index.

### COPT_SetObjSense

**Synopsis**

```
int COPT_SetObjSense(copt_prob *prob, int iObjSense)
```

**Description**

Change the objective function sense.

**Arguments**

`prob`

The COPT problem.

`iObjSense`

The optimization sense, either `COPT_MAXIMIZE` or `COPT_MINIMIZE`.

### COPT_SetObjConst

**Synopsis**

```
int COPT_SetObjConst(copt_prob *prob, double dObjConst)
```

**Description**

Set the constant term of objective function.

**Arguments**

`prob`

The COPT problem.

`dObjConst`

The constant term of objective function.

### COPT_SetColObj/Type/Lower/Upper/Names

#### Synopsis

```
int COPT_SetColObj(copt_prob *prob, int num, const int *list, const
double *obj)
```

```
int COPT_SetColType(copt_prob *prob, int num, const int *list, const
char *type)
```

```
int COPT_SetColLower(copt_prob *prob, int num, const int *list,
const double *lower)
```

```
int COPT_SetColUpper(copt_prob *prob, int num, const int *list,
const double *upper)
```

```
int COPT_SetColNames(copt_prob *prob, int num, const int *list, char
const *const *names)
```

#### Description

These five API functions each modifies

objective coefficients

variable types

lower bounds

upper bounds

names

of `num` variables (columns) in the problem.

#### Arguments

`prob`

The COPT problem.

`num`

Number of variables to modify.

`list`

A list of index of variables to modify.

`obj`

New objective coefficients for each variable in the `list`.

`types`

New types for each variable in the `list`.

`lower`

New lower bounds for each variable in the `list`.

`upper`

New upper bounds for each variable in the `list`.

`names`

New names for each variable in the `list`.

### COPT_SetPSDColNames

#### Synopsis

```
int COPT_SetPSDColNames(copt_prob *prob, int num, const int *list,
char const *const *names)
```

#### Description

Modify names of `num` PSD variables.

#### Arguments

`prob`

The COPT problem.

`num`

Number of PSD variables to modify.

`list`

A list of index of PSD variables to modify.

`names`

New names for each PSD variable in the `list`.

### COPT_SetRowLower/Upper/Names

#### Synopsis

```
int COPT_SetRowLower(copt_prob *prob, int num, const int *list,
const double *lower)
```

```
int COPT_SetRowUpper(copt_prob *prob, int num, const int *list,
const double *upper)
```

```
int COPT_SetRowNames(copt_prob *prob, int num, const int *list, char
const *const *names)
```

#### Description

These three API functions each modifies

lower bounds

upper bounds

names

of `num` constraints (rows) in the problem.

#### Arguments

`prob`

The COPT problem.

`num`

Number of constraints to modify.

`list`

A list of index of constraints to modify.

`lower`

New lower bounds for each constraint in the `list`.

`upper`

New upper bounds for each constraint in the `list`.

`names`

New names for each constraint in the `list`.

### COPT_SetQConstrSense/Rhs/Names

**Synopsis**

```
int COPT_SetQConstrSense(copt_prob *prob, int num, const int *list,
const char *sense)
```

```
int COPT_SetQConstrRhs(copt_prob *prob, int num, const int *list,
const double *rhs)
```

```
int COPT_SetQConstrNames(copt_prob *prob, int num, const int *list,
char const *const *names)
```

**Description**

These three API functions each modifies

senses

RHS

names

of `num` quadratic constraints (rows) in the problem.

**Arguments**

`prob`

The COPT problem.

`num`

Number of quadratic constraints to modify.

`list`

A list of index of quadratic constraints to modify.

`sense`

New senses for each quadratic constraint in the `list`.

`rhs`

New RHS for each quadratic constraint in the `list`.

`names`

New names for each quadratic constraint in the `list`.

### COPT_SetPSDConstrLower/Upper/Names

**Synopsis**

```
int COPT_SetPSDConstrLower(copt_prob *prob, int num, const int
*list, const double *lower)
```

```
int COPT_SetPSDConstrUpper(copt_prob *prob, int num, const int
*list, const double *upper)
```

```
int COPT_SetPSDConstrNames(copt_prob *prob, int num, const int
*list, char const *const *names)
```

**Description**

These three API functions each modifies

lower bounds

upper bounds

names

of `num` PSD constraints in the problem.

**Arguments**

`prob`

The COPT problem.

`num`

Number of PSD constraints to modify.

`list`

A list of index of PSD constraints to modify.

`lower`

New lower bounds for each PSD constraint in the `list`.

`upper`

New upper bounds for each PSD constraint in the `list`.

`names`

New names for each PSD constraint in the `list`.

### COPT_SetLMIConstrRhs

**Synopsis**

```
int COPT_SetLMIConstrRhs(copt_prob *prob, int num, const int *list,
const int *newIdx)
```

**Description**

Modify the constant-term symmetric matrix of `num` LMI constraints.

**Arguments**

`prob`

The COPT problem.

`num`

Number of LMI constraints to modify.

`list`

A list of index of LMI constraints to modify.

`newIdx`

The new index of the constant-term symmetric matrix to be set.

### COPT_SetLMIConstrNames

**Synopsis**

```
int COPT_SetLMIConstrNames(copt_prob *prob, int num, const int
*list, char const *const *names)
```

**Description**

Modify the names of `num` LMI constraints.

**Arguments**

`prob`

The COPT problem.

`num`

Number of LMI constraints to modify.

`list`

A list of index of LMI constraints to modify.

`names`

New names for each LMI constraint in the `list`.

### COPT_SetIndicatorNames

**Synopsis**

```
int COPT_SetIndicatorNames(copt_prob *prob, int num, const int
*list, char const *const *names)
```

**Description**

Modifies names of `num` indicator constraints in the problem.

**Arguments**

`prob`

The COPT problem.

`num`

Number of indicator constraints to modify.

`list`

A list of indexes of indicator constraints to modify.

`names`

New names for each indicator constraint in the `list`.

### COPT_SetAffineConeNames

**Synopsis**

```
int COPT_SetAffineConeNames(copt_prob *prob, int num, const int
*list, char const *const *names)
```

**Description**

Modifies names of `num` affine cones in the problem.

**Arguments**

`prob`

The COPT problem.

`num`

Number of affine cones to modify.

`list`

A list of indexes of affine cones to modify.

`names`

New names for each affine cone in the `list`.

### COPT_SetNLConstrLower/Upper/Names

#### Synopsis

```
int COPT_SetNLConstrLower(copt_prob *prob, int num, const int *list,
const double *lower)
```

```
int COPT_SetNLConstrUpper(copt_prob *prob, int num, const int *list,
const double *upper)
```

```
int COPT_SetNLConstrNames(copt_prob *prob, int num, const int *list,
char const *const *names)
```

#### Description

These three functions respectively modify the following properties of `num` nonlinear expression constraints:

Lower bound

Upper bound

Name

#### Arguments

`prob`

The COPT problem.

`num`

Number of nonlinear expression constraints to modify.

`list`

List of nonlinear expression constraints to modify.

`lower`

New lower bounds for the listed nonlinear expression constraints.

`upper`

New upper bounds for the listed nonlinear expression constraints.

`names`

New names for the listed nonlinear expression constraints.

### COPT_ReplaceColObj

**Synopsis**

```
int COPT_ReplaceColObj(copt_prob *prob, int num, const int *list,
const double *obj)
```

**Description**

Replace objective function with new objective function represented by specified objective costs.

**Arguments**

`prob`

The COPT problem.

`num`

Number of variables to be modified.

`list`

Index of variables to be modified.

`obj`

Objective costs of modified variables.

### COPT_ReplacePSDObj

**Synopsis**

```
int COPT_ReplacePSDObj(copt_prob *prob, int num, const int *list,
const int *idx)
```

**Description**

Replace PSD terms in objective function with specified PSD terms.

**Arguments**

`prob`

The COPT problem.

`num`

Number of PSD variables to be modified.

`list`

Index of PSD variables to be modified.

`idx`

Symmetric matrix index of modified PSD variables.

### COPT_SetQuadObj

**Synopsis**

```
int COPT_SetQuadObj(copt_prob *prob, int num, int *qRow, int *qCol,
double *qElem)
```

**Description**

Set the quadratic terms of the quadratic objective function.

**Arguments**

`prob`

The COPT problem.

`num`

Number of non-zero quadratic terms of the quadratic objective function.

`qRow`

Row index of non-zero quadratic terms of the quadratic objective function.

`qCol`

Column index of non-zero quadratic terms of the quadratic objective function.

`qElem`

Values of non-zero quadratic terms of the quadratic objective function.

### COPT_SetNLObj

**Synopsis**

```
int COPT_SetNLObj(copt_prob *prob, int nToken, int nTokenElem, const
int *token, const double *tokenElem)
```

**Description**

Set the nonlinear expression terms in the nonlinear objective function.

**Arguments**

`prob`

The COPT problem.

`nToken`

Number of tokens in the expression.

`nTokenElem`

Number of constants in the expression.

`token`

Array of tokens in the expression.

`tokenElem`

Array of constants in the expression.

#### COPT_SetPSDObj

**Synopsis**

```
int COPT_SetPSDObj(copt_prob *prob, int iCol, int newIdx)
```

**Description**

Set PSD terms of objective function.

**Arguments**

`prob`

The COPT problem.

`iCol`

PSD variable index.

`newIdx`

Symmetric matrix index.

### 23.6.3 Reading and writing the problem

#### COPT_ReadMps

**Synopsis**

```
int COPT_ReadMps(copt_prob *prob, const char *mpsfilename)
```

**Description**

Reads a problem from a MPS file.

**Arguments**

`prob`

The COPT problem.

`mpsfilename`

The path to the MPS file.

#### COPT_ReadLp

**Synopsis**

```
int COPT_ReadLp(copt_prob *prob, const char *lpfilename)
```

**Description**

Read a problem from a LP file.

**Arguments**

`prob`

The COPT problem.

`lpfilename`

The path to the LP file.

### COPT_ReadSDPA

**Synopsis**

```
int COPT_ReadSDPA(copt_prob *prob, const char *sdpafilename)
```

**Description**

Reads a problem from SDPA format file.

**Arguments**

`prob`

The COPT problem.

`sdpafilename`

The path to the SDPA format file.

### COPT_ReadCbf

**Synopsis**

```
int COPT_ReadCbf(copt_prob *prob, const char *cbffilename)
```

**Description**

Reads a problem from CBF format file.

**Arguments**

`prob`

The COPT problem.

`cbffilename`

The path to the CBF format file.

### COPT_ReadBin

**Synopsis**

```
int COPT_ReadBin(copt_prob *prob, const char *binfilename)
```

**Description**

Reads a problem from a COPT binary format file.

**Arguments**

`prob`

The COPT problem.

`binfilename`

The path to the COPT binary format file.

### COPT_ReadBlob

**Synopsis**

```
int COPT_ReadBlob(copt_prob *prob, void *blob, COPT_INT64 len)
```

**Description**

Reads a problem from COPT serialized data.

**Arguments**

`prob`

The COPT problem.

`blob`

Serialized data.

`len`

Length of serialized data.

### COPT_WriteMps

**Synopsis**

```
int COPT_WriteMps(copt_prob *prob, const char *mpsfilename)
```

**Description**

Writes the problem to a MPS file.

For a linear programming problem, set `COPT_INTPARAM_WRITEDUALPROB` to 1 to write the dual model instead of the original one.

**Arguments**

`prob`

The COPT problem.

`mpsfilename`

The path to the MPS file.

### COPT_WriteMpsStr

**Synopsis**

```
int COPT_WriteMpsStr(copt_prob *prob, char *str, int nStrSize, int
*pReqSize)
```

**Description**

Writes the problem to a string buffer as MPS format.

**Arguments**

`prob`

The COPT problem.

`str`

String buffer of MPS-format problem.

`nStrSize`

The size of string buffer.

`pReqSize`

Minimum space requirement of string buffer for problem.

### COPT_WriteLp

**Synopsis**

```
int COPT_WriteLp(copt_prob *prob, const char *lpfilename)
```

**Description**

Writes the problem to a LP file.

For a linear programming problem, set `COPT_INTPARAM_WRITEDUALPROB` to 1 to write the dual model instead of the original one.

**Arguments**

`prob`

The COPT problem.

`lpfilename`

The path to the LP file.

### COPT_WriteCbf

**Synopsis**

```
int COPT_WriteCbf(copt_prob *prob, const char *cbffilename)
```

**Description**

Writes the problem to a CBF format file.

**Arguments**

`prob`

The COPT problem.

`cbffilename`

The path to the CBF format file.

### COPT_WriteBin

**Synopsis**

```
int COPT_WriteBin(copt_prob *prob, const char *binfilename)
```

**Description**

Writes the problem to a COPT binary format file.

For a linear programming problem, set `COPT_INTPARAM_WRITEDUALPROB` to 1 to write the dual model instead of the original one.

**Arguments**

`prob`

The COPT problem.

`binfilename`

The path to the COPT binary format file.

#### COPT_WriteNL

**Synopsis**

```
int COPT_WriteNL(copt_prob *prob, const char *nlfilename)
```

**Description**

Export the internal COPT model to an NL format file.

**Arguments**

`prob`

The COPT problem.

`nlfilename`

Path to the NL format file.

#### COPT_WriteBlob

**Synopsis**

```
int COPT_WriteBlob(copt_prob *prob, int tryCompress, void **p_blob,
COPT_INT64 *pLen)
```

**Description**

Writes the problem to COPT serialized data.

**Arguments**

`prob`

The COPT problem.

`tryCompress`

Whether to compress data.

`p_blob`

Output pointer of serialized data.

`pLen`

Pointer to length of serialized data.

### 23.6.4 Solving the problem and accessing solutions

#### COPT_SolveLp

**Synopsis**

```
int COPT_SolveLp(copt_prob *prob)
```

**Description**

Solves the LP, QP, QCP, SOCP and SDP problem.

If problem is a MIP, then integer restrictions on variables will be ignored, and SOS constraints, indicator constraints will be discarded, and the problem will be solved as a LP.

**Arguments**

`prob`

The COPT problem.

### COPT_Solve

**Synopsis**

```
int COPT_Solve(copt_prob *prob)
```

**Description**

Solves the problem.

**Arguments**

`prob`

The COPT problem.

### COPT_GetSolution

**Synopsis**

```
int COPT_GetSolution(copt_prob *prob, double *colVal)
```

**Description**

Obtains MIP solution.

**Arguments**

`prob`

The COPT problem.

`colVal`

Solution values of variables.

### COPT_GetPoolObjVal

**Synopsis**

```
int COPT_GetPoolObjVal(copt_prob *prob, int iSol, double *p_objVal)
```

**Description**

Obtains the `iSol` -th objective value in solution pool.

**Arguments**

`prob`

The COPT problem.

`iSol`

Index of solution.

`p_objVal`

Pointer to objective value.

### COPT_GetPoolSolution

**Synopsis**

```
int COPT_GetPoolSolution(copt_prob *prob, int iSol, int num, const
int *list, double *colVal)
```

**Description**

Obtains the `iSol` -th solution.

**Arguments**

`prob`

The COPT problem.

`iSol`

Index of solution.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colVal`

Array of solution.

### COPT_GetLpSolution

**Synopsis**

```
int COPT_GetLpSolution(copt_prob *prob, double *value, double
*slack, double *rowDual, double *redCost)
```

**Description**

Obtains LP, QP, QCP, SOCP and SDP solutions.

**Note:** For SDP, please use `COPT_GetPSDColInfo` to obtain primal/dual solution of PSD variable.

**Arguments**

`prob`

The COPT problem.

`value`

Solution values of variables. Can be `NULL`.

`slack`

Solution values of slack variables. They are also known as activities of constraints. Can be `NULL`.

`rowDual`

Dual values of constraints. Can be `NULL`.

`redCost`

Reduced cost of variables. Can be `NULL`.

### COPT_SetLpSolution

**Synopsis**

```
int COPT_SetLpSolution(copt_prob *prob, double *value, double
*slack, double *rowDual, double *redCost)
```

**Description**

Set LP solution.

**Arguments**

`prob`

The COPT problem.

`value`

Solution values of variables.

`slack`

Solution values of slack variables.

`rowDual`

Dual values of constraints.

`redCost`

Reduced cost of variables.

### COPT_GetBasis

**Synopsis**

```
int COPT_GetBasis(copt_prob *prob, int *colBasis, int *rowBasis)
```

**Description**

Obtains LP basis.

**Arguments**

`prob`

The COPT problem.

`colBasis` and `rowBasis`

The basis status of variables and constraints. Please refer to basis constants for possible values and their meanings.

### COPT_SetBasis

**Synopsis**

```
int COPT_SetBasis(copt_prob *prob, const int *colBasis, const int
*rowBasis)
```

**Description**

Sets LP basis. It can be used to warm-start an LP optimization.

**Arguments**

`prob`

The COPT problem.

`colBasis` and `rowBasis`

The basis status of variables and constraints. Please refer to basis constants for possible values and their meanings.

### COPT_SetSlackBasis

**Synopsis**

```
int COPT_SetSlackBasis(copt_prob *prob)
```

**Description**

Sets a slack basis for LP. The slack basis is the default starting basis for an LP problem. This API function can be used to restore an LP problem to its starting basis.

**Arguments**

`prob`

The COPT problem.

### COPT_FSolve

**Synopsis**

```
int COPT_FSolve(copt_prob *prob, double *pVal, int *pInd, int *pLen)
```

**Description**

Given a sparse vector, performs a forward solve. The result is stored in place in the same vector. This is an advanced simplex routine intended for expert users.

**Arguments**

`prob`

The COPT problem.

`pVal`

Pointer to the values of the sparse vector. On input, the values of the input vector; on output, the values of the result, up to row size.

`pInd`

Pointer to the indices of the sparse vector. On input, the indices of the input vector; on output, the indices of the result, up to row size.

`pLen`

Pointer to the length of the sparse vector. On input, the length of the input vector; on output, the length of the result.

### COPT_BSolve

**Synopsis**

```
int COPT_BSolve(copt_prob *prob, double *pVal, int *pInd, int *pLen)
```

**Description**

Given a sparse vector, performs a backward solve. The result is stored in place in the same vector. This is an advanced simplex routine intended for expert users.

**Arguments**

`prob`

The COPT problem.

`pVal`

Pointer to the values of the sparse vector. On input, the values of the input vector; on output, the values of the result, up to row size.

`pInd`

Pointer to the indices of the sparse vector. On input, the indices of the input vector; on output, the indices of the result, up to row size.

`pLen`

Pointer to the length of the sparse vector. On input, the length of the input vector; on output, the length of the result.

### COPT_GetBasisHead

**Synopsis**

```
int COPT_GetBasisHead(copt_prob *prob, int *bHead)
```

**Description**

Gets the indices of the variables that make up the current basis.

**Arguments**

`prob`

The COPT problem.

`bHead`

Pointer to the output array storing the basis head indices. The array must be allocated by the caller, with a capacity of at least the number of rows in the model.

### COPT_ComputeShadowPriceRange

**Synopsis**

```
int COPT_ComputeShadowPriceRange(
    copt_prob *prob,
    int iRow,
    int ifRhs,
    double *dShadowPriceLow,
    double *dShadowPriceUp)
```

**Description**

Computes the shadow price range for a given constraint.

**Arguments**

`prob`

The COPT problem.

`iRow`

The constraint (row) index.

`ifRhs`

For a range constraint, whether to use the RHS for computing the shadow price.

`dShadowPriceLow`

Pointer to the output lower bound of the shadow price.

`dShadowPriceUp`

Pointer to the output upper bound of the shadow price.

### COPT_Reset

**Synopsis**

```
int COPT_Reset(copt_prob *prob, int iClearAll)
```

**Description**

Reset basis and LP/MIP solution in problem, which forces next solve start from scratch. If `iClearAll` is `1`, then clear additional information such as MIP start as well.

**Arguments**

`prob`

The COPT problem.

`iClearAll`

Whether to clear additional information.

### COPT_ReadSol

**Synopsis**

```
int COPT_ReadSol(copt_prob *prob, const char *solfilename)
```

**Description**

Reads a MIP solution from file as MIP start information.

**Note:** The default solution value is 0, i.e. a partial solution will be automatically filled in with zeros.

**Arguments**

`prob`

The COPT problem.

`solfilename`

The path to the solution file.

### COPT_ReadJsonSol

**Synopsis**

```
int COPT_ReadJsonSol(copt_prob *prob, const char *solfilename)
```

**Description**

Read solution in format of JSON from file.

**Arguments**

`prob`

The COPT problem.

`solfilename`

The path to the solution file.

### COPT_WriteSol

**Synopsis**

```
int COPT_WriteSol(copt_prob *prob, const char *solfilename)
```

**Description**

Writes a LP/MIP solution to a file.

**Arguments**

`prob`

The COPT problem.

`solfilename`

The path to the solution file.

### COPT_WriteJsonSol

**Synopsis**

```
int COPT_WriteJsonSol(copt_prob *prob, const char *solfilename)
```

**Description**

Writes solution to a file of type `".json"`.

**Arguments**

`prob`

The COPT problem.

`solfilename`

The path to the solution file.

### COPT_WritePoolSol

**Synopsis**

```
int COPT_WritePoolSol(copt_prob *prob, int iSol, const char
*solfilename)
```

**Description**

Writes selected pool solution to a file.

**Arguments**

`prob`

The COPT problem.

`iSol`

Index of pool solution.

`solfilename`

The path to the solution file.

### COPT_WriteBasis

**Synopsis**

```
int COPT_WriteBasis(copt_prob *prob, const char *basfilename)
```

**Description**

Writes the internal LP basis to a file.

**Arguments**

`prob`

The COPT problem.

`basfilename`

The path to the basis file.

### COPT_ReadBasis

**Synopsis**

```
int COPT_ReadBasis(copt_prob *prob, const char *basfilename)
```

**Description**

Reads the LP basis from a file. It can be used to warm-start an LP optimization.

**Arguments**

`prob`

The COPT problem.

`basfilename`

The path to the basis file.

### 23.6.5 Accessing information of problem

#### COPT_GetCols

**Synopsis**

```
int COPT_GetCols(
    copt_prob *prob,
    int nCol,
    const int *list,
    int *colMatBeg,
    int *colMatCnt,
    int *colMatIdx,
    double *colMatElem,
    int nElemSize,
    int *pReqSize)
```

**Description**

Extract coefficient matrix by columns.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `colMatBeg`, `colMatCnt`, `colMatIdx` and `colMatElem`, we get number of non-zeros elements by `pReqSize` specified by `nCol` and `list`. Secondly, allocate sufficient memory for CCS-format matrix and call this function again to extract coefficient matrix. If the memory of coefficient matrix passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nCol` columns will be returned.

**Arguments**

`prob`

The COPT problem.

`nCol`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colMatBeg`, `colMatCnt`, `colMatIdx` and `colMatElem`

Defines the coefficient matrix in compressed column storage (CCS) format. Please see **other information** of `COPT_LoadProb` for an example of the CCS format.

`nElemSize`

Length of array for non-zero coefficients.

`pReqSize`

Pointer to minimal length of array for non-zero coefficients. Can be `NULL`.

### COPT_GetPSDCols

**Synopsis**

```
int COPT_GetPSDCols(copt_prob *prob, int nCol, int *list, int*
colDims, int *colLens)
```

**Description**

Get dimension and flattened length of `nCol` PSD variables.

**Arguments**

`prob`

The COPT problem.

`nCol`

Number of PSD variables.

`list`

Index of PSD variables. Can be `NULL`.

`colDims`

Dimension of PSD variables.

`colLens`

Flattened length of PSD variables.

### COPT_GetRows

**Synopsis**

```
int COPT_GetRows(
    copt_prob *prob,
    int nRow,
    const int *list,
    int *rowMatBeg,
    int *rowMatCnt,
    int *rowMatIdx,
    double *rowMatElem,
    int nElemSize,
    int *pReqSize)
```

**Description**

Extract coefficient matrix by rows.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`, we get number of non-zeros elements by `pReqSize` specified by `nRow` and `list`. Secondly, allocate sufficient memory for CRS-format matrix and call this function again to extract coefficient matrix. If the memory of coefficient matrix passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nRow` rows will be returned.

**Arguments**

`prob`

The COPT problem.

`nRow`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. Please see **other information** of `COPT_LoadProb` for an example of the CRS format.

`nElemSize`

Length of array for non-zero coefficients.

`pReqSize`

Pointer to minimal length of array for non-zero coefficients. Can be `NULL`.

### COPT_GetElem

**Synopsis**

```
int COPT_GetElem(copt_prob *prob, int iCol, int iRow, double
*p_elem)
```

**Description**

Get coefficient of specified row and column.

**Arguments**

`prob`

The COPT problem.

`iCol`

Column index.

`iRow`

Row index.

`p_elem`

Pointer to requested coefficient.

### COPT_GetPSDElem

**Synopsis**

```
int COPT_GetPSDElem(copt_prob *prob, int iCol, int iRow, int *p_idx)
```

**Description**

Get symmetric matrix index of specified PSD constraint and PSD variable.

**Arguments**

`prob`

The COPT problem.

`iCol`

PSD variable index.

`iRow`

PSD constraint index.

`p_idx`

Pointer to requested symmetric matrix index.

### COPT_GetLMIElem

**Synopsis**

```
int COPT_GetLMIElem(copt_prob *prob, int iCol, int iRow, int *p_idx)
```

**Description**

Get symmetric matrix index of specified LMI constraint and scalar variable.

**Arguments**

`prob`

The COPT problem.

`iCol`

Scalar variable index.

`iRow`

LMI constraint index.

`p_idx`

Pointer to requested coefficient matrix index.

### COPT_GetSymMat

**Synopsis**

```
int COPT_GetSymMat(
    copt_prob *prob,
    int iMat,
    int *p_nDim,
    int *p_nElem,
    int *rows,
    int *cols,
    double *elems)
```

**Description**

Get specified symmetric matrix.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `rows`, `cols` and `elems`, we get dimension and number of non-zeros of symmetric matrix by `p_nDim` and `p_nElem`, then allocate enough memory for `rows`, `cols` and `elems` and call this function to get the data of symmetric matrix.

**Arguments**

`prob`

The COPT problem.

`iMat`

Symmetric matrix index.

`p_nDim`

Pointer to dimension of symmetric matrix.

`p_nElem`

Pointer to number of nonzeros of symmetric matrix.

`rows`

Row index of symmetric matrix.

`cols`

Column index of symmetric matrix.

`elems`

Nonzero elements of symmetric matrix.

### COPT_GetQuadObj

**Synopsis**

```
int COPT_GetQuadObj(copt_prob* prob, int* p_nQElem, int* qRow, int*
qCol, double* qElem)
```

**Description**

Get the quadratic terms of the quadratic objective function.

**Arguments**

`prob`

The COPT problem.

`p_nQElem`

Pointer to number of non-zero quadratic terms . Can be `NULL`.

`qRow`

Row index of non-zero quadratic terms of the quadratic objective function.

`qCol`

Column index of non-zero quadratic terms of the quadratic objective function.

`qElem`

Values of non-zero quadratic terms of the quadratic objective function.

### COPT_GetPSDObj

**Synopsis**

```
int COPT_GetPSDObj(copt_prob *prob, int iCol, int *p_idx)
```

**Description**

Get the specified PSD term of objective function.

**Arguments**

`prob`

The COPT problem.

`iCol`

PSD variable index.

`p_idx`

Pointer to symmetric matrix index.

### COPT_GetNLObj

**Synopsis**

```
int COPT_GetNLObj(
    copt_prob *prob,
    int *p_nToken,
    int *p_nTokenElem,
    int *token,
    double *tokenElem)
```

**Description**

Retrieve the nonlinear expression terms in the objective function.

**Arguments**

`prob`

The COPT problem.

`p_nToken`

Pointer to the number of tokens in the expression.

`p_nTokenElem`

Pointer to the number of constants in the expression.

`token`

Array of tokens in the expression.

`tokenElem`

Array of constants in the expression.

### COPT_GetSOSs

#### Synopsis

```
int COPT_GetSOSs(
    copt_prob *prob,
    int nSos,
    const int *list,
    int *sosMatBeg,
    int *sosMatCnt,
    int *sosMatIdx,
    double *sosMatWt,
    int nElemSize,
    int *pReqSize)
```

#### Description

Get the weight matrix of SOS constraints.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `sosMatBeg`, `sosMatCnt`, `sosMatIdx` and `sosMatWt`, we get number of non-zeros elements by `pReqSize` specified by `nSos` and `list`. Secondly, allocate sufficient memory for CRS-format matrix and call this function again to extract weight matrix. If the memory of weight matrix passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nSos` rows will be returned.

#### Arguments

`prob`

The COPT problem.

`nSos`

Number of SOS constraints.

`list`

Index of SOS constraints. Can be `NULL`.

`sosMatBeg`, `sosMatCnt`, `sosMatIdx` and `sosMatWt`

Defines the weight matrix of SOS constraints in compressed row storage (CRS) format. Please see **other information** of `COPT_LoadProb` for an example of the CRS format.

`nElemSize`

Length of array for non-zero weights.

`pReqSize`

Pointer to minimal length of array for non-zero weights. Can be `NULL`.

### COPT_GetCones

#### Synopsis

```
int COPT_GetCones(
    copt_prob *prob,
    int nCone,
    const int *list,
    int *coneBeg,
    int *coneCnt,
    int *coneIdx,
    int nElemSize,
    int *pReqSize)
```

#### Description

Get the matrix of Second-Order-Cone (SOC) constraints.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `coneBeg`, `coneCnt` and `coneIdx`, we get number of subscripts of variables by `pReqSize` specified by `nCone` and `list`. Secondly, allocate sufficient memory for CRS-format matrix and call this function again to extract weight matrix. If the memory of weight matrix passed to function is not sufficient, then return the first `nElemSize` subscripts of variables, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nCone` rows will be returned.

#### Arguments

`prob`

The COPT problem.

`nCone`

Number of SOC constraints.

`list`

Index of SOC constraints. Can be `NULL`.

`coneBeg, coneCnt, coneIdx`

Defines the matrix of SOC constraints in compressed row storage (CRS) format. Please see **other information** of `COPT_LoadProb` for an example of the CRS format.

`nElemSize`

Length of array for non-zero weights.

`pReqSize`

Pointer to minimal length of array for non-zero weights. Can be `NULL`.

### COPT_GetExpCones

#### Synopsis

```
int COPT_GetExpCones(
    copt_prob *prob,
    int nCone,
    const int *list,
    int *coneType,
    int *coneIdx,
    int nElemSize,
    int *pReqSize)
```

#### Description

Get the array of exponential cone constraints.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `coneIdx`, we get number of subscripts of variables by `nElemSizes` specified by `nCone` and `list`. Secondly, allocate sufficient memory for array and call this function again to extract weight array. If the memory of weight array passed to function is not sufficient, then return the first `nElemSize` subscripts, and the minimal required length of subscripts by `pReqSize`. If `list` is `NULL`, then the first `nCone` rows will be returned.

#### Arguments

`prob`

The COPT problem.

`nCone`

Number of exponential cone constraints.

`list`

Index of exponential cone constraints. Can be `NULL`.

`coneType`

Type of exponential cone constraints. Please refer to *Exponential Cone type* for possible values.

`coneIdx`

Array of subscripts of variables constituting the exponential cone constraints.

`nElemSize`

Length of array for subscripts of variables.

`pReqSize`

Pointer to minimal length of array for subscripts of variables. Can be `NULL`.

### COPT_GetAffineCone

#### Synopsis

```
int COPT_GetAffineCone(
    copt_prob *prob,
    int affConeIdx,
    int *coneType,
    int *nConeDim,
    int *nAlphaDim,
    double *alphaElem,
    int *psdBeg,
    int *psdCnt,
    int *psdColIdx,
    int *psdMatIdx,
    int nPsdElemSize,
    int *pPsdReqSize,
    int *rowMatBeg,
    int *rowMatCnt,
    int *rowMatIdx,
    double *rowMatElem,
    double *rowConst,
    int nElemSize,
    int *pReqSize)
```

#### Description

Retrieve the array of affine cone constraints from the model.

Generally, the user needs to call this function twice to complete the extraction of the affine cone constraint.

First, pass `NULL` to `affConeIdx` to obtain the number of indices for the affine cone constraint array specified by `nConeDim` and `list` through `pReqSize`. Then, allocate appropriate space for the array parameters, and call the function again to retrieve the specified affine cone constraint array.

If the length of the affine cone constraint array passed in is insufficient, the function will return the first `nElemSize` variable index arrays and return the minimum required array length through `pReqSize`. If `list` is `NULL`, the function returns the affine cone constraint array corresponding to the first `nConeDim` rows.

#### Arguments

`prob`

The COPT problem.

`affConeIdx`

The index of the affine cone.

`coneType`

The type of the affine cone. Please refer to *SOC constraint types* and *Exponential Cone type* for possible values.

`nConeDim`

The dimension of the affine cone.

`nAlphaDim`

Reserved parameter, currently not in use.

`alphaElem`

Reserved parameter, currently not in use.

`psdBeg, psdCnt, psdColIdx, psdMatIdx`

Represents the PSD terms in the affine cone.

`psdBeg` indicates the starting position of the PSD terms in each affine cone term, `psdCnt` specifies the number of PSD terms, `psdColIdx` indicates the index of the PSD variable, and `psdMatIdx` refers to the index of the symmetric matrix.

First, the number of PSD terms is returned via `pPsdReqSize`, and `nPsdElemSize` represents the number of PSD terms to be retrieved.

`rowMatBeg, rowMatCnt, rowMatIdx, rowMatElem`

Represents the linear terms in the affine cone.

The coefficient matrix is provided in CRS-format. For detailed examples, please refer to **Additional Information** in `COPT_LoadProb`.

`rowConst`

The constant term in the affine cone.

`nElemSize`

Length of array for subscripts of variables.

`pReqSize`

Pointer to minimal length of array for subscripts of variables of the affine cone. Can be `NULL`.

### COPT_GetQConstr

**Synopsis**

```
int COPT_GetQConstr(
    copt_prob *prob,
    int qConstrIdx,
    int *qMatRow,
    int *qMatCol,
    double *qMatElem,
    int nQElemSize,
    int *pQReqSize,
    int *rowMatIdx,
    double *rowMatElem,
    char *cRowSense,
```

```
    double *dRowBound,
    int nElemSize,
    int *pReqSize)
```

**Description**

Get quadratic constraint.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `qMatRow`, `qMatCol`, `qMatElem`, `rowMatIdx` and `rowMatElem`, we get number of non-zero quadratic terms by `pQReqSize` and number of non-zero linear terms by `pReqSize` specified by `qConstrIdx`. Secondly, allocate sufficient memory for the quadratic terms and the linear terms, and call this function again to extract the quadratic constraint. If the memory of the array of the quadratic terms passed to function is not sufficient, then return the first `nQElemSize` quadratic terms, and the minimal required length of quadratic terms by `pQReqSize`. If the memory of the array of the linear terms passed to function is not sufficient, then return the first `nElemSize` linear terms, and the minimal required length of linear terms by `pReqSize`.

**Arguments**

`prob`

The COPT problem.

`qConstrIdx`

Index of the quadratic constraint.

`qMatRow`

Row index of non-zero quadratic terms of the quadratic constraint (row).

`qMatCol`

Column index of non-zero quadratic terms of the quadratic constraint (row).

`qMatElem`

Values of non-zero quadratic terms of the quadratic constraint (row).

`nQElemSize`

Length of array for non-zero quadratic terms of the quadratic constraint (row).

`pQReqSize`

Pointer to minimal length of array for non-zero quadratic terms of the quadratic constraint (row). Can be `NULL`.

`rowMatIdx`

Column index of non-zero linear terms of the quadratic constraint (row).

`rowMatElem`

Values of non-zero linear terms of the quadratic constraint (row).

`cRowSense`

The sense of the quadratic constraint (row).

`dRowBound`

Right hand side of the quadratic constraint (row).

`nElemSize`

Length of array for non-zero linear terms of the quadratic constraint (row).

`pReqSize`

Pointer to minimal length of array for non-zero linear terms of the quadratic constraint (row). Can be `NULL`.

### COPT_GetPSDConstr

**Synopsis**

```
int COPT_GetPSDConstr(
    copt_prob *prob,
    int psdConstrIdx,
    int *psdColIdx,
    int *symMatIdx,
    int nColSize,
    int *pColReqSize,
    int *rowMatIdx,
    double *rowMatElem,
    double *dRowLower,
    double *dRowUpper,
    int nElemSize,
    int *pReqSize)
```

**Description**

Get PSD constraint.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `psdColIdx` and `symMatIdx`, we get number of PSD terms by `pColReqSize` specified by `psdConstrIdx`, by passing `NULL` to arguments `rowMatIdx` and `rowMatElem`, we get number of linear terms by `pReqSize` specified by `qConstrIdx`. Secondly, allocate sufficient memory for the PSD terms and the linear terms, and call this function again to extract the PSD constraint. If the memory of the array of the PSD terms passed to function is not sufficient, then return the first `nColSize` PSD terms, and the minimal required length of PSD terms by `pColReqSize`. If the memory of the array of the linear terms passed to function is not sufficient, then return the first `nElemSize` linear terms, and the minimal required length of linear terms by `pReqSize`.

**Arguments**

`prob`

The COPT problem.

`psdConstrIdx`

PSD constraint index.

`psdColIdx`

PSD variable index.

`symMatIdx`

Symmetric matrix index.

`nColSize`

Length of array for PSD terms of the PSD constraint.

`pColReqSize`

Pointer to minimal length of array for PSD terms of the PSD constraint. Can be `NULL`.

`rowMatIdx`

Column index of non-zero linear terms of the PSD constraint.

`rowMatElem`

Values of non-zero linear terms of the PSD constraint.

`dRowLower`

Pointer to lower bound of the PSD constraint.

`dRowUpper`

Pointer to upper bound of the PSD constraint.

`nElemSize`

Length of array for non-zero linear terms of the PSD constraint.

`pReqSize`

Pointer to minimal length of array for non-zero linear terms of the PSD constraint (row). Can be `NULL`.

### COPT_GetLMIConstr

**Synopsis**

```
int COPT_GetLMIConstr(
    copt_prob *prob,
    int lmiConstrIdx,
    int *nDim,
    int *nLMILen,
    int *colIdx,
    int *symMatIdx,
    int *constMatIdx,
    int nElemSize,
    int *pReqSize)
```

**Description**

Gets the LMI constraint with the specified index in the model.

**Arguments**

`prob`

The COPT problem.

`lmiConstrIdx`

LMI constraint index.

`nDim`

Pointer to the dimension of symmetric matrix in the LMI constraint.

`nLMILen`

Pointer to the flattened length of the LMI constraint.

`colIdx`

Index of scalar variable in the LMI constraint.

`symMatIdx`

Index of symmetric coefficient matrix in the LMI constraint.

`constMatIdx`

Pointer to the index of symmetric constant-term matrix in the LMI constraint.

`nElemSize`

Length of array for non-zero linear terms of the LMI constraint.

`pReqSize`

Pointer to minimal length of array for non-zero linear terms of the LMI constraint (row). Can be `NULL`.

### COPT_GetIndicator

#### Synopsis

```
int COPT_GetIndicator(
    copt_prob *prob,
    int rowIdx,
    int *binColIdx,
    int *binColVal,
    int *nRowMatCnt,
    int *rowMatIdx,
    double *rowMatElem,
    char *cRowSense,
    double *dRowBound,
    int nElemSize,
    int *pReqSize)
```

#### Description

Get the data of an indicator constraint.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `nRowMatCnt`, `rowMatIdx` and `rowMatElem`, we get number of non-zeros elements by `pReqSize` specified by `rowIdx`. Secondly, allocate sufficient memory for sparse row vector and call this function again to extract data.

If the memory of sparse row vector passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`.

#### Arguments

`prob`

The COPT problem.

`rowIdx`

Index of the indicator constraint.

`binColIdx`

Index of the indicator variable (column).

`binColVal`

Value of the indicator variable (column).

`nRowMatCnt`

Number of non-zeros elements in the linear constraint (row).

`rowMatIdx`

Column index of non-zeros elements in the linear constraint (row).

`rowMatElem`

Values of non-zero elements in the linear constraint (row).

`cRowSense`

The sense of the linear constraint (row).

`dRowBound`

Right hand side of the linear constraint (row).

`nElemSize`

Length of array for non-zero coefficients.

`pReqSize`

Pointer to minimal length of array for non-zero coefficients. Can be `NULL`.

### COPT_GetIndicators

**Synopsis**

```
int COPT_GetIndicators(
    copt_prob *prob,
    int nInd,
    int *list,
    int *indType,
    int *binColIdx,
    int *binColVal,
    int *rowMatBeg,
    int *rowMatCnt,
    int *rowMatIdx,
    double *rowMatElem,
    char *cRowSense,
    double *dRowBound,
    int nElemSize,
    int *pReqSize)
```

**Description**

Get the data of a set of indicator constraints.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `rowMatBeg`, `rowMatCnt`, `rowMatIdx` , we get number of non-zeros elements by `pReqSize` specified by `nInd` and `list`. Secondly, allocate sufficient memory for CRS-format matrix and call this function again to extract coefficient matrix.

If the memory of coefficient matrix passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nInd` rows will be returned.

**Arguments**

`prob`

The COPT problem.

`nInd`

Number of indicator constraints (rows).

`list`

Index of indicator constraints. Can be `NULL`.

`indType`

Type of indicator constraints. Please refer to *Indicator constraint types* for possible values.

`binColIdx`

Index of the indicator variable (column).

`binColVal`

Value of the indicator variable (column).

`rowMatBeg, rowMatCnt, rowMatIdx` and `rowMatElem`

Defines the coefficient matrix of indicator constraints in compressed row storage (CRS) format. Please see **other information** of `COPT_LoadProb` for an example of the CRS format.

`cRowSense`

The sense of the linear constraint (row).

`dRowBound`

Right hand side of the linear constraint (row).

`nElemSize`

Length of array for non-zero coefficients.

`pReqSize`

Pointer to minimal length of array for non-zero coefficients. Can be `NULL`.

### COPT_GetNLConstr

**Synopsis**

```
int COPT_GetNLConstr(
    copt_prob *prob,
    int nlConstrIdx,
    int *token,
    double *tokenElem,
    int nToken,
    int nTokenElem,
    int *pReqToken,
    int *pReqTokenElem,
    int *rowMatIdx,
    double *rowMatElem,
    double *dRowLower,
    double *dRowUpper,
    int nElemSize,
    int *pReqElemSize)
```

**Description**

Retrieve the nonlinear expression constraint at the specified index.

**Arguments**

`prob`

The COPT problem.

`nlConstrIdx`

Index of the constraint.

`token`

Array of tokens in the expression.

`tokenElem`

Array of constants in the expression.

`nToken`

Number of tokens in the expression.

`nTokenElem`

Number of constants in the expression.

`pReqToken`

Pointer to the number of tokens in the expression.

`pReqTokenElem`

Pointer to the number of constants in the expression.

`rowMatIdx`

Indices of linear terms in the constraint.

`rowMatElem`

Coefficients of linear terms in the constraint.

`dRowLower`

Pointer to the lower bound of the constraint.

`dRowUpper`

Pointer to the upper bound of the constraint.

`nElemSize`

Number of linear terms in the constraint.

`pReqElemSize`

Pointer to the number of linear terms.

### COPT_GetColIdx

**Synopsis**

```
int COPT_GetColIdx(copt_prob *prob, const char *colName, int
*p_iCol)
```

**Description**

Get index of column by name.

**Arguments**

`prob`

The COPT problem.

`colName`

Name of column.

`p_iCol`

Pointer to index of column.

### COPT_GetPSDColIdx

**Synopsis**

```
int COPT_GetPSDColIdx(copt_prob *prob, const char *psdColName, int
*p_iPSDCol)
```

**Description**

Get index of PSD variable by name.

**Arguments**

`prob`

The COPT problem.

`psdColName`

Name of PSD variable.

`p_iPSDCol`

Pointer to index of PSD variable.

### COPT_GetRowIdx

**Synopsis**

```
int COPT_GetRowIdx(copt_prob *prob, const char *rowName, int
*p_iRow)
```

**Description**

Get index of row by name.

**Arguments**

`prob`

The COPT problem.

`rowName`

Name of row.

`p_iRow`

Pointer to index of row.

### COPT_GetQConstrIdx

**Synopsis**

```
int COPT_GetQConstrIdx(copt_prob *prob, const char *qConstrName, int
*p_iQConstr)
```

**Description**

Get index of quadratic constraint by name.

**Arguments**

`prob`

The COPT problem.

`qConstrName`

Name of quadratic constraint.

`p_iQConstr`

Pointer to index of quadratic constraint.

### COPT_GetPSDConstrIdx

**Synopsis**

```
int COPT_GetPSDConstrIdx(copt_prob *prob, const char *psdConstrName,
int *p_iPSDConstr)
```

**Description**

Get index of PSD constraint by name.

**Arguments**

`prob`

The COPT problem.

`psdConstrName`

Name of PSD constraint.

`p_iPSDConstr`

Pointer to index of PSD constraint.

### COPT_GetLMIConstrIdx

**Synopsis**

```
int COPT_GetLMIConstrIdx(copt_prob *prob, const char *lmiConstrName,
int *p_iLMIConstr)
```

**Description**

Get index of LMI constraint by name.

**Arguments**

`prob`

The COPT problem.

`lmiConstrName`

Name of LMI constraint.

`p_iLMIConstr`

Pointer to index of LMI constraint.

### COPT_GetIndicatorIdx

**Synopsis**

```
int COPT_GetIndicatorIdx(copt_prob *prob, const char *indicatorName,
int *p_iIndicator)
```

**Description**

Get index of indicator constraint by name.

**Arguments**

`prob`

The COPT problem.

`indicatorName`

Name of indicator constraint.

`p_iIndicator`

Pointer to index of indicator constraint.

### COPT_GetAffineConeIdx

**Synopsis**

```
int COPT_GetAffineConeIdx(copt_prob *prob, const char *affConeName,
int *p_iAffCone)
```

**Description**

Get index of affine cone by name.

**Arguments**

`prob`

The COPT problem.

`affConeName`

Name of affine cone.

`p_iAffCone`

Pointer to index of affine cone.

### COPT_GetNLConstrIdx

**Synopsis**

```
int COPT_GetNLConstrIdx(copt_prob *prob, const char *nlConstrName,
int *p_iNLConstr)
```

**Description**

Retrieve the index of a nonlinear expression constraint by name.

**Arguments**

`prob`

The COPT problem.

`nlConstrName`

Name of the nonlinear expression constraint.

`p_iNLConstr`

Pointer to the internal index of the nonlinear expression constraint in COPT.

### COPT_GetColInfo

**Synopsis**

```
int COPT_GetColInfo(copt_prob *prob, const char *infoName, int num,
const int *list, double *info)
```

**Description**

Get information of column. If `list` is `NULL`, then information of the first `num` columns will be returned.

**Arguments**

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`info`

Array of information.

### COPT_GetPSDColInfo

**Synopsis**

```
int COPT_GetPSDColInfo(copt_prob *prob, const char *infoName, int
iCol, double *info)
```

**Description**

Get information of PSD variable.

**Arguments**

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`iCol`

Index of PSD variable.

`info`

Array of information.

### COPT_GetRowInfo

**Synopsis**

```
int COPT_GetRowInfo(copt_prob *prob, const char *infoName, int num,
const int *list, double *info)
```

**Description**

Get information of row. If `list` is `NULL`, then information of the first `num` rows will be returned.

**Arguments**

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`num`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`info`

Array of information.

### COPT_GetQConstrInfo

#### Synopsis

```
int COPT_GetQConstrInfo(copt_prob *prob, const char *infoName, int
num, const int *list, double *info)
```

#### Description

Get information of quadratic constraints. If `list` is `NULL`, then information of the first `num` quadratic constraints will be returned.

#### Arguments

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`num`

Number of quadratic constraints.

`list`

Index of quadratic constraints. Can be `NULL`.

`info`

Array of information.

### COPT_GetPSDConstrInfo

#### Synopsis

```
int COPT_GetPSDConstrInfo(copt_prob *prob, const char *infoName, int
num, const int* list, double *info)
```

#### Description

Get information of PSD constraints. If `list` is `NULL`, then information of the first `num` PSD constraints will be returned.

#### Arguments

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`num`

Number of PSD constraints.

`list`

Index of PSD constraints. Can be `NULL`.

`info`

Array of information.

### COPT_GetLMIConstrInfo

**Synopsis**

```
int COPT_GetLMIConstrInfo(copt_prob *prob, const char *infoName, int
iLMI, double *info)
```

**Description**

Get a set of information about LMI constraints.

**Arguments**

`prob`

The COPT problem.

`infoName`

Name of information. Possible values are: `COPT_DBLINFO_SLACK` and `COPT_DBLINFO_DUAL`.

`iLMI`

The index of the LMI constraint whose information is to be retrieved.

`info`

Array of information.

### COPT_GetNLConstrInfo

**Synopsis**

```
int COPT_GetNLConstrInfo(copt_prob *prob, const char *infoName,  int
num, const int *list, double *info)
```

**Description**

Retrieve information about nonlinear expression constraints. If `list` is `NULL`, the function returns information for the first `num` nonlinear expression constraints.

**Arguments**

`prob`

The COPT problem.

`infoName`

Name of the information to retrieve. Currently supported nonlinear expression constraint information is: `"LB"` , `"UB"` and `"Slack"` .

`num`

Number of nonlinear expression constraints to retrieve information for.

`list`

Index list of nonlinear expression constraints to retrieve information for. Can be `NULL`.

`info`

Array to store the retrieved information.

### COPT_GetColType

#### Synopsis

```
int COPT_GetColType(copt_prob *prob, int num, const int *list, char
*type)
```

#### Description

Get types of columns. If `list` is `NULL`, then types of the first `num` columns will be returned.

#### Arguments

`prob`

The COPT problem.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`type`

Types of columns.

### COPT_GetColBasis

#### Synopsis

```
int COPT_GetColBasis(copt_prob *prob, int num, const int *list, int
*colBasis)
```

#### Description

Get basis status of columns. If `list` is `NULL`, then basis status of the first `num` columns will be returned.

#### Arguments

`prob`

The COPT problem.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colBasis`

Basis status of columns.

### COPT_GetRowBasis

#### Synopsis

```
int COPT_GetRowBasis(copt_prob *prob, int num, const int *list, int
*rowBasis)
```

#### Description

Get basis status of rows. If `list` is `NULL`, then basis status of the first `num` rows will be returned.

#### Arguments

`prob`

The COPT problem.

`num`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`rowBasis`

Basis status of rows.

### COPT_GetQConstrSense

#### Synopsis

```
int COPT_GetQConstrSense(copt_prob *prob, int num, const int *list,
char *sense)
```

#### Description

Get senses of quadratic constraints. If `list` is `NULL`, then types of the first `num` quadratic constraints will be returned.

#### Arguments

`prob`

The COPT problem.

`num`

Number of quadratic constraints.

`list`

Index of quadratic constraints. Can be `NULL`.

`sense`

Array of senses.

### COPT_GetQConstrRhs

#### Synopsis

```
int COPT_GetQConstrRhs(copt_prob *prob, int num, const int *list,
double *rhs)
```

#### Description

Get RHS of quadratic constraints. If `list` is `NULL`, then types of the first `num` quadratic constraints will be returned.

#### Arguments

`prob`

The COPT problem.

`num`

Number of quadratic constraints.

`list`

Index of quadratic constraints. Can be `NULL`.

`rhs`

Array of RHS.

### COPT_GetColName

#### Synopsis

```
int COPT_GetColName(copt_prob *prob, int iCol, char *buff, int
buffSize, int *pReqSize)
```

#### Description

Get name of column by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

#### Arguments

`prob`

The COPT problem.

`iCol`

Index of column.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

### COPT_GetPSDColName

#### Synopsis

```
int COPT_GetPSDColName(copt_prob *prob, int iPSDCol, char *buff, int
buffSize, int *pReqSize)
```

#### Description

Get name of PSD variable by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

#### Arguments

`prob`

The COPT problem.

`iPSDCol`

Index of PSD variable.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

### COPT_GetRowName

#### Synopsis

```
int COPT_GetRowName(copt_prob *prob, int iRow, char *buff, int
buffSize, int *pReqSize)
```

#### Description

Get name of row by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

#### Arguments

`prob`

The COPT problem.

`iRow`

Index of row.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

### COPT_GetQConstrName

#### Synopsis

```
int COPT_GetQConstrName(copt_prob *prob, int iQConstr, char *buff,
int buffSize, int *pReqSize)
```

#### Description

Get name of quadratic constraint by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

#### Arguments

`prob`

The COPT problem.

`iQConstr`

Index of quadratic constraint.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

### COPT_GetPSDConstrName

#### Synopsis

```
int COPT_GetPSDConstrName(copt_prob *prob, int iPSDConstr, char
*buff, int buffSize, int *pReqSize)
```

#### Description

Get name of PSD constraint by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

#### Arguments

`prob`

The COPT problem.

`iPSDConstr`

Index of PSD constraint.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

### COPT_GetLMIConstrName

**Synopsis**

```
int COPT_CALL COPT_GetLMIConstrName(copt_prob *prob, int iLMIConstr,
char *buff, int buffSize, int *pReqSize)
```

**Description**

Get name of LMI constraint by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

**Arguments**

`prob`

The COPT problem.

`iLMIConstr`

Index of LMI constraint.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

### COPT_GetIndicatorName

**Synopsis**

```
int COPT_GetIndicatorName(copt_prob *prob, int iIndicator, char
*buff, int buffSize, int *pReqSize)
```

**Description**

Get name of indicator constraints by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

**Arguments**

`prob`

The COPT problem.

`iIndicator`

Index of indicator constraint.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

### COPT_GetAffineConeName

#### Synopsis

```
int COPT_GetAffineConeName(copt_prob *prob, int iAffCone, char
*buff, int buffSize, int *pReqSize)
```

#### Description

Get name of affine cone by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

#### Arguments

`prob`

The COPT problem.

`iAffCone`

Index of affine cone.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

### COPT_GetNLConstrName

#### Synopsis

```
int COPT_GetNLConstrName(copt_prob *prob, int iNLConstr, char *buff,
int buffSize, int *pReqSize)
```

#### Description

Retrieve the name of nonlinear constraint by index. If the provided `buff` length is insufficient, the function returns a substring of length `buffSize`, and the full required string length is returned via `pReqSize`.

If `buff` is `NULL`, the function returns the required string length via `pReqSize`.

#### Arguments

`prob`

The COPT problem.

`iNLConstr`

Index of the nonlinear expression constraint.

`buff`

Array to store the retrieved name string.

`buffSize`

Size of the provided array.

`pReqSize`

Pointer to store the minimum required string length for the full name. Can be `NULL`.

#### COPT_GetLMIConstrRhs

**Synopsis**

```
int COPT_GetLMIConstrRhs(copt_prob *prob, int num, const int *list,
int *constMatIdx)
```

**Description**

Get the constant-term symmetric matrix of `num` LMI constraints.

**Arguments**

`prob`

The COPT problem.

`num`

Number of LMI constraints.

`list`

Index of LMI constraints.

`constMatIdx`

Index of constant-term symmetric in the LMI constraints.

### 23.6.6 Accessing and setting parameters

#### COPT_SetIntParam

**Synopsis**

```
int COPT_SetIntParam(copt_prob *prob, const char *paramName, int
intParam)
```

**Description**

Sets an integer parameter.

**Arguments**

`prob`

The COPT problem.

`paramName`

The name of the integer parameter.

`intParam`

The value of the integer parameter.

### COPT_GetIntParam, COPT_GetIntParamDef/Min/Max

#### Synopsis

```
int COPT_GetIntParam(copt_prob *prob, const char *paramName, int
*p_intParam)
```

```
int COPT_GetIntParamDef(copt_prob *prob, const char *paramName, int
*p_intParam)
```

```
int COPT_GetIntParamMin(copt_prob *prob, const char *paramName, int
*p_intParam)
```

```
int COPT_GetIntParamMax(copt_prob *prob, const char *paramName, int
*p_intParam)
```

#### Description

Gets the

current

default

minimal

maximal

value of an integer parameter.

#### Arguments

`prob`

The COPT problem.

`paramName`

The name of the integer parameter.

`p_intParam`

Pointer to the value of the integer parameter.

### COPT_SetDblParam

#### Synopsis

```
int COPT_SetDblParam(copt_prob *prob, const char *paramName, double
dblParam)
```

#### Description

Sets a double parameter.

#### Arguments

`prob`

The COPT problem.

`paramName`

The name of the double parameter.

`dblParam`

The value of the double parameter.

### COPT_GetDblParam, COPT_GetDblParamDef/Min/Max

#### Synopsis

```
int COPT_GetDblParam(copt_prob *prob, const char *paramName, double
*p_dblParam)
```

```
int COPT_GetDblParamDef(copt_prob *prob, const char *paramName,
double *p_dblParam)
```

```
int COPT_GetDblParamMin(copt_prob *prob, const char *paramName,
double *p_dblParam)
```

```
int COPT_GetDblParamMax(copt_prob *prob, const char *paramName,
double *p_dblParam)
```

#### Description

Gets the

current

default

minimal

maximal

value of a double parameter.

#### Arguments

`prob`

The COPT problem.

`paramName`

The name of the double parameter.

`p_dblParam`

Pointer to the value of the double parameter.

### COPT_ResetParam

#### Synopsis

```
int COPT_ResetParam(copt_prob *prob)
```

#### Description

Reset parameters to default settings.

#### Arguments

`prob`

The COPT problem.

### COPT_WriteParam

**Synopsis**

```
int COPT_WriteParam(copt_prob *prob, const char *parfilename)
```

**Description**

Writes user defined parameters to a file. This API function will write out all the parameters that are different from their default values.

**Arguments**

`prob`

The COPT problem.

`parfilename`

The path to the parameter file.

### COPT_WriteParamStr

**Synopsis**

```
int COPT_WriteParamStr(copt_prob *prob, char *str, int nStrSize, int
*pReqSize)
```

**Description**

Writes the modified parameters to a string buffer.

**Arguments**

`prob`

The COPT problem.

`str`

String buffer of modified parameters.

`nStrSize`

The size of string buffer.

`pReqSize`

Minimum space requirement of string buffer for modified parameters.

### COPT_ReadParam

**Synopsis**

```
int COPT_ReadParam(copt_prob *prob, const char *parfilename)
```

**Description**

Reads and applies parameters settings as defined in the parameter file.

**Arguments**

`prob`

The COPT problem.

`parfilename`

The path to the parameter file.

#### COPT_ReadParamStr

**Synopsis**

```
int COPT_ReadParamStr(copt_prob *prob, const char *strParam)
```

**Description**

Read parameter settings from string buffer, and set parameters in COPT.

**Arguments**

`prob`

The COPT problem.

`strParam`

String buffer of parameter settings.

### 23.6.7 Accessing attributes

#### COPT_GetIntAttr

**Synopsis**

```
int COPT_GetIntAttr(copt_prob *prob, const char *attrName, int
*p_intAttr)
```

**Description**

Gets the value of an integer attribute.

**Arguments**

`prob`

The COPT problem.

`attrName`

The name of the integer attribute.

`p_intAttr`

Pointer to the value of the integer attribute.

#### COPT_GetDblAttr

**Synopsis**

```
int COPT_GetDblAttr(copt_prob *prob, const char *attrName, int
*p_dblAttr)
```

**Description**

Gets the value of an double attribute.

**Arguments**

`prob`

The COPT problem.

`attrName`

The name of the double attribute.

`p_dblAttr`

Pointer to the value of the double attribute.

### 23.6.8 Logging utilities

#### COPT_SetLogFile

**Synopsis**

```
int COPT_SetLogFile(copt_prob *prob, char *logfilename)
```

**Description**

Set log file for the problem.

**Arguments**

`prob`

The COPT problem.

`logfilename`

The path to the log file.

#### COPT_SetLogCallback

**Synopsis**

```
int COPT_SetLogCallback(copt_prob *prob, void (*logcb)(char *msg,
void *userdata), void *userdata)
```

**Description**

Set message callback for the problem.

**Arguments**

`prob`

The COPT problem.

`logcb`

Callback function for message.

`userdata`

User defined data. The data will be passed to the solver without modification.

### 23.6.9 MIP start utilities

#### COPT_AddMipStart

**Synopsis**

```
int COPT_AddMipStart(copt_prob *prob, int num, const int *list,
double *colVal)
```

**Description**

Add MIP start information for the problem. If `list` is `NULL`, then information of the first `num` columns will be added.

One MIP start information will be added for each call to this function.

**Arguments**

`prob`

The COPT problem.

`num`

Number of variables (columns).

`list`

Index of variables (columns). Can be `NULL`.

`colVal`

MIP start information.

#### COPT_ReadMst

**Synopsis**

```
int COPT_ReadMst(copt_prob *prob, const char *mstfilename)
```

**Description**

Read MIP start information from file, and used as initial solution for the problem.

**Arguments**

`prob`

The COPT problem.

`mstfilename`

The path to the MIP start file.

#### COPT_WriteMst

**Synopsis**

```
int COPT_WriteMst(copt_prob *prob, const char *mstfilename)
```

**Description**

Write solution or existed MIP start information in problem to file.

**Arguments**

`prob`

The COPT problem.

`mstfilename`

The path to the MIP start file.

### 23.6.10 Nonlinear start point utilities

#### COPT_SetNLPrimalStart

**Synopsis**

```
int COPT_SetNLPrimalStart(copt_prob *prob, int num, const int *list,
double *colVal)
```

**Description**

Set nonlinear primal start information for the problem. If `list` is `NULL`, then information of the first `num` columns will be set.

**Arguments**

`prob`

The COPT problem.

`num`

Number of variables (columns).

`list`

Index of variables (columns). Can be `NULL`.

`colVal`

Nonlinear primal start information.

### 23.6.11 IIS utilities

#### COPT_ComputeIIS

**Synopsis**

```
int COPT_ComputeIIS(copt_prob *prob)
```

**Description**

Compute IIS (Irreducible Inconsistent Subsystem) for infeasible problem.

**Arguments**

`prob`

The COPT problem.

#### COPT_WriteIIS

**Synopsis**

```
int COPT_WriteIIS(copt_prob *prob, const char *iisfilename)
```

**Description**

Writes the computed IIS model to a file.

**Arguments**

`prob`

The COPT problem.

`iisfilename`

The path to the IIS model file.

#### COPT_GetColLowerIIS

**Synopsis**

```
int COPT_GetColLowerIIS(copt_prob *prob, int num, const int *list,
int *colLowerIIS)
```

**Description**

Get IIS status of lower bounds of columns. If `list` is `NULL`, then IIS status of the first `num` columns will be returned.

**Arguments**

`prob`

The COPT problem.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colLowerIIS`

IIS status of lower bounds of columns.

### COPT_GetColUpperIIS

**Synopsis**

```
int COPT_GetColUpperIIS(copt_prob *prob, int num, const int *list,
int *colUpperIIS)
```

**Description**

Get IIS status of upper bounds of columns. If `list` is `NULL`, then IIS status of the first `num` columns will be returned.

**Arguments**

`prob`

The COPT problem.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colUpperIIS`

IIS status of upper bounds of columns.

### COPT_GetRowLowerIIS

**Synopsis**

```
int COPT_GetRowLowerIIS(copt_prob *prob, int num, const int *list,
int *rowLowerIIS)
```

**Description**

Get IIS status of lower bounds of rows. If `list` is `NULL`, then IIS status of the first `num` rows will be returned.

**Arguments**

`prob`

The COPT problem.

`num`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`rowLowerIIS`

IIS status of lower bounds of rows.

### COPT_GetRowUpperIIS

**Synopsis**

```
int COPT_GetRowUpperIIS(copt_prob *prob, int num, const int *list,
int *rowUpperIIS)
```

**Description**

Get IIS status of upper bounds of rows. If `list` is `NULL`, then IIS status of the first `num` rows will be returned.

**Arguments**

`prob`

The COPT problem.

`num`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`rowUpperIIS`

IIS status of upper bounds of rows.

### COPT_GetSOSIIS

**Synopsis**

```
int COPT_GetSOSIIS(copt_prob *prob, int num, const int *list, int
*sosIIS)
```

**Description**

Get IIS status of SOS constraints. If `list` is `NULL`, then IIS status of the first `num` SOS constraints will be returned.

**Arguments**

`prob`

The COPT problem.

`num`

Number of SOS constraints.

`list`

Index of SOS constraints. Can be `NULL`.

`sosIIS`

IIS status of SOS constraints.

#### COPT_GetIndicatorIIS

**Synopsis**

```
int COPT_GetIndicatorIIS(copt_prob *prob, int num, const int *list,
int *indicatorIIS)
```

**Description**

Get IIS status of indicator constraints. If `list` is `NULL`, then IIS status of the first `num` indicator constraints will be returned.

**Arguments**

`prob`

The COPT problem.

`num`

Number of indicator constraints.

`list`

Index of indicator constraints. Can be `NULL`.

`indicatorIIS`

IIS status of indicator constraints.

### 23.6.12 Feasibility relaxation utilities

#### COPT_FeasRelax

**Synopsis**

```
int COPT_FeasRelax(copt_prob *prob, double *colLowPen, double
*colUppPen, double *rowBndPen, double *rowUppPen)
```

**Description**

Compute feasibility relaxation to infeasible problem.

**Arguments**

`prob`

The COPT problem.

`colLowPen`

Penalty for lower bounds of columns. If `NULL`, then no relaxation for lower bounds of columns are allowed; If penalty in `colLowPen` is `COPT_INFINITY`, then no relaxation is allowed for corresponding bound.

`colUppPen`

Penalty for upper bounds of columns. If `NULL`, then no relaxation for upper bounds of columns are allowed; If penalty in `colUppen` is `COPT_INFINITY`, then no relaxation is allowed for corresponding bound.

`rowBndPen`

Penalty for bounds of rows. If `NULL`, then no relaxation for bounds of rows are allowed; If penalty in `rowBndPen` is `COPT_INFINITY`, then no relaxation is allowed for corresponding bound.

`rowUppPen`

Penalty for upper bounds of rows. If problem has two-sided rows, and `rowUppPen` is not `NULL`, then `rowUppPen` is penalty for upper bounds of rows; If penalty in `rowUppPen` is `COPT_INFINITY`, then no relaxation is allowed for corresponding bound.

**Note:** Normally, just set `rowUppPen` to `NULL`.

#### COPT_WriteRelax

**Synopsis**

```
int COPT_WriteRelax(copt_prob *prob, const char *relaxfilename)
```

**Description**

Write feasrelax problem to file.

**Arguments**

`prob`

The COPT problem.

`relaxfilename`

Name of feasrelax problem file.

### 23.6.13 Parameter tuning utilities

#### COPT_Tune

**Synopsis**

```
int COPT_Tune(copt_prob *prob)
```

**Description**

Parameter tuning of the model.

**Arguments**

`prob`

COPT model.

#### COPT_LoadTuneParam

**Synopsis**

```
int COPT_LoadTuneParam(copt_prob *prob, int idx)
```

**Description**

Load the parameter tuning results of the specified number into the model.

**Arguments**

`prob`

COPT model.

`idx`

The number of the parameter tuning result.

### COPT_ReadTune

**Synopsis**

```
int COPT_ReadTune(copt_prob *prob, const char *tunefilename)
```

**Description**

Read the parameter combination to be tuned from the tuning file to the model.

**Arguments**

`prob`

COPT model.

`tunefilename`

Tuning file names.

### COPT_WriteTuneParam

**Synopsis**

```
int COPT_WriteTuneParam(copt_prob *prob, int idx, const char
*parfilename)
```

**Description**

Output the parameter tuning result of the specified number to the parameter file.

**Arguments**

`prob`

COPT model.

`idx`

The number of the parameter tuning result.

`parfilename`

parameter file name.

### COPT_ReadOrd

**Synopsis**

```
int COPT_ReadOrd(copt_prob *prob, const char *ordfilename)
```

**Description**

Read a branching order file (ORD-format) into the current model to reuse a predefined branching strategy.

**Arguments**

`prob`

The COPT problem.

`ordfilename`

Name of the branching order file.

#### COPT_WriteOrd

**Synopsis**

```
int COPT_WriteOrd(copt_prob *prob, const char *ordfilename)
```

**Description**

Write the branching order information of the current model into an ORD-format file, so that the branching strategy can be saved and reused.

**Arguments**

`prob`

The COPT problem.

`ordfilename`

Path of the ORD-format file.

### 23.6.14 Callback utilities

Certain callback utilities methods can only be called in certain contexts, which are listed below:

Table 23.1: Callback utilities

| Context | Methods |
|---|---|
| COPT_CBCONTEXT_INCUMBEN | *COPT_AddCallbackSolution*, *COPT_GetCallbackInfo* |
| COPT_CBCONTEXT_MIPSOL | *COPT_AddCallbackLazyConstr*, *COPT_AddCallbackLazyConstrs*, *COPT_AddCallbackSolution*, *COPT_GetCallbackInfo* |
| COPT_CBCONTEXT_MIPRELAX | *COPT_AddCallbackUserCut*, *COPT_AddCallbackUserCuts*, *COPT_AddCallbackSolution*, *COPT_GetCallbackInfo* |
| COPT_CBCONTEXT_MIPNODE | *COPT_AddCallbackSolution*, *COPT_GetCallbackInfo* |

#### COPT_AddLazyConstr

**Synopsis**

```
int COPT_AddLazyConstr(
    copt_prob *prob,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char *rowName)
```

**Description**

Add a lazy constraint to the MIP model.

**Arguments**

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero elements in the lazy constraint.

`rowMatIdx`

Column index of non-zero elements in the lazy constraint.

`rowMatElem`

Values of non-zero elements in the lazy constraint.

`cRowSense`

The sense of the new lazy constraint.

Please refer to *Constraint senses* for all the supported types.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` will be treated as lower and upper bounds for the constraint. This is the recommended method for defining constraints.

If `cRowSense` is provided, then `dRowBound` and `dRowUpper` will be treated as RHS and **range** for the constraint. In this case, `dRowUpper` is only required when `cRowSense = COPT_RANGE`, where

lower bound is `dRowBound - dRowUpper`

upper bound is `dRowBound`

`dRowBound`

Lower bound or RHS of the lazy constraint.

`dRowUpper`

Upper bound or **range** of the lazy constraint.

`rowName`

The name of the lazy constraint. Can be `NULL`.

### COPT_AddLazyConstrs

#### Synopsis

```
int COPT_AddLazyConstrs(
    copt_prob *prob,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowBound,
    const double *rowUpper,
    char const *const *rowNames)
```

#### Description

Add a set of lazy constraints to the MIP model.

#### Arguments

`prob`

The COPT problem.

`nAddRow`

Number of new lazy constraints.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new lazy constraints.

Please refer to *Constraint senses* for all the supported types.

If `rowSense` is `NULL`, then `rowBound` and `rowUpper` will be treated as lower and upper bounds for constraints. This is the recommended method for defining constraints.

If `rowSense` is provided, then `rowBound` and `rowUpper` will be treated as RHS and **range** for constraints. In this case, `rowUpper` is only required when there are `COPT_RANGE` constraints, where the

lower bound is `rowBound[i] - fabs(rowUpper[i])`

upper bound is `rowBound[i]`

`rowBound`

Lower bounds or RHS of new lazy constraints.

`rowUpper`

Upper bounds or **range** of new lazy constraints.

`rowNames`

Names of new lazy constraints. Can be `NULL`.

### COPT_AddUserCut

#### Synopsis

```
int COPT_AddUserCut(
    copt_prob* prob,
    int nRowMatCnt,
    const int* rowMatIdx,
    const double* rowMmatElem,
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char* rowName)
```

#### Description

Add a user cut to the MIP model.

#### Arguments

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero elements in the user cut.

`rowMatIdx`

Column index of non-zero elements in the user cut.

`rowMatElem`

Values of non-zero elements in the user cut.

`cRowSense`

The sense of the user cut.

Please refer to *Constraint senses* for all the supported types.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` will be treated as lower and upper bounds for the constraint. This is the recommended method for defining constraints.

If `cRowSense` is provided, then `dRowBound` and `dRowUpper` will be treated as RHS and **range** for the constraint. In this case, `dRowUpper` is only required when `cRowSense = COPT_RANGE`, where

lower bound is `dRowBound - dRowUpper`

upper bound is `dRowBound`

`dRowBound`

Lower bound or RHS of the user cut.

`dRowUpper`

Upper bound or **range** of the user cut.

`rowName`

The name of the user cut. Can be `NULL`.

### COPT_AddUserCuts

#### Synopsis

```
int COPT_CALL COPT_AddUserCuts(
    copt_prob *prob,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowBound,
    const double *rowUpper,
    char const *const *rowNames)
```

#### Description

Add a set of user cuts to the MIP model.

**Arguments**

`prob`

The COPT problem.

`nAddRow`

Number of new user cuts.

`rowMatBeg, rowMatCnt, rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new user cuts.

Please refer to *Constraint senses* for all the supported types.

If `rowSense` is `NULL`, then `rowBound` and `rowUpper` will be treated as lower and upper bounds for constraints. This is the recommended method for defining constraints.

If `rowSense` is provided, then `rowBound` and `rowUpper` will be treated as RHS and **range** for constraints. In this case, `rowUpper` is only required when there are `COPT_RANGE` constraints, where the

lower bound is `rowBound[i] - fabs(rowUpper[i])`

upper bound is `rowBound[i]`

`rowBound`

Lower bounds or RHS of new user cuts.

`rowUpper`

Upper bounds or **range** of new user cuts.

`rowNames`

Names of new user cuts. Can be `NULL`.

## COPT_SetCallback

**Synopsis**

```
int COPT_SetCallback(
    copt_prob *prob,
    int (COPT_CALL *cb)(copt_prob *prob, void *cbdata, int cbctx,
    void *userdata),
    int cbctx,
    void *userdata)
```

**Description**

Set the callback function of the model.

**Arguments**

`prob`

The COPT problem.

`cb`

Callback function.

`cbctx`

Callback context. Please refer to *Callback context* .

`userdata`

User defined data. The data will be passed to the solver without modification.

### COPT_GetCallbackInfo

**Synopsis**

```
int COPT_GetCallbackInfo(void *cbdata, const char* cbinfo, void
*p_val)
```

**Description**

Retrieve the value of the specified callback information.

**Arguments**

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_GetCallbackInfo().

`cbinfo`

The name of the callback information. Please refer to *Callback information* for possible values.

`p_val`

Pointer to the value of the callback information.

### COPT_AddCallbackSolution

**Synopsis**

```
int COPT_AddCallbackSolution(void *cbdata, const double *sol,
double* p_objval)
```

**Description**

Set feasible solutions for the specified variables.

**Arguments**

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackSolution().

`sol`

The solution vector.

`p_objval`

Pointer to the objective value for solution.

### COPT_AddCallbackUserCut

#### Synopsis

```
int COPT_AddCallbackUserCut(
    void *cbdata,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowRhs)
```

#### Description

Add a user cut to the MIP model from within the callback function.

#### Arguments

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackUserCut().

`nRowMatCnt`

Number of non-zero elements in the user cut.

`rowMatIdx`

Column index of non-zero elements in the user cut.

`rowMatElem`

Values of non-zero elements in the user cut.

`cRowSense`

The sense of the new user cut. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

The user cut added from within callback can only have a single bound.

`dRowRhs`

RHS of the user cut.

### COPT_AddCallbackUserCuts

#### Synopsis

```
int COPT_AddCallbackUserCuts(
    void *cbdata,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
```

```
    const double *rowRhs)
```

**Description**

Add a set of user cuts to the MIP model from within the callback function.

**Arguments**

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackUserCuts().

`nAddRow`

Number of new user cuts.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new user cuts. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

The user cuts added from within callback can only have single bounds.

`rowRhs`

RHS of new user cuts.

### COPT_AddCallbackLazyConstr

**Synopsis**

```
int COPT_AddCallbackLazyConstr(
    void *cbdata,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowRhs)
```

**Description**

Add a lazy constraint to the MIP model from within the callback function.

**Arguments**

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackLazyConstr().

`nRowMatCnt`

Number of non-zero elements in the lazy constraint.

When `nRowMatCnt<=0`, the MIP candidate solution will be directly rejected without adding a lazy constraint. And `rowMatIdx`, `rowMatElem`, `cRowSense` and `dRowRhs` will be ignored.

`rowMatIdx`

Column index of non-zero elements in the lazy constraint.

`rowMatElem`

Values of non-zero elements in the lazy constraint.

`cRowSense`

The sense of new lazy constraint. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

The lazy constraint added from within callback can only have a single bound.

`dRowRhs`

RHS of the lazy constraint.

### COPT_AddCallbackLazyConstrs

**Synopsis**

```
int COPT_AddCallbackLazyConstrs(
    void *cbdata,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowRhs)
```

**Description**

Add a set of lazy constraints to the MIP model from within the callback function.

**Arguments**

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackLazyConstrs().

`nAddRow`

Number of new lazy constraints.

When `nAddRow<=0`, the MIP candidate solution will be directly rejected without adding lazy constraints.

And other parameters(apart from `cbdata`) will be ignored.

`rowMatBeg, rowMatCnt, rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new lazy constraints. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

The lazy constraints added from within callback can only have single bounds.

`rowRhs`

RHS of new lazy constraints.

Note: If the user gives a solution to COPT in a callback, the user should ensure that this fulfills the lazy constraint callback since that won't be called for user-provided solutions.

### 23.6.15 Other API functions

#### COPT_GetBanner

**Synopsis**

```
int COPT_GetBanner(char *buff, int buffSize)
```

**Description**

Obtains a C-style string as banner, which describes the COPT version information.

**Arguments**

`buff`

A buffer for holding the resulting string.

`buffSize`

The size of the above buffer.

#### COPT_GetRetcodeMsg

**Synopsis**

```
int COPT_GetRetcodeMsg(int code, char *buff, int buffSize)
```

**Description**

Obtains a C-style string which explains a return code value in plain text.

**Arguments**

`code`

The return code from a COPT API function.

`buff`

A buffer for holding the resulting string.

`buffSize`

The size of the above buffer.

#### COPT_Interrupt

**Synopsis**

```
int COPT_Interrupt(copt_prob *prob)
```

**Description**

Interrupt solving process of current problem.

**Arguments**

`prob`

The COPT problem.

### 23.6.16 Multi-objective Model API Functions

#### COPT_MultiObjSetColObj

**Synopsis**

```
int COPT_MultiObjSetColObj(
    copt_prob *prob,
    int iObj,
    int num,
    const int *list,
    const double *colObj)
```

**Description**

Set the specified linear objective in a multi-objective model.

Note: this operation overwrites the existing objective definition.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`num`

Number of linear objective coefficients.

`list`

Index list of linear objective coefficients.

`colObj`

Value list of linear objective coefficients.

### COPT_MultiObjGetColObj

**Synopsis**

```
int COPT_MultiObjGetColObj(
    copt_prob *prob,
    int iObj,
    int num,
    const int *list,
    double *colObj)
```

**Description**

Retrieve the specified linear objective in a multi-objective model.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`num`

Number of nonzero coefficients in the linear objective.

`list`

Indices of the nonzero coefficients in the linear objective.

`colObj`

Values of the nonzero coefficients in the linear objective.

### COPT_MultiObjDelColObj

**Synopsis**

```
int COPT_MultiObjDelColObj(
    copt_prob *prob,
    int iObj)
```

**Description**

Delete the specified linear objective in a multi-objective model.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

### COPT_MultiObjSetObjSense

**Synopsis**

```
int COPT_MultiObjSetObjSense(
    copt_prob *prob,
    int iObj,
    int iObjSense)
```

**Description**

Set the objective sense for the specified objective in a multi-objective model.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`iObjSense`

The optimization sense, either `COPT_MAXIMIZE` or `COPT_MINIMIZE`.

### COPT_MultiObjSetObjConst

**Synopsis**

```
int COPT_MultiObjSetObjConst(
    copt_prob *prob,
    int iObj,
    double dObjConst)
```

**Description**

Set the objective constant for the specified objective in a multi-objective model.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`dObjConst`

Constant term of the objective function.

### COPT_MultiObjSetObjParam

#### Synopsis

```
int COPT_MultiObjSetObjParam(
    copt_prob *prob,
    int iObj,
    const char *paramName,
    double dblParam)
```

#### Description

Set the objective parameter value for the specified objective in a multi-objective model.

#### Arguments

`prob`

COPT problem.

`iObj`

Index of the objective.

`paramName`

Name of the objective parameter.

Possible values: `COPT_MULTIOBJ_PRIORITY` , `COPT_MULTIOBJ_WEIGHT` , `COPT_MULTIOBJ_ABSTOL` , `COPT_MULTIOBJ_RELTOL` .

`dblParam`

Value to set for the objective parameter.

### COPT_MultiObjGetObjParam

#### Synopsis

```
int COPT_MultiObjGetObjParam(
    copt_prob *prob,
    int iObj,
    const char *paramName,
    double *p_dblParam)
```

#### Description

Retrieve the objective parameter value for the specified objective in a multi-objective model.

#### Arguments

`prob`

COPT problem.

`iObj`

Index of the objective.

`paramName`

Name of the objective parameter.

Possible values are: `COPT_MULTIOBJ_PRIORITY` , `COPT_MULTIOBJ_WEIGHT` , `COPT_MULTIOBJ_ABSTOL` , `COPT_MULTIOBJ_RELTOL` .

`p_dblParam`

Pointer to store the retrieved parameter value.

### COPT_MultiObjResetObjParam

**Synopsis**

```
int COPT_MultiObjResetObjParam(
    copt_prob *prob,
    int iObj)
```

**Description**

Reset all objective parameters of the specified objective in a multi-objective model to their default values.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

### COPT_MultiObjSetIntParam

**Synopsis**

```
int COPT_MultiObjSetIntParam(
    copt_prob *prob,
    int iObj,
    const char *paramName,
    int intParam)
```

**Description**

Set an integer solver parameter for the model corresponding to the specified objective in a multi-objective problem.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`paramName`

Name of the parameter.

`intParam`

Integer value of the parameter to be set.

### COPT_MultiObjGetIntParam

**Synopsis**

```
int COPT_MultiObjGetIntParam(
    copt_prob *prob,
    int iObj,
    const char *paramName,
    int *p_intParam)
```

**Description**

Get the integer solver parameter for the model corresponding to the specified objective in a multi-objective problem.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`paramName`

Name of the parameter.

`p_intParam`

Pointer to the retrieved integer parameter value.

### COPT_MultiObjSetDblParam

**Synopsis**

```
int COPT_MultiObjSetDblParam(
    copt_prob *prob,
    int iObj,
    const char *paramName,
    int dblParam)
```

**Description**

Set the double solver parameter for the model corresponding to the specified objective in a multi-objective problem.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`paramName`

Name of the parameter.

`dblParam`

Value of the double parameter to set.

### COPT_MultiObjGetDblParam

**Synopsis**

```
int COPT_MultiObjGetDblParam(
    copt_prob *prob,
    int iObj,
    const char *paramName,
    int *p_dblParam)
```

**Description**

Get the double solver parameter of the model corresponding to the specified objective in a multi-objective problem.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`paramName`

Name of the parameter.

`p_dblParam`

Pointer to store the value of the double parameter.

### COPT_MultiObjResetParam

**Synopsis**

```
int COPT_MultiObjResetParam(
    copt_prob *prob,
    int iObj)
```

**Description**

Reset all solver parameters of the model corresponding to the specified objective in a multi-objective problem to their default settings.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

### COPT_MultiObjGetIntAttr

**Synopsis**

```
int COPT_MultiObjGetIntAttr(
    copt_prob *prob,
    int iObj,
    const char *attrName,
    int *p_intAttr)
```

**Description**

Retrieve an integer attribute of the specified objective in a multi-objective problem.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`attrName`

Name of the attribute.

`p_intAttr`

Pointer to the integer value of the attribute.

### COPT_MultiObjGetDblAttr

**Synopsis**

```
int COPT_MultiObjGetDblAttr(
    copt_prob *prob,
    int iObj,
    const char *attrName,
    double *p_dblAttr)
```

**Description**

Retrieve a double attribute of the specified objective in a multi-objective problem.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`attrName`

Name of the attribute.

Possible values are: `COPT_DBLATTR_LPOBJVAL`, `COPT_DBLATTR_BESTOBJ`, `COPT_DBLATTR_OBJCONST`.

`p_dblAttr`

Pointer to the double value of the attribute.

#### COPT_MultiObjGetPoolObjVal

**Synopsis**

```
int COPT_MultiObjGetPoolObjVal(
    copt_prob *prob,
    int iObj,
    int iSol,
    double *p_objVal)
```

**Description**

Retrieve the value of the specified objective at the solution with index `iSol` in the solution pool.

**Arguments**

`prob`

COPT problem.

`iObj`

Index of the objective.

`iSol`

Index of the solution in the solution pool.

`p_objVal`

Pointer to the value of the objective.

### 23.6.17 CPU and Memory Binding Functions

#### COPT_BindNumaCpu

**Synopsis**

```
int COPT_BindNumaCpu(
    copt_env *env,
    int numaNode)
```

**Description**

Bind the CPUs of the current process to a NUMA node.

**Arguments**

`env`

The COPT environment.

`numaNode`

ID of the NUMA node.

### COPT_BindNumaMem

**Synopsis**

```
int COPT_BindNumaMem(
    copt_env *env,
    int numaNode)
```

**Description**

Bind memory for the current process to a NUMA node (Linux only).

**Arguments**

`env`

The COPT environment.

`numaNode`

ID of the NUMA node.

### COPT_GetCpuAffinity

**Synopsis**

```
int COPT_GetCpuAffinity(
    copt_env *env,
    int *cpuList,
    int len,
    int *pReqSize)
```

**Description**

Get CPU affinity for the current process, which is saved in an integer array.

**Arguments**

`env`

The COPT environment.

`cpuList`

Integer array to store the list of CPU IDs.

`len`

Length of the `cpuList` array.

`pReqSize`

Pointer to store the actual number of bound CPUs.

### COPT_GetNumaNodeCount

**Synopsis**

```
int COPT_GetNumaNodeCount(
    copt_env *env,
    int *pCnt)
```

**Description**

Get count of NUMA nodes.

**Arguments**

`env`

The COPT environment.

`pCnt`

Pointer to store the count of NUMA nodes.

### COPT_SetCpuAffinity

**Synopsis**

```
int COPT_SetCpuAffinity(
    copt_env *env,
    const char *hexMask)
```

**Description**

Set CPU affinity with given mask string.

**Arguments**

`env`

The COPT environment.

`hexMask`

CPU mask string of hexadecimal characters.

# Chapter 24

# Python API Reference

The **Cardinal Optimizer** provides a Python API library. This chapter documents all COPT Python constants and API functions for python applications.

## 24.1 Constants

Python Constants are necessary to solve a problem using the Python interface. There are four types of constants defined in COPT Python API library. They are general constants, information constants, parameters and attributes.

### 24.1.1 General Constants

For the contents of Python general constants, see *General Constants*.

General constants are those commonly used in modeling, such as optimization directions, variable types, and solving status, etc. Users may refer to general constants with a `'COPT'` prefix. For instance, `COPT.VERSION_MAJOR` is the major version number of the **Cardinal Optimizer**.

### 24.1.2 Attributes

For the contents of Python attribute constants, see *Attributes*.

In the Python API, users may refer to attributes using a `'COPT.Attr'` prefix. For instance, `COPT.Attr.Cols` is the number of variables or columns in the model.

In the Python API, user can get the attribute value by specifying the attribute name. Attributes are mostly used in `Model.getAttr()` method to query properties of the model, please refer to *Python API: Model Class* for details. Here is an example:

- `Model.getAttr()`: `Model.getAttr("Cols")` obtain the number of variables or columns in the model.

### 24.1.3 Information

For the contents of Python API information class constants, see *Information*.

In the Python API, user can access the information through the `COPT.Info` prefix. For instance, `COPT.Info.Obj` is the objective function coefficients for variables (columns).

In the Python API, user can get or set the information value of the object by specifying the information name:

- Get the value of variable or constraint information: `Model.getInfo()` / `Var.getInfo()` / `Constraint.getInfo()`
- Set the value of variable or constraint information: `Model.setInfo()` / `Var.setInfo()` / `Constraint.setInfo()`

### 24.1.4 Callback Information

For the content of Python API callback information class constants, see *Callback Information*.

In the Python API, callback-related information constants are defined in the `CbInfo` class. User can access the callback information via `COPT.CbInfo.` prefix.

For instance, `COPT.CbInfo.BestObj` is the current best objective.

In the Python API, user can get the value of callback information by specifying the information name.

For instance, `CallbackBase.getInfo(COPT.CbInfo.BestObj)` : get the value of the current best objective.

### 24.1.5 Parameters

For the contents of Python API Parameters class constants, see *Parameters*.

Parameters control the operation of the **Cardinal Optimizer**. They can be modified before the optimization begins.

In the Python API, user can access parameters through the `COPT.Param` prefix. For instance, `COPT.Param.TimeLimit` is time limit in seconds of the optimization.

In the Python API, user can get and set the parameter value by specifying the parameter name. The provided functions are as follows, please refer to *Python API: Model Class* for details.

- Get detailed information of the specified parameter (current value/max/min): `Model.getParamInfo()`
- Get the current value of the specified parameter: `Model.getParam()`
- Set the specified parameter value: `Model.setParam()`

## 24.2 Python Modeling Classes

Python modeling classes are essential for the Python interface of Cardinal Optimizer. It provides plentiful easy-to-use methods to quickly build optimization models in complex practical scenarios. This section will explain these functions and their usage.

### 24.2.1 EnvrConfig Class

EnvrConfig object contains operations related to client configuration, and provides the following methods:

#### EnvrConfig()

**Synopsis**

`EnvrConfig()`

**Description**

Constructor of EnvrConfig class. This method creates and returns an *EnvrConfig Class* object.

**Example**

```
# Create client configuration
envconfig = EnvrConfig()
```

#### EnvrConfig.set()

**Synopsis**

`set(name, value)`

**Description**

Set client configuration.

**Arguments**

`name`

Name of configuration parameter. Please refer to *Client configuration* for possible values.

`value`

Value of configuration parameter.

**Example**

```
# Set client configuration
envconfig.set(COPT.CLIENT_WAITTIME, 600)
envconfig.set(COPT.CLIENT_CLUSTER, "127.0.0.1")
# Turn off the banner output when creating COPT environment (such as version, etc.)
envconfig.set("nobanner", "1")
```

### 24.2.2 Envr Class

Envr object contains operations related to COPT optimization environment, and provides the following methods:

### Envr()

**Synopsis**

`Envr(arg=None)`

**Description**

Function for constructing Envr object. This method creates and returns an *Envr Class* object.

**Arguments**

`arg`

Path of license file or client configuration. Optional argument, defaults to `None`.

**Example**

```
# Create solving environment
env = Envr()
```

### Envr.createModel()

**Synopsis**

`createModel(name="")`

**Description**

Create optimization model and return a *Model Class* object.

**Arguments**

`name`

The name of the Model object to be created. Optional, `""` by default.

**Example**

```
# Create optimization model
model = env.createModel("coptprob")
```

### Envr.close()

**Synopsis**

`close()`

**Description**

Close connection to remote server. (For floating or cluster license)

This also releases the license token held by the environment, sooner than `dispose()` does.

The method does not release the environment itself. The memory held by the environment, including the models created from it, is freed only by `dispose()`.

**Example**

```
# Close connection to remote server
env.close()
```

### Envr.dispose()

**Synopsis**

```
dispose()
```

**Description**

Release the environment on the calling thread. May be called more than once, and the environment must not be used afterwards. `Envr` also supports the context manager protocol, so a `with` block releases the environment on exit.

This frees all memory held by the environment, including the models created from it, and releases the license token as well. For a floating or cluster license, `close()` returns the token more quickly.

See *Best practice: Release environments and models* for details.

**Example**

```
# Release the environment
env.dispose()
```

### Envr.bindNumaCpu()

**Synopsis**

```
bindNumaCpu(numaNode)
```

**Description**

Bind the CPUs for the current process to a NUMA node.

**Arguments**

`numaNode`

ID of a NUMA node.

**Example**

```
env.bindNumaCpu(0)
```

### Envr.bindNumaMem()

**Synopsis**

```
bindNumaMem(numaNode)
```

**Description**

Bind memory for the current process to a NUMA node (Linux only).

**Arguments**

`numaNode`

ID of a NUMA node.

**Example**

```
env.bindNumaMem(0)
```

#### Envr.getCpuAffinity()

**Synopsis**

`getCpuAffinity()`

**Description**

Get CPU affinity for the current process, which is saved in an integer array.

**Example**

```
cpu_list = env.getCpuAffinity()
```

#### Envr.getNumaNodeCount()

**Synopsis**

`getNumaNodeCount()`

**Description**

Get count of NUMA nodes.

**Example**

```
count = env.getNumaNodeCount()
```

#### Envr.setCpuAffinity()

**Synopsis**

`setCpuAffinity(hexMask)`

**Description**

Set CPU affinity with given mask string.

**Arguments**

`hexMask`

CPU mask string of hexadecimal characters.

**Example**

```
env.setCpuAffinity("0f")
```

### 24.2.3 Model Class

For easy access to model's attributes and optimization parameters, Model object provides methods such as `Model.Rows`. The full list of attributes can be found in *Attributes* section. For convenience, attributes can be accessed by their names in capital or lower case.

Note that for LP or MIP, both the objective value and the solution status can be accessed through `Model.objval` and `Model.status`.

For optimization parameters, they can be set in the form `"Model.Param.TimeLimit = 10"`. For details of the parameter names supported, please refer to *Parameters* section.

Class Model contains COPT model-related operations and provides the following methods:

### Model.addVar()

#### Synopsis

```
addVar(lb=0.0, ub=COPT.INFINITY, obj=0.0, vtype=COPT.CONTINUOUS,
name="", column=None)
```

#### Description

Add a decision variable to model and return the added *Var Class* object.

#### Arguments

`lb`

Lower bound for new variable. Optional, 0.0 by default.

`ub`

Upper bound for new variable. Optional, `COPT.INFINITY` by default.

`obj`

Objective parameter for new variable. Optional, 0.0 by default.

`vtype`

Variable type. Optional, `COPT.CONTINUOUS` by default. Please refer to *Variable types* for possible types.

`name`

Name for new variable. Optional, `""` by default, which is automatically generated by solver.

`column`

Column corresponds to the variable. Optional, `None` by default.

#### Example

```
# Add a continuous variable
x = m.addVar()
# Add a binary variable
y = m.addVar(vtype=COPT.BINARY)
# Add an integer variable with lowerbound -1.0, upperbound 1.0, objective coefficient␣
↪1.0 and variable name "z"
z = m.addVar(-1.0, 1.0, 1.0, COPT.INTEGER, "z")
```

### Model.addVars()

#### Synopsis

```
addVars(*indices, lb=0.0, ub=COPT.INFINITY, obj=0.0, vtype=COPT.
CONTINUOUS, nameprefix="C")
```

#### Description

Add multiple new variables to a model. Return a *tupledict Class*, whose key is index of the variable and value is the *Var Class* object.

#### Arguments

`*indices`

Indices for accessing the new variables.

`lb`

Lower bounds for new variables. Optional, 0.0 by default.

`ub`

Upper bounds for new variables. Optional, `COPT.INFINITY` by default.

`obj`

Objective costs for new variables. Optional, 0.0 by default.

`vtype`

Variable types. Optional, `COPT.CONTINUOUS` by default. Please refer to *Variable types* for possible types.

`nameprefix`

Name prefix for new variables. Optional, `"C"` by default. The actual name and the index of the variables are automatically generated by COPT.

**Example**

```
# Add three-dimensional integer variable, 6 variables in total
x = m.addVars(1, 2, 3, vtype=COPT.INTEGER)
# Add two continuous variable y, whose index is designated by elements in tuplelist␣
↪and prefix is "tl"
tl = tuplelist([(0, 1), (1, 2)])
y  = m.addVars(tl, nameprefix="tl")
```

### Model.addMVar()

**Synopsis**

`addMVar(shape, lb=0.0, ub=COPT.INFINITY, obj=0.0, vtype=COPT.CONTINUOUS, nameprefix="")`

**Description**

Add *MVar Class* object to the model. It is used in matrix modeling and can be operated like a multidimensional array in NumPy, its shape and dimensions are similarly defined.

**Arguments**

`shape`

The value is an integer, or a tuple of integers. which represents the shape of a *MVar Class* object.

`lb`

The lower bound of the variable. Optional parameter, defaults to 0.0.

`ub`

The upper bound of the variable. Optional parameter, defaults to `COPT.INFINITY`.

`obj`

The objective function coefficients for the variables. Optional parameter, defaults to 0.0.

`vtype`

The type of the variable. Optional parameter, the default is `COPT.CONTINUOUS`, see the possible values in *Variable types*.

`nameprefix`

Variable name prefix. Optional parameter, the default is "", its actual name is automatically generated by combining the subscript of the variable.

**Return value**

Returns a *MVar Class* object

**Example**

```
model.addMVar((2, 3), lb=0.0, nameprefix="mx")
```

### Model.addConstr()

**Synopsis**

```
addConstr(lhs, sense=None, rhs=None, name="")
```

**Description**

Add a linear constraint to the model, return *Constraint Class* object or *MConstr Class* object;

Add a semidefinite constraint to the model, return *PsdConstraint Class* object or *MPsdConstr Class* object;

Add an indicator constraint to the model and return the *GenConstr Class* object;

Adds a LMI constraint to the model, returning a *LmiConstraint Class* object.

If a linear constraint added, then the parameter `lhs` can take the value of *Var Class* object, *LinExpr Class* object, *ConstrBuilder Class* object or *MConstrBuilder Class* .

If a positive semi-definite constraint added, then the parameter `lhs` can take the value of *PsdExpr Class* object, *PsdConstrBuilder Class* object or *MPsdConstrBuilder Class* .

If an indicator constraint added, then the parameter `lhs` is *GenConstrBuilder Class* object, ignoring other parameters.

If a LMI constraint added, then the parameter `lhs` can take the value of *LmiExpr Class* object.

**Arguments**

`lhs`

Left-hand side expression for new linear constraint or constraint builder.

`sense`

Sense for the new constraint. Optional, None by default. Please refer to *Constraint type* for possible values.

`rhs`

Right-hand side expression for the new constraint. Optional, None by default. It can be a constant, or *Var Class* object, or *LinExpr Class* object.

`name`

Name for new constraint. Optional, "" by default, generated by solver automatically.

**Example**

```
# Add a linear constraint: x + y == 2
m.addConstr(x + y, COPT.EQUAL, 2)
# Add a linear constraint: x + 2*y >= 3
m.addConstr(x + 2*y >= 3.0)
# Add an indicator constraint
m.addConstr((x == 1) >> (2*y + 3*z <= 4))
```

### Model.addBoundConstr()

#### Synopsis

```
addBoundConstr(expr, lb=-COPT.INFINITY, ub=COPT.INFINITY, name="")
```

#### Description

Add a constraint with a lower bound and an upper bound to a model and return the added *Constraint Class* object.

#### Arguments

`expr`

Expression for the new constraint, which can be *Var Class* object or *LinExpr Class* object.

`lb`

Lower bound for the new constraint. Optional, `-COPT.INFINITY` by default.

`ub`

Upper bound for the new constraint. Optional, `COPT.INFINITY` by default.

`name`

Name for new constraint. Optional, `""` by default, automatically generated by solver.

#### Example

```
# Add linear bilateral constraint: -1 <= x + y <= 1
m.addBoundConstr(x + y, -1.0, 1.0)
```

### Model.addConstrs()

#### Synopsis

```
addConstrs(generator, nameprefix="R")
```

#### Description

Add a set of linear constraints, semidefinite constraints, or indicator constraints to the model.

If parameter `generator` is integer, the return a *ConstrArray Class* object consisting of `generator` number of empty *Constraint Class* objects, and users need to specify these constraints.

If parameter `generator` is expression generator, then return a *tupledict Class* object whose key is the index of linear constraint and value is the corresponding *Constraint Class* object. Every iteration generates a *Constraint Class* object.

If the parameter `generator` is a matrix expression generator, return a *MConstr Class* object or a *MPsdConstr Class* object.

If the parameter `generator` is an indicator expression generator, return a *GenConstrArray Class* object.

**Arguments**

`generator`

A generator expression, where each iteration produces a *Constraint Class* object, or a matrix expression builder, or an indicator expression generator.

`nameprefix`

Name prefix for new constraints. Optional, `"R"` by default. The actual name and the index of the constraints are automatically generated by COPT.

**Example**

```
# Add 10 linear constraints, each constraint shaped like: x[0] + y[0] >= 2.0
m.addConstrs(x[i] + y[i] >= 2.0 for i in range(10))
```

### Model.addMConstr()

**Synopsis**

```
addMConstr(A, x, sense, b, nameprefix="")
```

**Description**

By means of matrix modeling, a set of linear constraints are added to the model. If the value of `sense` here is `COPT.LESS_EQUAL` , the added constraint is $Ax <= b$.

It is more convenient to generate *MLinExpr Class* objects by matrix multiplication, and then use overloaded comparison operators to generate *MConstrBuilder Class* object, which can be used as input of `Model.addConstrs()` to generate a set of linear constraints.

**Arguments**

`A`

Parameter A is a two-dimensional NumPy matrix, SciPy compressed sparse column matrix ( `csc_matrix` ) or compressed sparse row matrix ( `csr_matrix` ).

`x`

The variable corresponding to the linear term can be a *MVar Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object. If it is `None`, all variables in the model are taken, and the number of columns of `A` must equal the number of variables in the model.

`sense`

The type of constraint. Possible values refer to *Constraint type*.

`b`

The value on the right side of the constraint, usually a floating point number, can also be a set of numbers, or a one-dimensional array of NumPy.

`nameprefix`

Constraint name prefix.

**Return value**

Returns a *MConstr Class* object

**Example**

```
A = np.full((2, 3), 1)
mx = model.addMVar(3, nameprefix="mx")
mc = model.addMConstr(A, mx, 'L', 1.0, nameprefix="mc")
```

### Model.addSOS()

**Synopsis**

```
addSOS(sostype, vars, weights=None)
```

**Description**

Add a SOS constraint to model and return the added *SOS Class* object.

If param `sostype` is *SOSBuilder Class* object, then the values of param `vars` and param``weights`` will be ignored;

If param `sostype` is SOS constraint type, it can be valued as *SOS-constraint types*, then param `vars` represents variables of SOS constraint, taking the value of *VarArray Class* object, list, dict or *tupledict Class* object;

If param `weights` is `None`, then variable weights of SOS constraints will be automatically generated by solver. Otherwise take the input from user as weights, possible values can be list, dictionary or *tupledict Class* object.

**Arguments**

`sostype`

SOS constraint type or SOS constraint builder.

`vars`

Variables of SOS constraints.

`weights`

Weights of variables in SOS constraints, optional, `None` by default.

**Example**

```
# Add an SOS1 constraint, including variable x and y, weights of 1 and 2.
m.addSOS(COPT.SOS_TYPE1, [x, y], [1, 2])
```

### Model.addGenConstrIndicator()

**Synopsis**

```
addGenConstrIndicator(binvar, binval, lhs, sense=None, rhs=None,
type=COPT.INDICATOR_IF, name="")
```

**Description**

Add an indicator constraint with the specified type to a model and return the added *GenConstr Class* object.

If the parameter `lhs` is *ConstrBuilder Class* object, then the values of parameter `sense` and parameter `rhs` will be ignored.

If parameter `lhs` represents Left-hand side expression, it can take value of *Var Class* object or *LinExpr Class* object.

**Arguments**

`binvar`

Indicator variable.

`binval`

Value of indicator variable, can be `True` or `False`.

`lhs`

Left-hand side expression for the linear constraint triggered by the indicator or linear constraint builder.

`sense`

Sense for the linear constraint. Optional, `None` by default. Please refer to *Constraint type* for possible values.

`rhs`

Right-hand-side value for the linear constraint triggered by the indicator. Optional, `None` by default, value type is constant.

`type`

Type of the indicator constraint. Optional, `COPT.INDICATOR_IF` by default. Please refer to *Indicator Constraint type* for possible values.

`name`

Name for new indicator constraint. Optional, `""` by default, generated by solver automatically.

**Example**

```
# Add an indicator constraint, if x is True, then the linear constraint y + 2*z >= 3␣
↪should hold
m.addGenConstrIndicator(x, True, y + 2*z >= 3)
# Add an indicator constraint, if the linear constraint y + 2*z >= 3 hold, then x␣
↪should be True
m.addGenConstrIndicator(x, True, y + 2*z >= 3, type=COPT.INDICATOR_ONLYIF)
```

### Model.addGenConstrIndicators()

**Synopsis**

```
addGenConstrIndicators(builders, nameprefix="")
```

**Description**

Add a set of indicator constraints with the specified type to a model and return the added *GenConstrArray Class* object.

**Arguments**

`builders`

A set of indicator constraint builders. The possible values could be *GenConstrBuilderArray Class* object, a list or dictionary of *GenConstrBuilder Class* object.

`nameprefix`

Nameprefix of indicator variable. Optional, `""` by default, generated by solver automatically.

### Model.addGenConstrMin()

**Synopsis**

```
addGenConstrMin(resvar, vars, constant=None, name="")
```

**Description**

Add a constraint of the form $y = \min\{x_1, x_2, \cdots, x_n, c\}$ to the model.

**Arguments**

`resvar`

The term `y` on the left side of the equation and can be an object of class `Var` or `MVar` .

`vars`

The variable of the min{} function on the right side of the equation,

Possible values are `list` class objects.

`constant`

The constant term in the min{} function on the right side of the equation,

Optional parameter, the possible value is a floating number,

The default value is `None`.

`name`

Constraint name, optional parameter, default value is `""` .

**Return value**

It returns a `GenConstrX` Class object.

### Model.addGenConstrMax()

**Synopsis**

```
addGenConstrMax(resvar, vars, constant=None, name="")
```

**Description**

Add a constraint of the form $y = \max\{x_1, x_2, \cdots, x_n, c\}$ to the model.

**Arguments**

`resvar`

The term `y` on the left side of the equation and can be an object of class `Var` or `MVar` .

`vars`

The variable of the max{} function on the right side of the equation.

Possible values are `list` class objects.

`constant`

The constant term in the max{} function on the right side of the equation.

Optional parameter, the possible value is a floating number.

The default value is `None` .

`name`

Constraint name, optional parameter, default value is `""` .

**Return value**

It returns a `GenConstrX` Class object.

### Model.addGenConstrAbs()

**Synopsis**

```
addGenConstrAbs(resvar, argvar, name="")
```

**Description**

Add a constraint of the form $cy + d = |ax + b|$ to the model.

**Arguments**

`resvar`

$cy + d$ , possible values are objects of class `Var` / `MVar` or class `LinExpr` / `MLinExpr` .

`argvar`

$ax + b$ , the possible value is object of class `Var` / `MVar` or class `LinExpr` / `MLinExpr` .

`name`

Constraint name, optional parameter, default value is `""` .

**Return value**

It returns a `GenConstrX` Class object.

### Model.addGenConstrAnd()

**Synopsis**

```
addGenConstrAnd(resvar, vars, name="")
```

**Description**

Add a logical `and` constraint of the form $y = x_1$ and $x_2 \cdots$ and $x_n$ to the model.

**Arguments**

`resvar`

The term `y` on the left side of the equation and can be an object of class `Var` or `MVar` .

`vars`

Elements connected by logical operator `and` $x_i$, for $i \in \{1, 2, \cdots, n\}$

Possible values are `List` class (where the elements are binary `Var` class or `MVar` class objects).

`name`

Constraint name, optional parameter, default value is `""` .

**Return value**

It returns a `GenConstrX` Class object.

### Model.addGenConstrOr()

#### Synopsis

```
addGenConstrOr(resvar, vars, name="")
```

#### Description

Add a logical `or` constraint of the form $y = x_1$ or $x_2 \cdots$ or $x_n$ to the model.

#### Arguments

`resvar`

The term `y` on the left side of the equation and can be an object of class `Var` or `MVar` .

`vars`

Elements connected by logical operator `or` $x_i$, for $i \in \{1, 2, \cdots, n\}$

Possible values are `List` class (where the elements are binary `Var` class or `MVar` objects).

`name`

Constraint name, optional parameter, default value is `""` .

#### Return value

It returns a `GenConstrX` Class object.

### Model.addGenConstrPWL()

#### Synopsis

```
addGenConstrPWL(xvar, yvar, xpts, ypts, name="")
```

#### Description

Add a constraint of the form $y = f(x)$, where a piecewise linear function is defined as:

$$f(v) = \begin{cases} \tilde{y}_1 + \frac{\tilde{y}_2 - \tilde{y}_1}{\tilde{x}_2 - \tilde{x}_1}(v - \tilde{x}_1), & \text{if } v \leq x_1 \\ \tilde{y}_i + \frac{\tilde{y}_{i+1} - \tilde{y}_i}{\tilde{x}_{i+1} - \tilde{x}_i}(v - \tilde{x}_i), & \text{if } \tilde{x}_i \leq v \leq \tilde{x}_{i+1} \\ \tilde{y}_n + \frac{\tilde{y}_n - \tilde{y}_{n-1}}{\tilde{x}_n - \tilde{x}_{n-1}}(v - \tilde{x}_n), & \text{if } v \geq \tilde{x}_n \end{cases}$$

#### Arguments

`xvar`

`x`, which can be an object of class `Var` or class `MVar` .

`yvar`

The term `y` on the left side of the equation,

Possible values are objects of class `Var` / `MVar` or class `LinExpr` / `MLinExpr` .

`xpts`

$\tilde{\boldsymbol{x}}$, the abscissa of the segmentation point.

It should be arranged in ascending order of values, possible values are `List` class.

`ypts`

$\tilde{y}$ , the vertical coordinate of the segmentation point,

Possible values are `List` class.

`name`

Constraint name, optional parameter, default value is `""` .

**Return value**

It returns a `GenConstrX` Class object.

### Model.addConeByDim()

**Synopsis**

```
addConeByDim(dim, ctype, vtype, nameprefix="ConeV")
```

**Description**

Add a Second-Order-Cone (SOC) constraint with given dimension, and return the added *Cone Class* object.

**Arguments**

`dim`

Dimension of SOC constraint.

`ctype`

Type of SOC constraint.

`vtype`

Variable types of SOC constraint.

`nameprefix`

Name prefix of variables in SOC constraint. Optional, default to `"ConeV"`.

**Example**

```
# Add a 5 dimension rotated SOC constraint
m.addConeByDim(5, COPT.CONE_RQUAD, None)
```

### Model.addExpConeByDim()

**Synopsis**

```
addExpConeByDim(ctype, vtype, nameprefix="ExpConeV")
```

**Description**

Add an exponential cone constraint and return the added *ExpCone Class* object.

**Arguments**

`ctype`

Type of exponential cone constraint.

`vtype`

Variable types of exponential cone constraint.

`nameprefix`

Name prefix of variables in exponential cone constraint. Optional, default to `"ExpConeV"` .

**Example**

```
# Add a primal exponential cone
m.addExpConeByDim(COPT.EXPCONE_PRIMAL, None)
```

### Model.addCone()

**Synopsis**

```
addCone(vars, ctype)
```

**Description**

Add a Second-Order-Cone (SOC) constraint with given variables.

If argument `vars` is a *ConeBuilder Class* object, then the value of argument `ctype` will be ignored; If argument `vars` are variables, the optional values are *VarArray Class* objects, Python list, Python dictionary or *tupledict Class* objects, argument `ctype` is the type of SOC constraint.

**Arguments**

`vars`

Variables of SOC constraint.

`ctype`

Type of SOC constraint. Please refer to *SOC constraint types* for possible values.

**Example**

```
# Add a SOC constraint with [z, x, y] as variables
m.addCone([z, x, y], COPT.CONE_QUAD)
```

### Model.addCones()

**Synopsis**

```
addCones(vars, ctype)
```

**Description**

Add a set of Second-Order-Cone (SOC) constraint with given variables.

If argument `vars` is a *ConeBuilder Class* object or *ConeBuilderArray Class* object, then the value of argument `ctype` will be ignored; If argument `vars` are *MVar Class* , the argument `ctype` is the type of SOC constraint, and it must be specified.

**Arguments**

`vars`

Variables of SOC constraint.

`ctype`

Type of SOC constraint. Please refer to *SOC constraint types* for possible values.

### Model.addExpCone()

**Synopsis**

```
addExpCone(vars, ctype)
```

**Description**

Add an exponential cone to the model.

For example, the primal exponential cone defined by the $\boldsymbol{x} \in \mathbb{R}^3$ :

$$K_{exp} = \text{cl}\left\{\boldsymbol{x} \in \mathbb{R}^3 \mid x_0 \geq x_1 \exp\left(\frac{x_2}{x_1}\right),\ x_1 > 0\right\}$$

Please refer to *Exponential Cone type* for more details.

If argument `vars` is a *ExpConeBuilder Class* object, then the value of argument `ctype` will be ignored.

If argument `vars` are variables, the optional values are *VarArray Class* objects, a one-dimensional *MVar Class* object, *tupledict Class* object, dictionary or list, the argument `ctype` specifies the type of the exponential cone and must be explicitly provided.

**Arguments**

`vars`

The variables forming the exponential cone.

Possible values are *ExpConeBuilder Class* object, *VarArray Class* object, one-dimensional *MVar Class* object, *tupledict Class* object, dictionary or list.

`ctype`

Type of the exponential cone. Please refer to *Exponential Cone type* for possible values.

**Example**

```
# Add a primal exponential cone formed by mx
mx = m.addMVar(3)
model.addExpCone(mx, ctype=COPT.EXPCONE_PRIMAL)
```

### Model.addExpCones()

**Synopsis**

```
addExpCones(vars, ctype)
```

**Description**

Add a batch of exponential cones to the model. Return a *ExpConeArray Class* object.

If argument `vars` is a *ExpConeBuilder Class* object, then the value of argument `ctype` will be ignored.

If argument `vars` is a *MVar Class* object, the `ctype` argument specifies the type of the exponential cones and must be explicitly provided.

**Arguments**

`vars`

Variables forming the exponential cones. Possible values are *ExpConeBuilderArray Class* object, 2-dimensional *MVar Class* object.

`ctype`

Type of the exponential cone. Please refer to *Exponential Cone type* for possible values.

**Example**

```
# Add two primal exponential cones formed by my
model.matrixmodelmode = 'experimental'
my = model.addMVar(shape=(2, 3))
model.addExpCones(my, COPT.EXPCONE_PRIMAL)
```

### Model.addAffineCone()

**Synopsis**

```
addAffineCone(exprs, ctype=None, name="")
```

**Description**

Adds an affine cone to the model.

If the argument `exprs` is a *AffineConeBuilder Class* object, the value of the argument `ctype` will be ignored.

If the argument `exprs` is an *MLinExpr Class* object or an *MPsdExpr Class* object, the argument `ctype` specifies the type of the affine cone and must be explicitly provided.

**Arguments**

`exprs`

An affine cone generator or a multi-dimensional array expression that forms the affine cone.

`ctype`

The type of the affine cone. Possible values are detailed in *SOC constraint types* or *Exponential Cone type*.

**Example**

```
# Add a standard second-order affine cone formed by Ax+b and c^Tx+d
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
b = 10
c = np.array([1, -1, 2])
d = 5
x = model.addMVar(3)
model.addAffineCone(cp.vstack(c @ x + d, A @ x + b), ctype=COPT.CONE_QUAD)
```

### Model.addAffineCones()

#### Synopsis

```
addAffineCones(exprs, ctype=None, nameprefix="AffineConV")
```

#### Description

Adds a set of affine cones to the model.

If the argument `exprs` is a *AffineConeBuilder Class* object or a *AffineConeBuilderArray Class* object, the value of the argument `ctype` is ignored.

If the argument `exprs` is an *MLinExpr Class* object or an *MPsdExpr Class* object, the argument `ctype` specifies the type of the affine cones and must be explicitly provided.

#### Arguments

`exprs`

Affine cone generators or multi-dimensional array expressions that form the affine cones.

`ctype`

The type of the affine cone. Possible values are detailed in *SOC constraint types* or *Exponential Cone type*.

### Model.addQConstr()

#### Synopsis

```
addQConstr(lhs, sense=None, rhs=None, name="")
```

#### Description

Add a linear or quadratic constraint, and return the added *Constraint Class* object or *QConstraint Class* object.

If the constraint is linear, then value of parameter `lhs` can be taken *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object; If the constraint is quadratic, then value of parameter `lhs` can be taken *QConstrBuilder Class* object, or *MQConstrBuilder Class* object and other parameters will be ignored.

#### Arguments

`lhs`

Left-hand side expression for new constraint or constraint builder.

`sense`

Sense for the new constraint. Optional, None by default. Please refer to *Constraint type* for possible values.

`rhs`

Right-hand side expression for the new constraint. Optional, None by default. It can be a constant, *Var Class* object, *LinExpr Class* object or *QuadExpr Class* object.

`name`

Name for new constraint. Optional, `""` by default, generated by solver automatically.

#### Example

```
# add a linear equality: x + y == 2
m.addQConstr(x + y, COPT.EQUAL, 2)
# add a quadratic inequality: x*x + y*y <= 3
m.addQConstr(x*x + y*y <= 3.0)
```

### Model.addMQConstr()

#### Synopsis

```
addMQConstr(Q, c, sense, rhs, xQ_L=None, xQ_R=None, xc=None,
name="")
```

#### Description

By means of matrix modeling, a quadratic constraint is added to the model. If the value of `sense` here is `COPT.LESS_EQUAL` , the added constraint is $x_{Q_L} Q x_{Q_R} + cx_c <= rhs$.

It is more convenient to generate *MQuadExpr Class* objects by matrix multiplication, and then use overloaded comparison operators to generate *MQConstrBuilder Class* object, which can be used as input of `Model.addQConstr()` to generate quadratic constraints.

#### Arguments

`Q`

If the quadratic term is not empty, the parameter Q needs to be provided, which is a two-dimensional NumPy matrix, SciPy compressed sparse column matrix ( `csc_matrix` ) or compressed sparse row matrix ( `csr_matrix` ).

`c`

If the item is non-empty, you need to provide the parameter c, which is a one-dimensional NumPy array, or a Python list.

`sense`

The type of constraint. Possible values refer to *Constraint type*.

`rhs`

The value on the right side of the constraint, usually a floating point number.

`xQ_L`

The variable on the left side of the quadratic term, can be a *MVar Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

If empty, all variables in the model are taken.

`xQ_R`

The variable on the right side of the quadratic term, can be a *MVar Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

If empty, all variables in the model are taken.

`xc`

The variable corresponding to the linear term can be a *MVar Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

If it is empty, but the parameter c is not empty, all variables in the model are taken.

`name`

constraint name.

**Return value**

Returns a *QConstraint Class* object

**Example**

```
Q = np.full((3, 3), 1)
mx = model.addMVar(3, nameprefix="mx")
mqc = model.addMQConstr(Q, None, 'L', 1.0, mx, mx, None, name="mqc")
```

### Model.addNlConstr()

**Synopsis**

`addNlConstr(lhs, sense=None, rhs=None, name="")`

**Description**

Add a nonlinear constraint to the model.

If `lhs` is a nonlinear expression type, both `sense` and `rhs` must be provided to specify the constraint direction and right-hand side.

If `lhs` is a *NlConstrBuilder Class* object, the constraint is created using the builder content. Other arguments are ignored.

If `lhs` is a *ConstrBuilder Class* object, a linear constraint is added. The return value is *Constraint Class*.

If `lhs` is a *QConstrBuilder Class* object, a quadratic constraint is added. The return value is *QConstraint Class*.

**Arguments**

`lhs`

The left-hand side expression or constraint builder.

`sense`

The constraint type. Optional. Defaults to `None`.

See *Constraint Types* for valid values.

`rhs`

The right-hand side of the constraint. Optional. Defaults to `None`.

Can be a constant or a *NlExpr Class* object.

`name`

The name of the constraint. Optional.

Defaults to `""`, and a name will be generated automatically by the solver.

### Model.addNlConstrs()

**Synopsis**

```
addNlConstrs(generator, nameprefix="NR")
```

**Description**

Add a set of nonlinear constraints to the model.

**Arguments**

`generator`

A constraint builder or a sequence of nonlinear constraint builders.

Possible values include: a single *NlConstrBuilder Class*, *QConstrBuilder Class*, or *ConstrBuilder Class* object, or an iterable of *NlConstrBuilder Class* objects.

`nameprefix`

The prefix used for naming constraints.

Optional. Defaults to `"NR"`.

### Model.addPsdVar()

**Synopsis**

```
addPsdVar(dim, name="")
```

**Description**

Add a positive semi-definite variable.

**Arguments**

`dim`

Dimension for the positive semi-definite variable.

`name`

Name for the positive semi-definite variable.

**Example**

```
# Add a three-dimensional positive semi-definite variable, "X"
m.addPsdVar(3, "X")
```

### Model.addPsdVars()

**Synopsis**

```
addPsdVars(dims, nameprefix="PSDV")
```

**Description**

Add multiple new positive semi-definite variables to a model.

**Arguments**

`dim`

Dimensions for new positive semi-definite variables.

`nameprefix`

Name prefix for new positive semi-definite variables.

**Example**

```
# Add two three-dimensional positive semi-definite variables
m.addPsdVars([3, 3])
```

### Model.addUserCut()

**Synopsis**

```
addUserCut(lhs, sense = None, rhs = None, name="")
```

**Description**

Add a user cut to the MIP model.

**Arguments**

`lhs`

Left-hand side expression for the new user cut. It can take the value of *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object.

`sense`

The sense of the new user cut. Please refer to *Constraint type* for possible values.

Optional. None by default.

`rhs`

Right-hand side expression for the new user cut.

Optional. None by default.

It can be a constant, or *Var Class* object, or *LinExpr Class* object.

`name`

Name for the new user cut. Optional, `""` by default, automatically generated by solver.

**Example**

```
model.addUserCut(x+y <= 1)

model.addUserCut(x+y == [0, 1])
```

### Model.addUserCuts()

**Synopsis**

```
addUserCuts(generator, nameprefix="U")
```

**Description**

Add a set of user cuts to the MIP model.

**Arguments**

`generator`

A generator expression, where each iteration produces a *Constraint Class* object, or *MConstrBuilder Class* object.

`nameprefix`

Name prefix for new user cuts. Optional, `"U"` by default. The actual name and the index of the constraints are automatically generated by COPT.

**Example**

```
model.addUserCuts(x[i]+y[i] <= 1 for i in range(10))
```

### Model.addLazyConstr()

**Synopsis**

```
addLazyConstr(lhs, sense = None, rhs = None, name="")
```

**Description**

Add a lazy constraint to the MIP model.

**Arguments**

`lhs`

Left-hand side expression for the new lazy constraint. It can take the value of *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object.

`sense`

The sense of the lazy constraint. Please refer to *Constraint type* for possible values.

Optional. None by default.

`rhs`

Right-hand side expression for the new lazy constraint.

Optional. None by default.

It can be a constant, or *Var Class* object, or *LinExpr Class* object.

`name`

Name for the new lazy constraint. Optional, `""` by default, automatically generated by solver.

**Example**

```
model.addLazyConstr(x+y <= 1)
model.addLazyConstr(x+y == [0, 1])
```

### Model.addLazyConstrs()

**Synopsis**

```
addLazyConstrs(generator, nameprefix="L")
```

**Description**

Add a set of lazy constraints to the MIP model.

**Arguments**

`generator`

A generator expression, where each iteration produces a *Constraint Class* object, or *MConstrBuilder Class* object.

`nameprefix`

Name prefix for new lazy constraints. Optional, "L" by default. The actual name and the index of the constraints are automatically generated by COPT.

**Example**

```
model.addLazyConstrs(x[i]+y[i] <= 1 for i in range(10))
```

### Model.addSparseMat()

**Synopsis**

```
addSparseMat(dim, rows, cols=None, vals=None)
```

**Description**

Add a sparse symmetric matrix in triplet format

**Arguments**

`dim`

Dimension for the matrix.

`rows`

Row indices for accessing rows of non-zero elements.

`cols`

Column indices for accessing columns of non-zero elements.

`vals`

Coefficient values for non-zero elements

**Example**

```
# Add a three-dimensional symmetric matrix
m.addSparseMat(3, [0, 1, 2], [0, 1, 2], [2.0, 5.0, 8.0])
# Add a two-dimensional symmetric matrix
m.addSparseMat(2, [(0, 0, 3.0), (1, 0, 1.0)])
```

### Model.addDenseMat()

**Synopsis**

```
addDenseMat(dim, vals)
```

**Description**

Add a dense symmetric matrix

**Arguments**

`dim`

Dimension for the matrix.

`vals`

Coefficient values, which can be a constant or a list.

**Example**

```
# Add a three-dimensional matrix (filled with ones)
m.addDenseMat(3, 1.0)
```

### Model.addDiagMat()

**Synopsis**

```
addDiagMat(dim, vals, offset=None)
```

**Description**

Add a diagonal symmetric matrix

**Arguments**

`dim`

Dimension for the matrix.

`vals`

Coefficient values, which can be a constant or a list.

`offset`

Offset of diagonal elements. Positive: above the diagonal; Negative: below the diagonal

**Example**

```
# Add a three-dimensional identity matrix
m.addDiagMat(3, 1.0)
```

### Model.addOnesMat()

**Synopsis**

```
addOnesMat(dim)
```

**Description**

Add a matrix filled with ones.

**Arguments**

`dim`

Dimension for the matrix.

**Example**

```
# Add a three-dimensional matrix (filled with ones)
m.addOnesMat(3)
```

### Model.addEyeMat()

**Synopsis**

```
addEyeMat(dim)
```

**Description**

Add an identity matrix

**Arguments**

`dim`

Dimension for the matrix.

**Example**

```
# Add a three-dimensional identity matrix
m.addEyeMat(3)
```

### Model.setObjective()

#### Synopsis

```
setObjective(expr, sense=None)
```

#### Description

Set the model objective.

#### Arguments

`expr`

Objective expression. Argument can be a constant, *Var Class* object, *LinExpr Class* object, *QuadExpr Class* object, *MLinExpr Class* object, or *MQuadExpr Class* object, or *NlExpr Class* object.

Note: If `expr` is a *LinExpr Class* object, the linear term in the objective will be updated; If it is a *QuadExpr Class* object, both the quadratic and linear terms in the objective will be updated.

`sense`

Optimization sense. Optional, `None` by default, which means no change to objective sense. Users can get access to current objective sense by attribute `ObjSense` . Please refer to *Optimization directions* for possible values.

#### Example

```
# Set objective function = x+y, optimization sense is maximization.
m.setObjective(x + y, COPT.MAXIMIZE)
```

### Model.setObjectiveN()

#### Synopsis

```
setObjectiveN(idx, expr, sense=None, priority=0.0, weight=1.0,
abstol=1e-6, reltol=0.0)
```

#### Description

Specify the objective function for multi-objective optimization.

#### Arguments

`idx`

The index of the objective function. Argument can be an integer constant.

`expr`

The expression of the objective function. Argument can be a linear expression.

`sense`

Optimization sense.

Optional, `None` by default, which means no change to objective sense. Please refer to *Optimization directions* for possible values.

`priority`

The priority of the objective function.

Optional, default is 0.0.

`weight`

The weight of the objective function.

Optional, default is 1.0.

`abstol`

The absolute tolerance for degradation.

Optional, default is 1e-6.

`reltol`

The relative tolerance for degradation.

Optional, default is 0.0.

### Model.setMObjective()

#### Synopsis

```
setMObjective(Q, c, constant, xQ_L=None, xQ_R=None, xc=None,
sense=None)
```

#### Description

Set the secondary objective of the model by matrix modeling. Objective functions of the form $x_{Q_L} Q x_{Q_R} + c x_c + constant$ can be added.

Even more convenient is to generate a *MQuadExpr Class* object via matrix multiplication, available as the input to `setObjective()` to set the objective function.

#### Arguments

`Q`

If the quadratic term is not empty, the parameter Q needs to be provided, which is a two-dimensional NumPy matrix, SciPy compressed sparse column matrix ( `csc_matrix` ) or compressed sparse row matrix ( `csr_matrix` ).

`c`

If the item is non-empty, you need to provide the parameter c, which is a one-dimensional NumPy array, or a Python list.

`constant`

Constant term, usually a floating point number.

`xQ_L`

The variable on the left side of the quadratic term, can be a *MVar Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

If empty, all variables in the model are taken.

`xQ_R`

The variable on the right side of the quadratic term, can be a *MVar Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

If empty, all variables in the model are taken.

`xc`

The variable corresponding to the linear term can be a *MVar Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

If it is empty, but the parameter c is not empty, all variables in the model are taken.

`sense`

The optimization direction of the objective function. Optional parameter, the default is `None`, which means that the optimization direction of the model will not be changed. The current optimization direction of the model is viewed through the property `ObjSense`. See *Optimization Direction* for possible values.

**Example**

```
Q = np.full((3, 3), 1)
mx = model.addMVar(3, nameprefix="mx")
my = model.addVars(3, nameprefix="my")
mqc = model.setMObjective(Q, None, 0.0, mx, my, None, sense=COPT.MINIMIZE)
```

### Model.setObjSense()

**Synopsis**

`setObjSense(sense)`

**Description**

Set optimization sense.

**Arguments**

`sense`

Optimization sense. Please refer to *Optimization directions* for possible values.

**Example**

```
# Set optimization sense as maximization
m.setObjSense(COPT.MAXIMIZE)
```

### Model.setObjConst()

**Synopsis**

`setObjConst(constant)`

**Description**

Set constant objective offset.

**Arguments**

`constant`

Constant objective offset.

**Example**

```
# Set constant objective offset 1.0
m.setObjConst(1.0)
```

### Model.getObjective()

**Synopsis**

```
getObjective()
```

**Description**

Retrieve current model objective. Return a *LinExpr Class* object.

**Example**

```
# Retrieve the optimization objective.
obj = m.getObjective()
```

### Model.getObjectiveN()

**Synopsis**

```
getObjectiveN(idx)
```

**Description**

Retrieve the expression of the objective function with the specified index in multi-objective optimization.

**Arguments**

`idx`

The index of the objective function.

**Return value**

Returns a *LinExpr Class* object.

### Model.delObjN()

**Synopsis**

```
delObjN(idx)
```

**Description**

Delete the linear part of the objective with the specified index from the multi-objective model.

**Arguments**

`idx`

The index of the objective function.

### Model.delQuadObj()

**Synopsis**

```
delQuadObj()
```

**Description**

Deletes the quadratic terms from the quadratic objective function.

**Example**

```
# Deletes the quadratic terms from the quadratic objective function
m.delQuadObj()
```

### Model.delNlObj()

**Synopsis**

```
delNlObj()
```

**Description**

Delete all nonlinear terms from the nonlinear objective function.

**Example**

```
# Delete nonlinear terms from the objective
m.delNlObj()
```

### Model.delPsdObj()

**Synopsis**

```
delPsdObj()
```

**Description**

Delete the positive semi-definite terms from the objective function

**Example**

```
# Delete the positive semi-definite terms from the objective function
m.delPsdObj()
```

### Model.getCol()

**Synopsis**

```
getCol(var)
```

**Description**

Retrieve the list of constraints in which a variable participates, Return value is a *Column Class* object that captures the set of constraints in which the variable participates.

**Example**

```
# Get column that captures the set of constraints in which x participates.
col = m.getCol(x)
```

### Model.getRow()

**Synopsis**

```
getRow(constr)
```

**Description**

Retrieve the list of variables that participate in a specific constraint and return a *LinExpr Class* object.

**Example**

```
# Return variables that participate in conx.
linexpr = m.getRow(conx)
```

### Model.getQuadRow()

**Synopsis**

```
getQuadRow(qconstr)
```

**Description**

Retrieve the list of variables that participate in a specific quadratic constraint and return a *QuadExpr Class* object.

**Example**

```
# Return variables that participate in qconx
quadexpr = m.getQuadRow(qconx)
```

### Model.getPsdRow()

**Synopsis**

```
getPsdRow(constr)
```

**Description**

Retrieve the list of variables that participate in a specific positive semi-definite constraint and return a *PsdExpr Class* object.

**Example**

```
# Retrieve the row corresponding to a specific positive semi-definite constraint
psdexpr = m.getPsdRow(psdcon)
```

### Model.getNlRow()

**Synopsis**

```
getNlRow(constr)
```

**Description**

Retrieve the row expression associated with the specified nonlinear constraint. Returns a *NlExpr Class* object.

**Example**

```
# Retrieve the row expression of nonlinear constraint
nlexpr = m.getNlRow(nlcon)
```

### Model.getVar()

**Synopsis**

```
getVar(idx)
```

**Description**

Retrieve a variable according to its index in the coefficient matrix. Return a *Var Class* object.

**Arguments**

`idx`

Index of the desired variable in the coefficient matrix, starting with `0`.

**Example**

```
# Retrieve variable with index of 1.
 x = m.getVar(1)
```

### Model.getVarByName()

**Synopsis**

`getVarByName(name)`

**Description**

Retrieves a variable by name. Return a *Var Class* object.

**Arguments**

`name`

Name of the desired variable.

**Example**

```
#  Retrieve variable with name "x".
x = m.getVarByName("x")
```

### Model.getVars()

**Synopsis**

`getVars()`

**Description**

Retrieve all variables in the model. Return a *VarArray Class* object.

**Example**

```
# Retrieve all variables in the model
vars = m.getVars()
```

### Model.getConstr()

**Synopsis**

`getConstr(idx)`

**Description**

Retrieve a constraint by its index in the coefficient matrix. Return a *Constraint Class* object.

**Arguments**

`idx`

Index of the desired constraint in the coefficient matrix, starting with `0`.

**Example**

```
# Retrieve linear constraint with index of 1.
r = m.getConstr(1)
```

### Model.getConstrByName()

**Synopsis**

`getConstrByName(name)`

**Description**

Retrieves a linear constraint by name. Return a *Constraint Class* object.

**Arguments**

`name`

The name of the constraint.

**Example**

```
# Retrieve linear constraint with name "r".
r = m.getConstrByName("r")
```

### Model.getConstrs()

**Synopsis**

`getConstrs()`

**Description**

Retrieve all constraints in the model. Return a *ConstrArray Class* object.

**Example**

```
# Retrieve all constraints in the model
cons = m.getConstrs()
```

### Model.getConstrBuilders()

**Synopsis**

`getConstrBuilders(constrs=None)`

**Description**

Retrieve linear constraint builders in current model.

If parameter `constrs` is `None`, then return a *ConstrBuilderArray Class* object composed of all linear constraint builders.

If parameter `constrs` is *Constraint Class* object, then return the *ConstrBuilder Class* object corresponding to the specific constraint.

If parameter `constrs` is a list or a *ConstrArray Class* object, then return a *ConstrBuilderArray Class* object composed of specified constraints' builders.

If parameter `constrs` is dictionary or *tupledict Class* object, then the index of the specified constraint is returned as key, the value is a *tupledict Class* object composed of the specified constraints' builders.

**Arguments**

`constrs`

The specified linear constraint. Optional, `None` by default.

**Example**

```
# Retrieve all of linear constraint builders.
conbuilders = m.getConstrBuilders()
# Retrieve the builder corresponding to linear contstraint x.
conbuilders = m.getConstrBuilders(x)
# Retrieve builders corresponding to linear constraint x and y.
conbuilders = m.getConstrBuilders([x, y])
# Retrieve builders corresponding to linear constraint in tupledict object xx.
conbuilders = m.getConstrBuilders(xx)
```

### Model.getNlConstr()

**Synopsis**

`getNlConstr(idx)`

**Description**

Retrieve a nonlinear constraint by its index in the model. Return a *NlConstraint Class* object.

**Arguments**

`idx`

Index of the nonlinear constraint in the model, starting with `0`.

**Example**

```
# Retrieve nonlinear constraint with index 1
r = m.getNlConstr(1)
```

### Model.getNlConstrByName()

**Synopsis**

`getNlConstrByName(name)`

**Description**

Retrieve a nonlinear constraint by its name. Return a *NlConstraint Class* object.

**Arguments**

`name`

Name of the nonlinear constraint.

**Example**

```
# Retrieve nonlinear constraint with name "r"
r = m.getNlConstrByName("r")
```

### Model.getNlConstrs()

**Synopsis**

`getNlConstrs()`

**Description**

Retrieve all nonlinear constraints in the model. Return a *NlConstrArray Class* object.

**Example**

```
# Retrieve all nonlinear constraints in the model
cons = m.getNlConstrs()
```

### Model.getNlConstrBuilders()

**Synopsis**

`getNlConstrBuilders(constrs=None)`

**Description**

Retrieve nonlinear constraint builders in the current model.

If parameter `constrs` is `None`, return a *NlConstrBuilderArray Class* object composed of all nonlinear constraint builders.

If parameter `constrs` is a *NlConstraint Class* object, return the *NlConstrBuilder Class* object corresponding to the specified constraint.

If parameter `constrs` is a list or a *NlConstrArray Class* object, return a *NlConstrBuilderArray Class* object composed of the specified constraints' builders.

If parameter `constrs` is a dictionary or *tupledict Class* object, return a *tupledict Class* where keys are the same as the input, and values are the corresponding nonlinear constraint builders.

**Arguments**

`constrs`

The specified nonlinear constraint(s).

Optional, `None` by default.

**Example**

```
# Retrieve all nonlinear constraint builders
conbuilders = m.getNlConstrBuilders()
# Retrieve the builder corresponding to nonlinear constraint nl
conbuilder = m.getNlConstrBuilders(nl)
# Retrieve builders corresponding to nonlinear constraints nl1 and nl2
conbuilders = m.getNlConstrBuilders([nl1, nl2])
# Retrieve builders corresponding to nonlinear constraints in a tupledict
conbuilders = m.getNlConstrBuilders(td)
```

### Model.getSOS(sos)

**Synopsis**

`getSOS(sos)`

**Description**

Retrieve the SOS constraint builder corresponding to specific SOS constraint. Return a *SOSBuilder Class* object

**Arguments**

`sos`

The specified SOS constraint.

**Example**

```
# Retrieve the builder corresponding to SOS constraint sosx.
sosbuilder = m.getSOS(sosx)
```

### Model.getSOSs()

**Synopsis**

`getSOSs()`

**Description**

Retrieve all SOS constraints in model and return a *SOSArray Class* object.

**Example**

```
#  Retrieve all SOS constraints in model.
soss = m.getSOSs()
```

### Model.getSOSBuilders()

**Synopsis**

`getSOSBuilders(soss=None)`

**Description**

Retrieve the SOS constraint builder corresponding to the specified SOS constraint.

If parameter `soss` is `None`, then return a *SOSBuilderArray Class* object consisting of builders corresponding to all SOS constraints.

If parameter `soss` is *SOS Class* object, then return a *SOSBuilder Class* corresponding to the specified SOS constraint.

If parameter `soss` is list or *SOSArray Class* object, then return a *SOSBuilderArray Class* object consisting of builders corresponding to the specific SOS constraints.

**Arguments**

`soss`
The specific SOS constraint. Optional, `None` by default.

**Example**

```
# Retrieve builders corresponding to all SOS constraints in the model.
soss = m.getSOSBuilders()
```

### Model.getGenConstrIndicator()

**Synopsis**

```
getGenConstrIndicator(genconstr)
```

**Description**

Retrieve the builder corresponding to specific indicator constraint. Return a *GenConstrBuilder Class* object.

**Arguments**

`genconstr`

The specified indicator constraint.

**Example**

```
# Retrieve the builder corresponding to indicator constraint genx.
indic = m.getGenConstrIndicator(genx)
```

### Model.getGenConstr()

**Synopsis**

```
getGenConstr(idx)
```

**Description**

Retrieve an indicator constraint by its index in the model. Return a *GenConstr Class* object.

**Arguments**

`idx`

Index of the desired constraint in the model, starting with `0`.

**Example**

```
# Retrieve indicator constraint with index of 0
genx = m.getGenConstr(0)
```

### Model.getGenConstrs()

**Synopsis**

```
getGenConstrs()
```

**Description**

Retrieve all indicator constraints in the model. Return a *GenConstrArray Class* object.

**Example**

```
# Retrieve all indicator constraints in the model
cons = m.getGenConstrs()
```

### Model.getGenConstrByName()

**Synopsis**

```
getGenConstrByName(name)
```

**Description**

Retrieves an indicator constraint by the specified name. Return a *GenConstr Class* object.

**Arguments**

`name`

The name of the indicator constraint.

**Example**

```
# Retrieve indicator constraint with name "r"
r = m.getGenConstrByName("r")
```

### Model.getGenConstrIndicators()

**Synopsis**

```
getGenConstrIndicators(genconstrs=None)
```

**Description**

Retrieve the specified indicator constraints' builders in the model. All indicator constraints will be retrieved in default. Return a *GenConstrBuilder Class* or *GenConstrBuilderArray Class* object.

**Example**

```
# Retrieve all indicator constraints' builders in the model
cons = m.getGenConstrIndicators()
```

### Model.getCone()

**Synopsis**

```
getCone(idx)
```

**Description**

Retrieves the second-order cone at the specified index in the model. Returns a *Cone Class* object.

**Arguments**

`idx`

The specified index. Indexing starts at 0.

**Example**

```
# Retrieve the second-order cone at index 1 in the model
cones = m.getCone(1)
```

### Model.getExpCone()

**Synopsis**

```
getExpCone(idx)
```

**Description**

Retrieves the exponential cone at the specified index in the model. Returns an *ExpCone Class* object.

**Arguments**

`idx`

The specified index. Indexing starts at 0.

**Example**

```
# Retrieve the exponential cone at index 1 in the model
cones = m.getExpCone(1)
```

### Model.getAffineCone()

**Synopsis**

```
getAffineCone(idx)
```

**Description**

Retrieves the affine cone at the specified index in the model. Returns an *AffineCone Class* object.

**Arguments**

`idx`

The specified index. Indexing starts at 0.

**Example**

```
# Retrieve the affine cone at index 1 in the model
cones = m.getAffineCone(1)
```

### Model.getAffineConeByName()

**Synopsis**

```
getAffineConeByName(name)
```

**Description**

Retrieves the affine cone with the specified name in the model. Returns a *AffineCone Class* object.

**Arguments**

`name`

The specified name.

**Example**

```
# Retrieve the affine cone with the name "afcone" in the model
cones = m.getAffineConeByName("afcone")
```

### Model.getCones()

**Synopsis**

```
getCones()
```

**Description**

Retrieve all Second-Order-Cone (SOC) constraints in model, and return a *ConeArray Class* object.

**Example**

```
# Retrieve all SOC constraints
cones = m.getCones()
```

### Model.getExpCones()

**Synopsis**

```
getExpCones()
```

**Description**

Retrieve all exponential cone constraints in model, and return a *ExpConeArray Class* object.

**Example**

```
# Retrieve all exponential cone constraints
cones = m.getExpCones()
```

### Model.getAffineCones()

**Synopsis**

```
getAffineCones()
```

**Description**

Retrieves all affine cones in the model. Returns a *AffineConeArray Class* object.

**Example**

```
# Retrieve all affine cones in the model
cones = m.getAffineCones()
```

### Model.getConeBuilders()

**Synopsis**

```
getConeBuilders(cones=None)
```

**Description**

Retrieve Second-Order-Cone (SOC) constraint builders for given SOC constraints.

If argument `cones` is `None`, then return a *ConeBuilderArray Class* object consists of all SOC constraints' builders; If argument `cones` is *Cone Class* object, then return a *ConeBuilder Class* object of given SOC constraint; If `cones` is Python list or *ConeArray Class* object, then return a *ConeBuilderArray Class* object consists of builders of given SOC constraints.

**Arguments**

`cones`

Given SOC constraints. Optional, default to `None`.

**Example**

```
# Retrieve all SOC constraints' builders
cones = m.getConeBuilders()
```

### Model.getExpConeBuilders()

**Synopsis**

`getExpConeBuilders(cones=None)`

**Description**

Retrieve the exponential cone constraint builders corresponding to the specified exponential cone constraint.

If argument `cones` is `None`, then return a *ExpConeBuilderArray Class* object consists of all exponential cone constraints' builders; If argument `cones` is *ExpCone Class* object, then return a *ExpConeBuilder Class* object of given exponential cone constraints; If `cones` is Python list or *ExpConeArray Class* object, then return a *ExpConeBuilderArray Class* object consists of builders of given exponential cone constraints.

**Arguments**

`cones`

Given exponential cone constraints. Optional, default to `None`.

**Example**

```
# Retrieve all exponential cone constraints' builders
cones = m.getExpConeBuilders()
```

### Model.getAffineConeBuilders()

**Synopsis**

`getAffineConeBuilders(cones=None)`

**Description**

Retrieves the affine cone builders corresponding to the specified affine cones.

If the argument `cones` is `None`, returns a *AffineConeBuilderArray Class* object containing the builders for all affine cones in the model.

If the argument `cones` is a *AffineCone Class* object, returns the corresponding *AffineConeBuilder Class* object.

If the argument `cones` is a list or a *AffineConeArray Class* object, returns a *AffineConeBuilderArray Class* object containing the builders for the specified affine cones.

**Arguments**

`cones`

The specified affine cones. Optional, defaulting to `None`.

**Example**

```
# Retrieve the builders for all affine cones in the model
cones = m.getAffineConeBuilders()
```

### Model.getQConstr()

**Synopsis**

```
getQConstr(idx)
```

**Description**

Retrieve a quadratic constraint by its index, and return a *QConstraint Class* object.

**Arguments**

`idx`

Index of the desired quadratic constraint, starting with `0`.

**Example**

```
# Retrieve a quadratic constraint with index of 1
qr = m.getQConstr(1)
```

### Model.getQConstrByName()

**Synopsis**

```
getQConstrByName(name)
```

**Description**

Retrieve a quadratic constraint by its name, and return a *QConstraint Class* object.

**Arguments**

`name`

Name of the desired quadratic constraint.

**Example**

```
# Retrieve a quadratic constraint with name "qr"
qr = m.getQConstrByName("qr")
```

### Model.getQConstrs()

**Synopsis**

```
getQConstrs()
```

**Description**

Retrieve all quadratic constraints in the model. Return a *QConstrArray Class* object.

**Example**

```
# Retrieve all quadratic constraints in the model
qcons = m.getQConstrs()
```

### Model.getQConstrBuilders()

**Synopsis**

```
getQConstrBuilders(qconstrs=None)
```

**Description**

Retrieve quadratic constraint builders in current model.

If parameter `qconstrs` is `None`, then return a *QConstrBuilderArray Class* object composed of all quadratic constraint builders.

If parameter `qconstrs` is *QConstraint Class* object, then return the *QConstrBuilder Class* object corresponding to the specific quadratic constraint.

If parameter `qconstrs` is a list or a *QConstrArray Class* object, then return a *QConstrBuilderArray Class* object composed of specified quadratic constraints' builders.

If parameter `qconstrs` is dictionary or *tupledict Class* object, then the index of the specified quadratic constraint is returned as key, the value is a *tupledict Class* object composed of the specified quadratic constraints' builders.

**Arguments**

`qconstrs`

The specified quadratic constraint. Optional, `None` by default.

**Example**

```
# Retrieve all of quadratic constraint builders.
qconbuilders = m.getQConstrBuilders()
# Retrieve the builder corresponding to quadratic contstraint qx.
qconbuilders = m.getQConstrBuilders(qx)
# Retrieve builders corresponding to quadratic constraint qx and qy.
qconbuilders = m.getQConstrBuilders([qx, qy])
# Retrieve builders corresponding to quadratic constraint in tupledict object qxx.
qconbuilders = m.getQConstrBuilders(qxx)
```

### Model.getPsdVar()

**Synopsis**

```
getPsdVar(idx)
```

**Description**

Retrieve a positive semi-definite variable according to its index in the model. Return a *PsdVar Class* object.

**Arguments**

`idx`

Index of the desired positive semi-definite variable in the model, starting with `0`.

**Example**

```
# Retrieve a positive semi-definite variable with index of 1
x = m.getPsdVar(1)
```

### Model.getPsdVarByName()

**Synopsis**

```
getPsdVarByName(name)
```

**Description**

Retrieve a positive semi-definite variable by name. Return a *PsdVar Class* object.

**Arguments**

`name`

The name of the positive semi-definite variable.

**Example**

```
# Retrieve a positive semi-definite variable with name "x".
x = m.getPsdVarByName("x")
```

### Model.getPsdVars()

**Synopsis**

```
getPsdVars()
```

**Description**

Retrieve all positive semi-definite variables in the model, and return a *PsdVarArray Class* object.

**Example**

```
# Retrieve all positive semi-definite variables in the model.
vars = m.getPsdVars()
```

### Model.getPsdConstr()

**Synopsis**

```
getPsdConstr(idx)
```

**Description**

Retrieve the positive semi-definite constraint according to its index in the model. Return a *PsdConstraint Class* object.

**Arguments**

`idx`

Index for the positive semi-definite constraint, starting with 0.

**Example**

```
# Retrieve the positive semi-definite constraint with index of 1
r = m.getPsdConstr(1)
```

### Model.getPsdConstrByName()

**Synopsis**

```
getPsdConstrByName(name)
```

**Description**

Retrieve a positive semi-definite constraint by name. Return a *PsdConstraint Class* object.

**Arguments**

`name`

The name of the positive semi-definite constraint.

**Example**

```
# Retrieve the positive semi-definite constraint with name "r".
r = m.getPsdConstrByName("r")
```

### Model.getPsdConstrs()

**Synopsis**

```
getPsdConstrs()
```

**Description**

Retrieve all positive semi-definite constraints in the model. Return a *PsdConstrArray Class* object.

**Example**

```
# Retrieve all positive semi-definite constraints in the model
cons = m.getPsdConstrs()
```

### Model.getPsdConstrBuilders()

**Synopsis**

```
getPsdConstrBuilders(constrs=None)
```

**Description**

Retrieve positive semi-definite constraint builders in current model.

If parameter `constrs` is `None`, then return a *PsdConstrBuilderArray Class* object composed of all positive semi-definite constraint builders.

If parameter `constrs` is *PsdConstraint Class* object, then return the *PsdConstrBuilder Class* object corresponding to the specific positive semi-definite constraint.

If parameter `constrs` is a list or a *PsdConstrArray Class* object, then return a *PsdConstrBuilderArray Class* object composed of specified positive semi-definite constraints' builders.

If parameter `constrs` is dictionary or *tupledict Class* object, then the index of the specified positive semi-definite constraint is returned as key, the value is a *tupledict Class* object composed of the specified positive semi-definite constraints' builders.

**Arguments**

`constrs`

The specified positive semi-definite constraint. Optional, `None` by default.

**Example**

```
# Retrieve all of positive semi-definite constraint builders.
conbuilders = m.getPsdConstrBuilders()
# Retrieve the builder corresponding to positive semi-definite contstraint x.
conbuilders = m.getPsdConstrBuilders(x)
# Retrieve builders corresponding to positive semi-definite constraint x and y.
conbuilders = m.getPsdConstrBuilders([x, y])
# Retrieve builders corresponding to positive semi-definite constraint in tupledict␣
↪object xx.
conbuilders = m.getPsdConstrBuilders(xx)
```

### Model.getLmiRow()

**Synopsis**

```
getLmiRow(constr)
```

**Description**

Get the LMI expression involved in the specified LMI constraint, including variables and corresponding coefficient matrices.

**Arguments**

`constr`

The specified LMI constraint.

**Example**

```
# Get the expression in LMI constraint c
expr = m.getLmiRow(c)
```

### Model.getLmiConstr()

**Synopsis**

```
getLmiConstr(idx)
```

**Description**

Get the LMI constraint corresponding to the specified index in the model.

**Arguments**

`idx`

The index of the LMI constraint in the model. Starts with `0`.

**Example**

```
# Get the 1st LMI constraint in the model
coeff = m.getLmiConstr(1)
```

### Model.getLmiConstrByName()

**Synopsis**

```
getLmiConstrByName(name)
```

**Description**

Get the LMI constraint of the specified name in the model.

**Arguments**

`name`

The specified name of the LMI constraint.

**Example**

```
# Get the LMI constraint named r1 in the model
name = m.getLmiConstrByName("r1")
```

### Model.getLmiConstrs()

**Synopsis**

```
getLmiConstrs()
```

**Description**

Get all LMI constraints in the model. Returns a *LmiConstrArray Class* object composed of LMI constraints.

### Model.getLmiRhs()

**Synopsis**

```
getLmiRhs(constr)
```

**Description**

Get the constant term of the specified LMI constraint. Returns a *SymMatrix Class* object.

**Arguments**

`constr`

The specified LMI constraint.

### Model.setLmiRhs()

**Synopsis**

```
setLmiRhs(constr, mat)
```

**Description**

Set the constant term of the specified LMI constraint.

**Arguments**

`constr`

The specified LMI constraint.

`mat`

The new constant-term symmetric to set.

**Example**

```
# Set the constant-term symmetric of the LMI constraint con to D
m.setLmiRhs(con, D)
```

### Model.getLmiSolution()

**Synopsis**

```
getLmiSolution()
```

**Description**

Get the value and dual value of the LMI constraint.

### Model.getLmiSlacks()

**Synopsis**

```
getLmiSlacks()
```

**Description**

Get the values of all slack variables of LMI constraints. Returns a list object.

### Model.getLmiDuals()

**Synopsis**

```
getLmiDuals()
```

**Description**

Get the values of all dual variables of the LMI constraint. Returns a list object.

### Model.getCoeff()

**Synopsis**

```
getCoeff(constr, var)
```

**Description**

Get the coefficient of variable in linear constraint, PSD constraint or LMI constraint.

**Arguments**

`constr`

The requested linear constraint, PSD constraint or LMI constraint.

`var`

The requested variable or PSD variable.

**Example**

```
# Get the coefficient of variable x in linear constraint c1
coeff1 = m.getCoeff(c1, x)
# Get the coefficient of PSD variable X in PSD constraint c2
coeff2 = m.getCoeff(c2, X)
```

### Model.setCoeff()

**Synopsis**

```
setCoeff(constr, var, newval)
```

**Description**

Set the coefficient of variable in linear constraint, PSD constraint or LMI constraint.

**Arguments**

`constr`

The requested linear constraint, PSD constraint or LMI constraint.

`var`

The requested variable or PSD variable.

`newval`

New coefficient or symmetric matrix coefficient.

**Example**

```
# Set the coefficient of variable x in linear constraint c to 1.0
m.setCoeff(c, x, 1.0)
```

### Model.setCoeffs()

**Synopsis**

```
setCoeffs(constrs, vars, newvals)
```

**Description**

Set the coefficients of variables in the linear constraints in batches.

**Note** The constraint and variable pair cannot repeat with the same or different coefficient to be set.

**Arguments**

`constrs`

Specifies the constraints related to the coefficients to be set, which could be a dictionary, *tupledict Class* , *ConstrArray Class* or a list of *Constraint Class* objects.

`vars`

Specifies the variables related to the coefficients to be set, which could be a dictionary, *tupledict Class* , *VarArray Class* or a list of *Var Class* objects.

`newvals`

The new coefficient values to be set, which could be a constant, or the list/dictionary corresponding to the `constrs` .

### Model.getA()

**Synopsis**

```
getA()
```

**Description**

Get the coefficient matrix of model, returns a `scipy.sparse.csc_matrix` object. This method requires the `scipy` package.

**Example**

```
# Get the coefficient matrix
A = model.getA()
```

### Model.loadMatrix()

**Synopsis**

```
loadMatrix(c, A, lhs, rhs, lb, ub, vtype=None)
```

**Description**

Load matrix and vector data to build model. This method requires the `scipy` package.

**Arguments**

`c`

Objective costs. If `None`, the objective costs are all zeros.

`A`

Coefficient matrix. Must be of type `scipy.sparse.csc_matrix`.

`lhs`

Lower bounds of constraints.

`rhs`

Upper bounds of constraints.

`lb`

Lower bounds of variables. If `None`, the lower bounds are all zeros.

`ub`

Upper bounds of variables. If `None`, the upper bounds are all `COPT.INFINITY`.

`vtype`

Variable types. Default to `None`, which means all variables are continuous.

**Example**

```
# Build model by problem matrix
m.loadMatrix(c, A, lhs, rhs, lb, ub)
```

### Model.loadCone()

**Synopsis**

```
loadCone(ncone, types, dims, indices)
```

**Description**

Load Second-Order-Cones (SOC) to the model.

**Arguments**

`ncone`

Number of SOC.

`types`

Type of SOC, please refer to *SOC constraint types* for possible values.

`dims`

Dimension of SOC.

`indices`

Array of subscripts for the variables that constitute the SOC.

### Model.loadExpCone()

**Synopsis**

```
loadExpCone(ncone, types, indices)
```

**Description**

Load exponential cones to the model.

**Arguments**

`ncone`

Number of exponential cones.

`types`

Type of exponential cones, please refer to *Exponential Cone type* for possible values.

`indices`

Array of subscripts for the variables that constitute the exponential cones.

### Model.loadNlData()

**Synopsis**

```
loadNlData(nCols, nRows, sense, nGrad, idxGrad, nJac, idxJacRow,
idxJacCol, nHess, idxHessRow, idxHessCol, colLower, colUpper,
rowLower, rowUpper, initX, evalType, cb)
```

**Description**

Load the structural information and initial data of a nonlinear optimization model, and register a user-implemented nonlinear callback object.

This interface is used to provide the solver with the model size information, variable and constraint bounds, and the sparsity structure of derivatives of the objective and constraint functions, while registering callback interfaces for evaluation in nonlinear optimization.

During the nonlinear optimization process, the solver will invoke the callback object `cb` at different iteration points to dynamically evaluate the values of the objective function, constraint functions, and their first-order and second-order derivatives, to advance the nonlinear optimization algorithm.

This interface adopts a callback-based nonlinear modeling approach, decoupling model structure definition from numerical computation logic. Users provide the required model and derivative information through callback methods.

**Arguments**

`nCols`

Number of variables.

Possible values: positive integers.

`nRows`

Number of constraints.

Possible values: non-negative integers.

`sense`

Optimization sense.

Possible values: `COPT.MINIMIZE` or `COPT.MAXIMIZE`.

`nGrad`

Number of nonzero elements in the objective gradient, or an identifier specifying the gradient storage format.

Possible values: a non-negative integer, or a dense storage type such as `COPT.DENSETYPE_ROWMAJOR` or `COPT.DENSETYPE_COLMAJOR`.

`idxGrad`

Index sequence of variables corresponding to the nonzero elements in the objective gradient.

Possible values: a sequence of integers, or `None`.

`nJac`

Number of nonzero elements in the constraint Jacobian matrix, or an identifier specifying the Jacobian storage format.

Possible values: a non-negative integer, or a dense storage type such as `COPT.DENSETYPE_ROWMAJOR` or `COPT.DENSETYPE_COLMAJOR`.

`idxJacRow`

Row index sequence of nonzero elements in the Jacobian matrix.

Possible values: a sequence of integers, or `None`.

`idxJacCol`

Column index sequence of nonzero elements in the Jacobian matrix.

Possible values: a sequence of integers, or `None`.

`nHess`

Number of nonzero elements in the Hessian matrix, or an identifier specifying the Hessian storage format.

Possible values: a non-negative integer, or a dense storage type such as `COPT.DENSETYPE_ROWMAJOR` or `COPT.DENSETYPE_COLMAJOR`.

`idxHessRow`

Row index sequence of nonzero elements in the Hessian matrix.

Possible values: a sequence of integers, or `None`.

`idxHessCol`

Column index sequence of nonzero elements in the Hessian matrix.

Possible values: a sequence of integers, or `None`.

`colLower`

Lower bounds of variables.

Possible values: a sequence of `float` numbers, or `None`.

If `None`, default lower bounds 0 are applied to all variables.

`colUpper`

Upper bounds of variables.

Possible values: a sequence of `float` numbers, or `None`.

If `None`, default upper bounds `COPT.INFINITY` are applied to all variables.

`rowLower`

Lower bounds of constraint functions.

Possible values: a sequence of `float` numbers.

If `None`, default lower bounds `-COPT.INFINITY` are applied to all constraints.

`rowUpper`

Upper bounds of constraint functions.

Possible values: a sequence of `float` numbers.

If `None`, default upper bounds `COPT.INFINITY` are applied to all constraints.

`initX`

Initial values of variables.

Possible values: a sequence of `float` numbers, or `None`.

`evalType`

Declares which callback methods are implemented.

This argument declares which information of the nonlinear model (including the objective function, constraint functions, and their first- and second-order derivatives) are available through the callback interface. During the nonlinear optimization process, the solver invokes the corresponding callback methods as required by the algorithm.

Possible values: see *Nonlinear Model Callback Evaluation Types*. Multiple flags can be combined using bitwise OR.

`cb`

User-implemented nonlinear callback object.

Possible values: an object derived from *NlpCallbackBase Class*.

**Example**

```
# Customized NLP callback
nlpcb = MyNlpCallback()

model.loadNlData(
    nCols=2,          # number of variables
    nRows=1,          # number of constraints
    sense=COPT.MAXIMIZE,

    nGrad=2,
    idxGrad=[0, 1],   # objective gradient structure

    nJac=2,
    idxJacRow=[0, 0], # Jacobian structure: row indices
    idxJacCol=[0, 1], # Jacobian structure: column indices

    nHess=2,
    idxHessRow=[0, 1],  # Hessian (lower triangular)
    idxHessCol=[0, 1],

    colLower=[-1.0, -COPT.INFINITY],
    colUpper=[1.0,  COPT.INFINITY],
    rowLower=[0.0],
    rowUpper=[0.0],
    initX=[0.5, 1.5],

    evalType=-1,      # all evaluation callbacks are provided
    cb=nlpcb,
)
```

**Notes**

1. Setting `evalType`

   If the nonlinear callback object implements all supported numerical computation methods, the `evalType` parameter may be set to a negative value (for example, `-1`).

   In this case, it is not necessary to explicitly combine individual `EVALTYPE_*` constants. During the nonlinear optimization process, the solver will automatically invoke the corresponding callback methods as required by the algorithm.

2. Rules for derivative structure arguments (`nGrad`, `nJac`, and `nHess`)

   The parameters `nGrad`, `nJac`, and `nHess` are used to describe the structural information of the objective gradient, the Jacobian matrix, and the Hessian matrix, respectively. These parameters follow the same specification rules:

   - If the value is a non-negative integer, it represents the number of nonzero elements in the corresponding derivative matrix. In this case, the locations of the nonzero elements must be specified through the associated index arrays (such as `idxGrad`, `idxJacRow` / `idxJacCol`, and `idxHessRow` / `idxHessCol`).
   - If the value is `COPT.DENSETYPE_ROWMAJOR` or `COPT.DENSETYPE_COLMAJOR`, the corresponding derivative matrix is treated as dense and stored in row-major or column-major order, respectively. In this case, the associated index arrays may be set to `None`.

   The chosen sparse or dense structure must be consistent with the ordering of numerical values returned by the callback methods.

### Model.getLpSolution()

**Synopsis**

```
getLpSolution()
```

**Description**

Retrieve the values of variables, slack variables, dual variables and reduced cost of variables. Return a quad tuple object, in which each element is a list.

**Example**

```
# Retrieve solutions of linear model.
values, slacks, duals, redcosts = m.getLpSolution()
```

### Model.setLpSolution()

**Synopsis**

```
setLpSolution(values, slack, duals, redcost)
```

**Description**

Set LP solution.

**Arguments**

`values`

Solution of variables.

`slack`

Solution of slack variables.

`duals`

Solution of dual variables.

`redcost`

Reduced costs of variables.

**Example**

```
# Set LP solution
m.setLpSolution(values, slack, duals, redcost)
```

### Model.getValues()

**Synopsis**

```
getValues()
```

**Description**

Retrieve solution values of all variables in a LP or MIP. Return a Python list.

**Example**

```
values = m.getValues()
```

### Model.getRedcosts()

**Synopsis**

```
getRedcosts()
```

**Description**

Retrieve reduced costs of all variables in a LP. Return a list.

**Example**

```
# Retrieve reduced cost of all variables in model.
redcosts = m.getRedcosts()
```

### Model.getSlacks()

**Synopsis**

```
getSlacks()
```

**Description**

Retrieve values of all slack variables in a LP. Return a Python list.

**Example**

```
# Retrieve value of all slack variables in model.
 slacks = m.getSlacks()
```

### Model.getDuals()

**Synopsis**

```
getDuals()
```

**Description**

Obtain values of all dual variables in a LP. Return a Python list.

**Example**

```
# Retrieve value of all dual variables in model.
duals = m.getDuals()
```

### Model.getVarBasis()

**Synopsis**

```
getVarBasis(vars=None)
```

**Description**

Obtain basis status of specified variables.

If parameter `vars` is `None`, then return a list object consisting of all variables' basis status. If parameter `vars` is *Var Class* object, then return basis status of the specified variable. If parameter `vars` is list or *VarArray Class* object, then return a list object consisting of the specified variables' basis status. If parameter `vars` is dictionary or *tupledict Class* object, then return index of the specified variable as key and *tupledict Class* object consisting of the specified variables' basis status as value.

**Arguments**

`vars`

The specified variables. Optional, `None` by default,

**Example**

```
# Retrieve all variables' basis status in model.
varbasis = m.getVarBasis()
# Retrieve basis status of variable x and y.
varbasis = m.getVarBasis([x, y])
# Retrieve basis status of tupledict object xx.
varbasis = m.getVarBasis(xx)
```

### Model.getConstrBasis()

**Synopsis**

`getConstrBasis(constrs=None)`

**Description**

Obtain the basis status of linear constraints in LP.

If parameter `constrs` is `None`, then return a list object consisting of all linear constraints' basis status. If parameter `constrs` is *Constraint Class* object, then return basis status of the specified linear constraint. If parameter `constrs` is list or *ConstrArray Class* object, then return a list object consisting of the specified linear constraints' basis status. If parameter `constrs` is dictionary or *tupledict Class* object, then return the index of the specified linear constraint as key and return *tupledict Class* object consisting of the specified linear constraints' basis status as value.

**Arguments**

`constrs`

The specified linear constraint. Optional, `None` by default.

**Example**

```
# Retrieve all linear constraints' basis status in model.
conbasis = m.getConstrBasis()
# Retrieve basis status corresponding to linear constraint r0 and r1 in model.
conbasis = m.getConstrBasis([r0, r1])
# Retrieve basis status of linear constraints in tupledict rr.
conbasis = m.getConstrBasis(rr)
```

### Model.getBasisHead()

**Synopsis**

`getBasisHead()`

**Description**

Obtain the indices of the variables that make up the current basis.

**Return value**

A *NdArray Class* object consisting of the basis head indices.

**Example**

```
# Retrieve indices of the variables that make up the current basis.
basishead = m.getBasisHead()
```

### Model.fsolve()

**Synopsis**

```
fsolve(vals, idxes)
```

**Description**

Given a sparse input vector, perform a forward solve and return the sparse result vector.

**Arguments**

`vals`

Values of the nonzero elements in the sparse input vector. An iterable object of floats.

`idxes`

Indices of the nonzero elements in the sparse input vector. An iterable object of integers, whose length should be the same as `vals`.

**Return value**

The sparse result vector, returned as a dictionary whose keys are element indices and whose values are the corresponding element values.

**Example**

```
# Perform a forward solve on the sparse input vector.
fsol = m.fsolve([1.0, 2.0], [0, 2])
```

### Model.bsolve()

**Synopsis**

```
bsolve(vals, idxes)
```

**Description**

Given a sparse input vector, perform a backward solve and return the sparse result vector.

**Arguments**

`vals`

Values of the nonzero elements in the sparse input vector. An iterable object of floats.

`idxes`

Indices of the nonzero elements in the sparse input vector. An iterable object of integers, whose length should be the same as `vals`.

**Return value**

The sparse result vector, returned as a dictionary whose keys are element indices and whose values are the corresponding element values.

**Example**

```
# Perform a backward solve on the sparse input vector.
bsol = m.bsolve([1.0, 2.0], [0, 2])
```

### Model.computeShadowPriceRange()

**Synopsis**

```
computeShadowPriceRange(rowidx, ifRhs)
```

**Description**

Compute the shadow price range of the given linear constraint.

**Arguments**

`rowidx`

Index of the linear constraint, or a *Constraint Class* object.

`ifRhs`

For a range constraint, whether to use the RHS to compute the shadow price.

**Return value**

A two-element tuple consisting of the lower and upper bound of the shadow price, that is `(lower, upper)`.

**Example**

```
# Compute the shadow price range of linear constraint r0.
lower, upper = m.computeShadowPriceRange(r0, 1)
```

### Model.getPoolObjVal()

**Synopsis**

```
getPoolObjVal(isol)
```

**Description**

Obtain the `isol` -th objective value in solution pool, return a constant.

**Arguments**

`isol`

Index of solution.

**Example**

```
# Obtain the second objective value
objval = m.getPoolObjVal(2)
```

### Model.getPoolSolution()

**Synopsis**

```
getPoolSolution(isol, vars)
```

**Description**

Obtain variable values in the `isol` -th solution of solution pool.

If parameter `vars` is *Var Class* object, then return values of the specified variable. If parameter `vars` is list or *VarArray Class* object, then return a list object consisting of the specified variables' values. If parameter `vars` is dictionary or *tupledict Class* object, then return index of the specified variable as key and *tupledict Class* object consisting of the specified variables' values as value.

**Arguments**

`isol`

Index of solution

`vars`

The specified variables.

**Example**

```
# Get value of x in the second solution
xval = m.getPoolSolution(2, x)
```

### Model.getPoolObjValN()

**Synopsis**

```
getPoolObjValN(idx, isol)
```

**Description**

In a multi-objective model, retrieve the objective value of the specified objective at the given solution index in the solution pool.

**Arguments**

`idx`

Index of the target objective in the multi-objective model.

`isol`

Index of the solution in the solution pool.

**Return value**

Returns a double value.

### Model.getVarLowerIIS()

**Synopsis**

```
getVarLowerIIS(vars)
```

**Description**

Obtain IIS status of lower bounds of variables.

If parameter `vars` is *Var Class* object, then return IIS status of lower bound of variable. If parameter `vars` is list or *VarArray Class* object, then return a list object consisting of the IIS status of lower bounds of variables. If parameter `vars` is dictionary or *tupledict Class* object, then return index of the specified variable as key and *tupledict Class* object consisting of the IIS status of lower bounds of variables as value.

**Arguments**

`vars`

The specified variables.

**Example**

```
# Retrieve IIS status of lower bounds of variable x and y.
lowerIIS = m.getVarLowerIIS([x, y])
# Retrieve IIS status of lower bounds of variables in tupledict object xx.
lowerIIS = m.getVarLowerIIS(xx)
```

### Model.getVarUpperIIS()

**Synopsis**

```
getVarUpperIIS(vars)
```

**Description**

Obtain IIS status of upper bounds of variables.

If parameter `vars` is *Var Class* object, then return IIS status of upper bound of variable. If parameter `vars` is list or *VarArray Class* object, then return a list object consisting of the IIS status of upper bounds of variables. If parameter `vars` is dictionary or *tupledict Class* object, then return index of the specified variable as key and *tupledict Class* object consisting of the IIS status of upper bounds of variables as value.

**Arguments**

`vars`

The specified variables.

**Example**

```
# Retrieve IIS status of upper bounds of variable x and y.
upperIIS = m.getVarUpperIIS([x, y])
# Retrieve IIS status of upper bounds of variables in tupledict object xx.
upperIIS = m.getVarUpperIIS(xx)
```

### Model.getConstrLowerIIS()

**Synopsis**

```
getConstrLowerIIS(constrs)
```

**Description**

Obtain the IIS status of lower bounds of constraints.

If parameter `constrs` is *Constraint Class* object, then return IIS status of lower bound of constraint. If parameter `constrs` is list or *ConstrArray Class* object, then return a list object consisting of the IIS status of lower bounds of constraints. If parameter `constrs` is dictionary or *tupledict Class* object, then return the index of the specified linear constraint as key and return *tupledict Class* object consisting of the IIS status of lower bounds of constraints.

**Arguments**

`constrs`

The specified linear constraint.

**Example**

```
# Retrieve IIS status corresponding to lower bounds of linear constraint r0 and r1 in␣
↪model.
lowerIIS = m.getConstrLowerIIS([r0, r1])
# Retrieve IIS status of lower bounds of linear constraints in tupledict rr.
lowerIIS = m.getConstrLowerIIS(rr)
```

### Model.getConstrUpperIIS()

**Synopsis**

```
getConstrUpperIIS(constrs)
```

**Description**

Obtain the IIS status of upper bounds of constraints.

If parameter `constrs` is *Constraint Class* object, then return IIS status of upper bound of constraint. If parameter `constrs` is list or *ConstrArray Class* object, then return a list object consisting of the IIS status of upper bounds of constraints. If parameter `constrs` is dictionary or *tupledict Class* object, then return the index of the specified linear constraint as key and return *tupledict Class* object consisting of the IIS status of upper bounds of constraints.

**Arguments**

`constrs`

The specified linear constraint.

**Example**

```
# Retrieve IIS status corresponding to upper bounds of linear constraint r0 and r1 in␣
↪model.
upperIIS = m.getConstrUpperIIS([r0, r1])
# Retrieve IIS status of upper bounds of linear constraints in tupledict rr.
upperIIS = m.getConstrUpperIIS(rr)
```

### Model.getSOSIIS()

**Synopsis**

```
getSOSIIS(soss)
```

**Description**

Obtain the IIS status of SOS constraints.

If parameter `soss` is *SOS Class* object, then return IIS status of SOS constraint. If parameter `soss` is list or *SOSArray Class* object, then return a list object consisting of the IIS status of SOS constraints. If parameter `soss` is dictionary or *tupledict Class* object, then return the index of the specified SOS constraint as key and return *tupledict Class* object consisting of the IIS status of SOS constraints.

**Arguments**

`soss`

The specified SOS constraint.

**Example**

```
# Retrieve IIS status corresponding to SOS constraint r0 and r1 in model.
sosIIS = m.getSOSIIS([r0, r1])
# Retrieve IIS status of SOS constraints in tupledict rr.
sosIIS = m.getSOSIIS(rr)
```

### Model.getIndicatorIIS()

**Synopsis**

```
getIndicatorIIS(genconstrs)
```

**Description**

Obtain the IIS status of indicator constraints.

If parameter `genconstrs` is *GenConstr Class* object, then return IIS status of indicator constraint. If parameter `genconstrs` is list or *GenConstrArray Class* object, then return a list object consisting of the IIS status of indicator constraints. If parameter `genconstrs` is dictionary or *tupledict Class* object, then return the index of the specified indicator constraint as key and return *tupledict Class* object consisting of the IIS status of indicator constraints.

**Arguments**

`genconstrs`

The specified indicator constraint.

**Example**

```
# Retrieve IIS status corresponding to indicator constraint r0 and r1 in model.
indicatorIIS = m.getIndicatorIIS([r0, r1])
# Retrieve IIS status of indicator constraints in tupledict rr.
indicatorIIS = m.getIndicatorIIS(rr)
```

### Model.getAttr()

**Synopsis**

```
getAttr(attrname)
```

**Description**

Get the value of an attribute of model. Return a constant.

**Arguments**

`attrname`

The specified attribute name. The full list of available attributes can be found in *Attributes* section.

**Example**

```
# Retrieve the constant terms of objective.
objconst = m.getAttr(COPT.Attr.ObjConst)
```

### Model.getInfo()

**Synopsis**

```
getInfo(infoname, args)
```

**Description**

Retrieve specified information.

If parameter `args` is *Var Class* object, *Constraint Class* object, *QConstraint Class* object, or *NlConstraint Class* object, then return info of the specified variable or constraint.

If parameter `args` is list or *VarArray Class* object, *ConstrArray Class* object, iterable *QConstraint Class* objects, or *NlConstraint Class* objects, then return a list consisting of the specified variables or constraints.

If parameter `args` is dictionary or *tupledict Class* object, then return the index of the specified variables or constraints as key and return *tupledict Class* object consisting of info corresponding to the specified variables or constraints as value.

If parameter `args` is *MVar Class* object, *MConstr Class* object, *MQConstr Class* object or *MPsdConstr Class* object, then return a *NdArray Class* object consisting of the info of the specified variables or constraints. In legacy matrix modeling mode ( `Model.matrixmodelmode = "legacy"` ) a `numpy.ndarray` is returned instead, see *Matrix Modeling*.

**Arguments**

`infoname`

The specified information name. The full list of available attributes can be found in *Information* section.

`args`

Variables and constraints to get information.

**Example**

```
# Retrieve lower bound information of all linear constraints in model.
lb = m.getInfo(COPT.Info.LB, m.getConstrs())
# Retrieve value information of variables x and y.
```

(continues on next page)

(continued from previous page)

```
sol = m.getInfo(COPT.Info.Value, [x, y])
# Retrieve the dual variable value corresponding to linear constraint in tupledict␣
↪object shipconstr.
dual = m.getInfo(COPT.Info.Dual, shipconstr)
```

### Model.getAttrN()

**Synopsis**

```
getAttrN(idx, attrname)
```

**Description**

Get the attribute value of the specified objective in multi-objective optimization.

**Arguments**

`idx`

Index of the objective function.

`attrname`

Name of the objective attribute.

Possible values: *HasQObj*, *HasNLObj*, *LpObjVal*, *BestObj*, *ObjSense*, *ObjConst*.

**Return value**

A double or integer value.

### Model.getParam()

**Synopsis**

```
getParam(paramname)
```

**Description**

Retrieve the current value of the specified parameter. Return a constant.

**Arguments**

`paramname`

The name of the parameter to get access to. The full list of available attributes can be found in *Parameters* section.

**Example**

```
# Retrieve current value of time limit.
timelimit = m.getParam(COPT.Param.TimeLimit)
```

### Model.getParamInfo()

**Synopsis**

```
getParamInfo(paramname)
```

**Description**

Retrieve information of the specified optimization parameter. Return a tuple object, consisting of param name, current value, default value, minimum value, maximum value.

**Arguments**

`paramname`

Name of the specified parameter. The full list of available values can be found in *Parameters* section.

**Example**

```
# Retrieve information of time limit.
pname, pcur, pdef, pmin, pmax = m.getParamInfo(COPT.Param.TimeLimit)
```

### Model.getObjParamN()

**Synopsis**

```
getObjParamN(idx, paramname)
```

**Description**

Get the value of a parameter for the specified objective in multi-objective optimization.

**Arguments**

`idx`

Index of the objective function.

`paramname`

Name of the objective parameter.

Possible values are *MultiObjPriority*, *MultiObjWeight*, *MultiObjRelTol*, *MultiObjAbsTol*.

**Return value**

A double or integer value.

### Model.getParamN()

**Synopsis**

```
getParamN(idx, paramname)
```

**Description**

Get the value of a solver parameter for the model associated with the specified objective in multi-objective optimization.

**Arguments**

`idx`

Index of the objective function.

`paramname`

Name of the solver parameter.

See *Parameters* for possible values.

**Return value**

A double or integer value.

### Model.setBasis()

**Synopsis**

```
setBasis(varbasis, constrbasis)
```

**Description**

Set basis status for all variables and linear constraints in LP. The parameters `varbasis` and `constrbasis` are list objects whose number of elements is the total number of variables or linear constraints.

**Arguments**

`varbasis`

The basis status of variables.

`constrbasis`

The basis status of constraints.

**Example**

```
# Set basis status for all variables and linear constraints in the model.
m.setBasis(varbasis, constrbasis)
```

### Model.setSlackBasis()

**Synopsis**

```
setSlackBasis()
```

**Description**

Set LP basis to be slack.

**Example**

```
#  Set LP basis to be slack.
m.setSlackBasis()
```

### Model.setVarType()

**Synopsis**

```
setVarType(vars, vartypes)
```

**Description**

Set the type of specific variable.

If parameter `vars` is *Var Class* object, then parameter `vartypes` is *Variable types* constant;

If parameter `vars` is dictionary or *tupledict Class* object, then parameter `vartypes` can be *Variable types* constant, dictionary or *tupledict Class* object;

If parameter `vars` is list or *VarArray Class* object, then parameter `vartypes` can be *Variable types* constant or list object.

**Arguments**

`vars`

The specified variable.

`vartypes`

Type of the specified variable.

**Example**

```
# Set variable x as integer variable.
m.setVarType(x, COPT.INTEGER)
# Set variables x and y as binary variables.
m.setVarType([x, y], COPT.BINARY)
# Set the variables in tupledict object xdict as continuous variables.
m.setVarType(xdict, COPT.CONTINUOUS)
```

### Model.setNames()

**Synopsis**

`setNames(args, names)`

**Description**

Sets the name(s) of the specified variable(s) or constraint(s).

**Arguments**

`args`

The specified variable(s) or constraint(s), which could be: *Var Class* , *Constraint Class* , *QConstraint Class* , *PsdVar Class* , *PsdConstraint Class* , *LmiConstraint Class* , *GenConstr Class* object, *AffineCone Class* or the dictionary/list they constitute.

`names`

The name(s) of the specified variable(s) or constraint(s), which could be a single string or the list/dictionary corresponding to the `args` .

**Example**

```
# Set the name of x to "var"
m.setNames(x, "var")
#Set the name of constr1 to "c1" and the name of constr2 to "c2"
m.setNames([constr1, constr2], ["c1", "c2"])
```

### Model.setMipStart()

**Synopsis**

```
setMipStart(vars, startvals)
```

**Description**

Set initial value for specified variables, valid only for integer programming.

If parameter `vars` is *Var Class* object, then parameter `startvals` is constant; If parameter `vars` is dictionary or *tupledict Class* object, then parameter `startvals` can be constant, dictionary or *tupledict Class* object; If parameter `vars` is list or *VarArray Class* object, then parameter `startvals` can be constant or list object.

**Notice:** You may want to call this method several times to input the MIP start. Please call `loadMipStart()` once when the input is done.

**Arguments**

`vars`

The specified variable.

`startvals`

Initial value of the specified variable

**Example**

```
# Set initial value of x as 1.
m.setMipStart(x, 1)
# Set initial value of x, y as 2, 3.
m.setMipStart([x, y], [2, 3])
# Set initial value of all variables in tupledict xdict as 1.
m.setMipStart(xdict, 1)

# Load initial solution to model
m.loadMipStart()
```

### Model.loadMipStart()

**Synopsis**

```
loadMipStart()
```

**Description**

Load the currently specified initial values into model.

**Notice:** After calling this method, the previously initial values will be cleared, and users can continue to set initial values for specified variables.

### Model.setNlPrimalStart()

**Synopsis**

```
setNlPrimalStart(vars, startvals)
```

**Description**

Set the initial values of variables in a nonlinear model.

**Arguments**

`vars`

The variables to be assigned initial values to.

Acceptable values include: a single *Var Class* object, a *VarArray Class* or any iterable of variables, a dictionary mapping arbitrary keys to *Var Class* objects, or a *MVar Class* multidimensional variable.

`startvals`

The corresponding start values. Can be a double value, a list, array, or a mapping structure consistent with the shape or keys of the variables.

### Model.setInfo()

**Synopsis**

`setInfo(infoname, args, newvals)`

**Description**

Set new information value for specific variables or constraints.

If parameter `args` is *Var Class* object or *Constraint Class* object, then parameter `newvals` is constant; If parameter `args` is dictionary or *tupledict Class* object, then parameter `newvals` can be constant, dictionary or *tupledict Class* object; If parameter `args` is list, *VarArray Class* object or *ConstrArray Class* object, then parameter `newvals` can be constant or list; If parameter `args` is *MVar Class* object, *MConstr Class* object, *MQConstr Class* object or *MPsdConstr Class* object. then parameter `newvals` can be constant or `numpy.ndarray`.

**Arguments**

`infoname`

The specified information name. The full list of available names can be found in *Information* section.

`args`

The specified variables of constraints.

`newvals`

Value of the specified information.

**Example**

```
m.setInfo(COPT.Info.LB, [x, y], [1.0, 2.0])

# Set the upperbound of variable x as 1.0
m.setInfo(COPT.Info.UB, x, 1.0)
# Set the lowerbound of variables x and y as 1.0, 2.0, respectively.
m.setInfo(COPT.Info.LB, [x, y], [1.0, 2.0])
# Set the objective of all variables in tupledict xdict as 0.
m.setInfo(COPT.Info.OBJ, xdict, 0.0)
```

### Model.setParam()

**Synopsis**

```
setParam(paramname, newval)
```

**Description**

Set the value of a parameter to a specific value.

**Arguments**

`paramname`

The name of parameter to be set. The list of available names can be found in *Parameters* section.

`newval`

New value of parameter.

**Example**

```
# Set time limit of solving to 1 hour.
 m.setParam(COPT.Param.TimeLimit, 3600)
```

### Model.resetParam()

**Synopsis**

```
resetParam()
```

**Description**

Reset all parameters to their default values.

**Example**

```
# Reset all parameters to their default values.
m.resetParam()
```

### Model.setObjParamN()

**Synopsis**

```
setObjParamN(idx, paramname, newval)
```

**Description**

Set a parameter of the specified objective in multi-objective optimization.

**Arguments**

`idx`

Index of the objective function.

`paramname`

Name of the objective parameter.

Possible values are *MultiObjPriority*, *MultiObjWeight*, *MultiObjRelTol*, *MultiObjAbsTol*.

`newval`

New value for the specified objective function parameter.

### Model.setParamN()

**Synopsis**

```
setParamN(idx, paramname, newval)
```

**Description**

Set a solver parameter for the model associated with the specified objective in multi-objective optimization.

**Arguments**

`idx`

Index of the objective function.

`paramname`

Name of the solver parameter.

See *Parameters* for possible values.

`newval`

New value of the specified solver parameter.

### Model.resetObjParamN()

**Synopsis**

```
resetObjParamN(idx)
```

**Description**

Reset all parameters of the specified objective in multi-objective optimization to their default values.

**Arguments**

`idx`

Index of the objective function.

### Model.resetParamN()

**Synopsis**

```
resetParamN(idx)
```

**Description**

Reset all solver parameters associated with the specified objective in multi-objective optimization to their default values.

**Arguments**

`idx`

Index of the objective function.

### Model.read()

**Synopsis**

`read(filename)`

**Description**

Determine the type of data by the file suffix and read it into a model.

Currently, it supports MPS files (suffix `'.mps'` or `'.mps.gz'`), LP files (suffix `'.lp'` or `'.lp.gz'`), SDPA files (suffix `'.dat-s'` or `'.dat-s.gz'`), CBF files (suffix `'.cbf'` or `'.cbf.gz'`), COPT binary format files (suffix `'.bin'`), basis files (suffix `'.bas'`), result files (suffix `'.sol'`), start files (suffix `'.mst'`), branching order files (suffix `'.ord'`), and parameter files (suffix `'.par'`).

**Arguments**

`filename`

Name of the file to be read.

**Example**

```
# Read MPS format model file
m.read('test.mps.gz')
# Read LP format model file
m.read('test.lp.gz')
# Read COPT binary format model file
m.read('test.bin')
# Read basis file
m.read('testlp.bas')
# Read solution file
m.read('testmip.sol')
# Read start file
m.read('testmip.mst')
# Read parameter file
m.read('test.par')
```

### Model.readMps()

**Synopsis**

`readMps(filename)`

**Description**

Read MPS file to model.

**Arguments**

`filename`

The name of the MPS file to be read.

**Example**

```
# Read file "test.mps.gz" according to mps file format
m.readMps('test.mps.gz')
# Read file "test.lp.gz" according mps file format
m.readMps('test.lp.gz')
```

### Model.readLp()

**Synopsis**

```
readLp(filename)
```

**Description**

Read a file to model according to LP file format.

**Arguments**

`filename`

Name of the LP file to be read.

**Example**

```
# Read file"test.mps.gz" according to LP file format
m.readLp('test.mps.gz')
# Read file"test.lp.gz" according to LP file format
m.readLp('test.lp.gz')
```

### Model.readSdpa()

**Synopsis**

```
readSdpa(filename)
```

**Description**

Read a file to model according to SDPA file format.

**Arguments**

`filename`

Name of the SDPA file to be read.

**Example**

```
# Read file"test.dat-s" according to SDPA file format
m.readSdpa('test.dat-s')
```

### Model.readCbf()

**Synopsis**

```
readCbf(filename)
```

**Description**

Read a file to model according to CBF file format.

**Arguments**

`filename`

Name of the CBF file to be read.

**Example**

```
# Read file"test.cbf" according to CBF file format
m.readCbf('test.cbf')
```

### Model.readBin()

**Synopsis**

`readBin(filename)`

**Description**

Read COPT binary format file to model.

**Arguments**

`filename`

The name of the COPT binary format file to be read.

**Example**

```
# Read file "test.bin" according COPT binary file format
m.readBin('test.bin')
```

### Model.readSol()

**Synopsis**

`readSol(filename)`

**Description**

Read a file to model according to solution file format.

**Notice:** The default solution value is 0, i.e. a partial solution will be automatically filled in with zeros. If read successfully, then the values read can be act as initial solution for integer programming.

**Arguments**

`filename`

Name of file to be read.

**Example**

```
# Read file "testmip.sol" according to solution file format.
m.readSol('testmip.sol')
# Read file testmip.txt" according to solution file format.
m.readSol('testmip.txt')
```

### Model.readJsonSol()

**Synopsis**

`readJsonSol(filename)`

**Description**

Read the complete solution in JSON format from a file.

**Arguments**

`filename`

Name of file to be read.

### Model.readBasis()

**Synopsis**

```
readBasis(filename)
```

**Description**

Read basis status of variables and linear constraints to model according to basis solution format, valid only for linear programming.

**Arguments**

`filename`

The name of the basis file to be read.

**Example**

```
# Read file "testmip.bas" to basis solution format
m.readBasis('testmip.bas')
# Read file "testmip.txt" to basis solution format
m.readBasis('testmip.txt')
```

### Model.readMst()

**Synopsis**

```
readMst(filename)
```

**Description**

Read initial solution data to model according to initial solution file format.

**Notice:** If read successfully, the read value will be act as initial solution for integer programming model. Variable values may not be specified completely, if the value of variable is specified for multiple times, the last specified value is used.

**Arguments**

`filename`

Name of the file to be read.

**Example**

```
# Read file "testmip.mst" according to initial solution file format.
m.readMst('testmip.mst')
# Read file "testmip.txt" according to initial solution file format.
m.readMst('testmip.txt')
```

### Model.readParam()

**Synopsis**

```
readParam(filename)
```

**Description**

Read optimization parameters to model according to parameter file format.

**Notice:** If any optimization parameter is specified multiple times, the last specified value is used.

**Arguments**

`filename`

The name of the parameter file to be read.

**Example**

```
# Read file "testmip.par" according to parameter file format.
m.readParam('testmip.par')
# Read file "testmip.txt" according to parameter file format.
m.readParam('testmip.txt')
```

### Model.readTune()

**Synopsis**

```
readTune(filename)
```

**Description**

Read the tuning parameters and combine them into the model according to the tuning file format.

**Arguments**

`filename`

The name of the file to be read.

**Example**

```
# Read the file "testmip.tune" according to the tuning file format
m.readTune('testmip.tune')
# Read the file "testmip.txt" according to the tuning file format
m.readTune('testmip.txt')
```

### Model.readOrd()

**Synopsis**

```
readOrd(filename)
```

**Description**

Read a branching order file (ORD format) into the current model to reuse a predefined branching strategy.

**Arguments**

`filename`

Name of the branching order file to be read.

**Example**

```
# Read branching order information from "testmip.ord"
m.readOrd("testmip.ord")
```

### Model.write()

**Synopsis**

`write(filename)`

**Description**

Currently, COPT supports writing of MPS files (suffix `'.mps'` ), LP files (suffix `'.lp'`), CBF files (suffix `'.cbf'`), COPT binary format files (suffix `'.bin'`), basis files (suffix `'.bas'`), LP solution files (suffix `'.sol'`), initial solution files (suffix `'.mst'`), branching order files (suffix `'.ord'`), and parameter files (suffix `'.par'`).

When writing an MPS, LP or COPT binary format file for a linear programming problem, set *WriteDualProb* to 1 to write the dual model instead of the original one.

**Arguments**

`filename`

The file name to be written.

**Example**

```
# Write MPS file
m.write('test.mps')
# Write LP file
m.write('test.lp')
# Write COPT binary format file
m.write('test.bin')
# Write basis file
m.write('testlp.bas')
# Write solution file
m.write('testmip.sol')
# Write initial solution file
m.write('testmip.mst')
# Write parameter file
m.write('test.par')
```

### Model.writeMps()

**Synopsis**

`writeMps(filename)`

**Description**

Write current model into an MPS file.

**Arguments**

`filename`

The name of the MPS file to be written.

**Example**

```
# Write MPS model file "test.mps"
m.writeMps('test.mps')
```

### Model.writeMpsStr()

**Synopsis**

```
writeMpsStr()
```

**Description**

Write current model into a buffer as MPS format.

**Example**

```
# Write model to buffer 'buff' and print model
buff = m.writeMpsStr()
print(buff.getData())
```

### Model.writeLp()

**Synopsis**

```
writeLp(filename)
```

**Description**

Write current optimization model to a LP file.

**Arguments**

`filename`

The name of the LP file to be written.

**Example**

```
# Write LP model file "test.lp"
m.writeLp('test.lp')
```

### Model.writeCbf()

**Synopsis**

```
writeCbf(filename)
```

**Description**

Write current optimization model to a CBF file.

**Arguments**

`filename`

The name of the CBF file to be written.

**Example**

```
# Write CBF model file "test.cbf"
m.writeCbf('test.cbf')
```

### Model.writeBin()

**Synopsis**

```
writeBin(filename)
```

**Description**

Write current model into a COPT binary format file.

**Arguments**

`filename`

The name of the COPT binary format file to be written.

**Example**

```
# Write COPT binary format model file "test.bin"
m.writeBin('test.bin')
```

### Model.writeIIS()

**Synopsis**

```
writeIIS(filename)
```

**Description**

Write current irreducible inconsistent subsystem into an IIS file.

**Arguments**

`filename`

The name of the IIS file to be written.

**Example**

```
# Write IIS file "test.iis"
m.writeIIS('test.iis')
```

### Model.writeRelax()

**Synopsis**

```
writeRelax(filename)
```

**Description**

Write the feasibility relaxation model into a Relax file.

**Arguments**

`filename`

The name of the Relax file to be written.

**Example**

```
# Write Relax file "test.relax"
m.writeRelax('test.relax')
```

### Model.writeSol()

**Synopsis**

```
writeSol(filename)
```

**Description**

Output the model solution to a solution file.

**Arguments**

`filename`

The name of the solution file to be written.

**Example**

```
# Write solution file "test.sol"
m.writeSol('test.sol')
```

### Model.writeJsonSol()

**Synopsis**

```
writeJsonSol(filename)
```

**Description**

Write the solution of the model to a `".json"` format result file.

**Arguments**

`filename`

The name of the solution file to be written.

### Model.writePoolSol()

**Synopsis**

```
writePoolSol(isol, filename)
```

**Description**

Output selected pool solution to a solution file.

**Arguments**

`isol`

Index of pool solution.

`filename`

The name of the solution file to be written.

**Example**

```
# Write 1-th pool solution to solution file "poolsol_1.sol"
m.writePoolSol('poolsol_1.sol')
```

### Model.writeBasis()

**Synopsis**

```
writeBasis(filename)
```

**Description**

Write the LP basic solution to a basis file.

**Arguments**

`filename`

The name of the basis file to be written.

**Example**

```
# Write the basis file "testlp.bas"
m.writeBasis('testlp.bas')
```

### Model.writeMst()

**Synopsis**

```
writeMst(filename)
```

**Description**

For integer programming models, write the best integer solution currently to the initial solution file. If there are no integer solutions, then the first set of initial solution stored in model is output.

**Arguments**

`filename`

Name of the file to be written.

**Example**

```
# Output initial solution file "testmip.mst"
m.writeMst('testmip.mst')
```

### Model.writeParam()

**Synopsis**

```
writeParam(filename)
```

**Description**

Output modified parameters to a parameter file.

**Arguments**

`filename`

The name of the parameter file to be written.

**Example**

```
# Output parameter file "testmip.par"
m.writeParam('testmip.par')
```

### Model.writeTuneParam()

**Synopsis**

```
writeTuneParam(idx, filename)
```

**Description**

Output the parameter tuning result of the specified number to the parameter file.

**Arguments**

`idx`

Parameter tuning result number.

`filename`

The name of the parameter file to be output.

**Example**

```
# Output the parameter tuning result of the specified number to the parameter file
↪"testmip.par"
m.writeTuneParam(0, 'testmip.par')
```

### Model.writeOrd()

**Synopsis**

```
writeOrd(filename)
```

**Description**

Write the branching order information of the current model into an ORD-format file, so that the branching strategy can be saved and reused.

**Arguments**

`filename`

Name of the branching order file to be written.

**Example**

```
# Write branching order information of the current model into "testmip.ord"
m.writeOrd("testmip.ord")
```

### Model.setLogFile()

**Synopsis**

```
setLogFile(logfile)
```

**Description**

Set the optimizer log file.

**Arguments**

`logfile`

The log file.

**Example**

```
# Set the log file as "copt.log"
m.setLogFile('copt.log')
```

### Model.setLogCallback()

**Synopsis**

```
setLogCallback(logcb)
```

**Description**

Set the call back function of log. The model keeps a reference to the callback until it is replaced or the model is released.

**Arguments**

`logcb`

User log callback: an instance of a class inheriting from `LogCallback` that overrides `Display(msg)`. A plain Python function receiving the log message is also accepted.

**Example**

```
# Set the call back function of log as a python function 'logcbfun'.
m.setLogCallback(logcbfun)
```

### Model.solve()

**Synopsis**

```
solve()
```

**Description**

Solve an optimization problem.

**Example**

```
# Solve the model.
m.solve()
```

### Model.solveLP()

**Synopsis**

```
solveLP()
```

**Description**

Solve LP model. If the model is integer programming, then the model will be solved as LP.

**Example**

```
# Solve a model calling LP solver.
m.solveLP()
```

### Model.computeIIS()

**Synopsis**

```
computeIIS()
```

**Description**

Compute IIS for infeasible model.

**Example**

```
# Compute IIS for infeasible model.
m.computeIIS()
```

### Model.feasRelax()

**Synopsis**

```
feasRelax(vars, lbpen, ubpen, constrs, rhspen, uppen=None)
```

**Description**

Compute the feasibility relaxation of an infeasible model.

**Arguments**

`vars`

The variables to relax.

Possible values are *Var Class* objects, *VarArray Class* objects, arrays, lists, dictionaries, or *tupledict Class* objects.

If `vars` is a *Var Class* object, then `lbpen` / `ubpen` can be constant.

If `vars` is a *VarArray Class* object, an array or a list. then `lbpen` / `ubpen` can correspondingly be array, list or constant.

If `vars` is a *tupledict Class* object or a dictionary, then `lbpen` / `ubpen` can be dictionary or constant.

`lbpen`

The penalty factor for the lower bound of the variables.

If `lbpen` is `None`, no relaxation of the variables' lower bound is allowed.

If `lbpen` is `COPT.INFINITY`, the lower bound of the corresponding variable in `vars` is not relaxed.

`ubpen`

The penalty factor for the upper bound of the variables.

If `ubpen` is `None`, no relaxation of the variables' lower bound is allowed.

If `ubpen` is `COPT.INFINITY`, the upper bound of the corresponding variable in `vars` is not relaxed.

`constrs`

The constraints to relax.

Possible values are *Constraint Class* objects, *ConstrArray Class* objects, arrays, lists, or dictionaries.

If `constrs` is a *Constraint Class* object, then `rhspen` / `uppen` can be constant.

If `constrs` is a *ConstrArray Class* object, an array, or a list, then `rhspen` / `uppen` can correspondingly be array, list, or constant.

If `constrs` is a dictionary, then `rhspen` / `uppen` can be dictionary or constant.

`rhspen`

The penalty factor for the right-hand side of the constraints.

If `rhspen` is `None`, no relaxation of the constraint boundaries is applied.

If `rhspen` is `COPT.INFINITY`, the constraint's right-hand side is not relaxed.

`uppen`

The penalty factor for the upper bound of bilateral constraints.

If `uppen` is `None`, the penalty factor is specified by `rhspen` .

If `uppen` is `COPT.INFINITY`, the upper bound of the constraint is not relaxed.

**Example**

```
# Compute the feasibility relaxation for an infeasible model
x = model.addVars(3, ub=1)
y = model.addVars(3, ub=2)
c = model.addConstrs(x[i] + y[i] <= 1 for i in range(3))
model.feasRelax([x[0], x[1]], [0.5, 1], [COPT.INFINITY, 2], c, 1)
```

### Model.feasRelaxS()

**Synopsis**

```
feasRelaxS(vrelax, crelax)
```

**Description**

Compute the feasibility relaxation of an infeasible model in a simplified mode.

**Arguments**

`vrelax`

Whether to relax variables.

`crelax`

Whether to relax constraints.

**Example**

```
# Compute the feasibility relaxation of model m
m.feasRelaxS(True, True)
```

### Model.feasRelaxN()

#### Synopsis

```
feasRelaxN(vars, lbpen, ubpen, constrs, rhspen, uppen=None)
```

#### Description

Computes the feasibility relaxation of an infeasible model formulated by matrix modeling.

Either `vars` or `constrs` must be a matrix modeling object; that is, at least one of the following conditions must hold:

- `vars` is a *MVar Class* object.
- `constrs` is a *MConstr Class* object.

#### Arguments

`vars`

The variables to relax.

Possible values are *MVar Class* objects, *Var Class* objects, *VarArray Class* objects, arrays, lists, dictionaries, or *tupledict Class* objects.

If `vars` is a *MVar Class* object, then `lbpen` / `ubpen` can be a *NdArray Class* object, array, list, dictionary, or constant.

If `vars` is a *Var Class* object, then `lbpen` / `ubpen` can be constant.

If `vars` is a *VarArray Class* object, an array, or a list, then `lbpen` / `ubpen` can be array, list, or constant.

If `vars` is a *tupledict Class* object or dictionary, then `lbpen` / `ubpen` can be dictionary or constant.

`lbpen`

The penalty factor for the lower bound of the variables.

If `lbpen` is `None`, no relaxation of the variables' lower bound is allowed.

If `lbpen` is `COPT.INFINITY`, the lower bound of the corresponding variable in `vars` is not relaxed.

`ubpen`

The penalty factor for the upper bound of the variables.

If `ubpen` is `None`, no relaxation of the variables' lower bound is allowed.

If `ubpen` is `COPT.INFINITY`, the upper bound of the corresponding variable in `vars` is not relaxed.

`constrs`

The constraints to relax.

Possible values are *MConstr Class* objects, *Constraint Class* objects, *ConstrArray Class* objects, arrays, lists, or dictionaries.

If `constrs` is a *MConstr Class* object, then `rhspen` / `uppen` can be a *NdArray Class* object, array, list, dictionary, or constant.

If `constrs` is a *Constraint Class* object, then `rhspen` / `uppen` can be constant.

If `constrs` is a *ConstrArray Class* object, an array, or a list, then `rhspen` / `uppen` can correspondingly be array, list, or constant.

If `constrs` is a dictionary, then `rhspen` / `uppen` can be dictionary or constant.

`rhspen`

The penalty factor for the right-hand side of the constraints.

If `rhspen` is `None`, no relaxation of the constraint boundaries is applied.

If `rhspen` is `COPT.INFINITY`, the constraint's right-hand side is not relaxed.

`uppen`

The penalty factor for the upper bound of bilateral constraints.

If `uppen` is `None`, the penalty factor is specified by `rhspen`.

If `uppen` is `COPT.INFINITY`, the upper bound of the constraint is not relaxed.

**Example**

```
# Compute the feasibility relaxation for an infeasible model formulated using matrix␣
↪modeling
mx = model.addMVar(shape=3, nameprefix="x")
# Define constraints
A = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([10, 20])
mc = model.addConstr(A @ x <= b, name="constrs")
# Define relaxation penalty factors
# The lower bound of the 2nd element of mx is not relaxed
lbpen = [1.0, COPT.INFINITY, 0.5]
# The upper bound of the 3rd element of mx is not relaxed
ubpen = NdArray([0.8, 1.5, COPT.INFINITY])
# Relaxation penalty factor for mc
rhspen = 2.0
# Compute feasibility relaxation for mx and mc in matrix modeling
model.feasRelaxN(mx, lbpen, ubpen, mc, rhspen)
```

### Model.tune()

**Synopsis**

```
tune()
```

**Description**

Parameter tuning of the model.

**Example**

```
# tune the parameters of the model
m.tune()
```

### Model.loadTuneParam()

**Synopsis**

```
loadTuneParam(idx)
```

**Description**

Load the parameter tuning results of the specified number into the model.

**Example**

```
# Load the best parameter tuning results into the model
m.loadTuneParam(0)
```

### Model.interrupt()

**Synopsis**

```
interrupt()
```

**Description**

Interrupt solving process of current problem.

**Example**

```
# Interrupt the solving process.
m.interrupt()
```

### Model.remove()

**Synopsis**

```
remove(args)
```

**Description**

Remove variables or constraints from a model.

To remove variable, then parameter `args` can be *Var Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

To remove linear constraint, then parameter `args` can be *Constraint Class* object, *ConstrArray Class* object, list, dictionary or *tupledict Class* object.

To remove SOS constraint, then parameter `args` can be *SOS Class* object, *SOSArray Class* object, list, dictionary or *tupledict Class* object.

To remove Second-Order-Cone constraints, then parameter `args` can be *Cone Class* object, *ConeArray Class* object, list, dictionary or *tupledict Class* object.

To remove exponential cone constraints, then parameter `args` can be *ExpCone Class* object, *ExpConeArray Class* object, list, dictionary or *tupledict Class* object.

To remove affine cone constraints, then parameter `args` can be *AffineCone Class* object, *AffineConeArray Class* object, list, dictionary or *tupledict Class* object.

To remove quadratic constraints, then parameter `args` can be *QConstraint Class* object, *QConstrArray Class* object, list, dictionary or *tupledict Class* object.

To remove positive semi-definite constraints, then parameter `args` can be *PsdConstraint Class* object, *PsdConstrArray Class* object, list, dictionary or *tupledict Class* object.

To remove indicator constraint, then parameter `args` can be *GenConstr Class* object, *GenConstrArray Class* object, list, dictionary or *tupledict Class* object.

To remove LMI constraint, then parameter `args` can be *LmiConstraint Class* object, *LmiConstrArray Class* object, list, dictionary or *tupledict Class* object.

To remove matrix variables or matrix constraints, then parameter `args` can be *MVar Class* object, *MConstr Class* object, *MQConstr Class* object or *MPsdConstr Class* object.

**Arguments**

`args`

Variables or constraints to be removed.

**Example**

```
# Remove linear constraint conx
m.remove(conx)
# Remove variables x and y
m.remove([x, y])
```

### Model.reset()

**Synopsis**

`reset()`

**Description**

Reset the result information of the model.

**Example**

```
# Reset the result information in model.
m.reset()
```

### Model.resetAll()

**Synopsis**

`resetAll()`

**Description**

Reset the result and other additional information of the model, such as MIP start, IIS, etc. By executing this function, the information that needs to be calculated of the model will be all cleared, only the original model itself will be kept.

**Example**

```
# Reset the result and other additional information of the model
m.resetAll()
```

### Model.clear()

**Synopsis**

```
clear()
```

**Description**

Clear all content of the whole model. By executing this function, all content of the whole model will be cleared, including the added variables, objective, and constraints.

**Example**

```
# Clear all content of the model.
m.clear()
```

### Model.dispose()

**Synopsis**

```
dispose()
```

**Description**

Release the model on the calling thread. May be called more than once, and the model must not be used afterwards. `Model` also supports the context manager protocol, so a `with` block releases the model on exit. Explicit release matters most for a model that sets a callback.

See *Best practice: Release environments and models* for details.

**Example**

```
# Release the model
m.dispose()
```

### Model.clone()

**Synopsis**

```
clone()
```

**Description**

Create a deep copy of an existing model. Return a *Model Class* object.

**Example**

```
# Create a deep copy of model
mcopy = m.clone()
```

#### Model.update()

**Synopsis**

`update()`

**Description**

Update the model, including numerical ranges, removed variables and constraints.

**Example**

```
# Update the model
m.update()
```

#### Model.setCallback()

**Synopsis**

`setCallback(cb, cbctx)`

**Description**

Set the callback function of the model.

**Arguments**

`cb`

Callback Class object, derived from *CallbackBase Class* .

`cbctx`

Callback context. Please refer to *Callback context* .

**Example**

```
cb = CoptCallback()
model.setCallback(cb, COPT.CBCONTEXT_MIPSOL)
```

### 24.2.4 Var Class

For easy access to information of variables, Var object provides methods such as `Var.LB`. The full list of information can be found in the *Information* section. For convenience, information can be accessed by names in original case or lowercase.

In addition, you can also access the value of the variable through `Var.x`, the variable type through `Var.vtype`, the name of the variable through `Var.name`, the Reduced cost value of the variable in LP through `Var.rc`, the basis status through `Var.basis`, and the index of the variable in the coefficient matrix through `Var.index` the Reduced cost value of the variable in LP through `Var.rc`, the basis status through `Var.basis`, and the index of the variable in the coefficient matrix through `Var.index`.

For the model-related information of the variables, as well as the variable type and name, the user can set the corresponding information value in the form of `"Var.LB = 0.0"`.

Var object contains related operations of COPT variables and provides the following methods:

### Var.getType()

**Synopsis**

```
getType()
```

**Description**

Retrieve the type of variable.

**Example**

```
# Retrieve the type of variable v
vtype = v.getType()
```

### Var.getName()

**Synopsis**

```
getName()
```

**Description**

Retrieve the name of variable.

**Example**

```
# Retrieve the name of variable v
varname = v.getName()
```

### Var.getBasis()

**Synopsis**

```
getBasis()
```

**Description**

Retrieve the basis status of variable.

**Example**

```
# Retrieve the basis status of variable v
varbasis = v.getBasis()
```

### Var.getLowerIIS()

**Synopsis**

```
getLowerIIS()
```

**Description**

Retrieve the IIS status of lower bound of variable.

**Example**

```
# Retrieve the IIS status of lower bound of variable v
lowerIIS = v.getLowerIIS()
```

### Var.getUpperIIS()

**Synopsis**

```
getUpperIIS()
```

**Description**

Retrieve the IIS status of upper bound of variable.

**Example**

```
# Retrieve the IIS status of upper bound of variable v
upperIIS = v.getUpperIIS()
```

### Var.getIdx()

**Synopsis**

```
getIdx()
```

**Description**

Retrieve the subscript of the variable in the coefficient matrix.

**Example**

```
#  Retrieve the subscript of variable v
vindex = v.getIdx()
```

### Var.setType()

**Synopsis**

```
setType(newtype)
```

**Description**

Set the type of variable.

**Arguments**

`newtype`

The type of variable to be set. Please refer to *Variable types* section for possible values.

**Example**

```
# Set the type of variable v
v.setType(COPT.BINARY)
```

### Var.setName()

**Synopsis**

```
setName(newname)
```

**Description**

Set the name of variable.

**Arguments**

`newname`

The name of variable to be set.

**Example**

```
# Set the name of variable v
v.setName(COPT.BINARY)
```

### Var.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Retrieve specified information. Return a constant.

**Arguments**

`infoname`

The name of the information. Please refer to *Information* for possible values.

**Example**

```
# Get lowerbound of variable x
x.getInfo(COPT.Info.LB)
```

### Var.setInfo()

**Synopsis**

```
setInfo(infoname, newval)
```

**Description**

Set new information value for a variable.

**Arguments**

`infoname`

The name of the information to be set. Please refer to *Information* section for possible values.

`newval`

New information value to set.

**Example**

```
# Set the lower bound of variable x
x.setInfo(COPT.Info.LB, 1.0)
```

#### Var.remove()

**Synopsis**

`remove()`

**Description**

Delete the variable from model.

**Example**

```
# Delete variable 'x'
x.remove()
```

### 24.2.5 VarArray Class

To facilitate users to operate on multiple *Var Class* objects, the Python interface of COPT provides VarArray object with the following methods:

#### VarArray()

**Synopsis**

`VarArray(vars=None)`

**Description**

Create a *VarArray Class* object.

If parameter `vars` is `None`, then create an empty *VarArray Class* object, otherwise initialize the new created *VarArray Class* object based on `vars`.

**Arguments**

`vars`

Variables to be added. Optional, `None` by default. `vars` can be *Var Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty VarArray object
vararr = VarArray()
# Create an empty VarArray object and initialize variables x, y.
vararr = VarArray([x, y])
```

#### VarArray.pushBack()

**Synopsis**

`pushBack(var)`

**Description**

Add single or multiple *Var Class* objects.

**Arguments**

`var`

Variables to be applied. `var` can be *Var Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add variable x to vararr
vararr.pushBack(x)
# Add variables x and y to vararr
vararr.pushBack([x, y])
```

### VarArray.getVar()

**Synopsis**

```
getVar(idx)
```

**Description**

Retrieve a variable from an index in a *VarArray Class* object. Return a *Var Class* object.

**Arguments**

`idx`

Subscript of the specified variable in *VarArray Class* object, starting with 0.

**Example**

```
# Get the variable with subscript of 1 in vararr
 vararr.getVar(1)
```

### VarArray.getAll()

**Synopsis**

```
getAll()
```

**Description**

Retrieve all variables in *VarArray Class* object. Returns a list object.

**Example**

```
# Get all variables in 'vararr'
varall = vararr.getAll()
```

### VarArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of variables in *VarArray Class* object.

**Example**

```
# Retrieve the number of variables in vararr.
arrsize = vararr.getSize()
```

### 24.2.6 PsdVar Class

PsdVar object contains related operations of COPT positive semi-definite variables and provides the following methods:

#### PsdVar.getName()

**Synopsis**

```
getName()
```

**Description**

Retrieve the name of positive semi-definite variable.

**Example**

```
# Retrieve the name of variable v
varname = v.getName()
```

#### PsdVar.getIdx()

**Synopsis**

```
getIdx()
```

**Description**

Retrieve the subscript of the variable in the model.

**Example**

```
# Retrieve the subscript of variable v
vindex = v.getIdx()
```

#### PsdVar.getDim()

**Synopsis**

```
getDim()
```

**Description**

Retrieve the dimension of positive semi-definite variable.

**Example**

```
# Retrieve the dimension of variable "v"
vdim = v.getDim()
```

#### PsdVar.getLen()

**Synopsis**

```
getLen()
```

**Description**

Retrieve the length of the expanded positive semi-definite variable.

**Example**

```
# Retrieve the length of the expanded positive semi-definite variable "v"
vlen = v.getLen()
```

### PsdVar.setName()

**Synopsis**

```
setName(newname)
```

**Description**

Set the name of positive semi-definite variable.

**Arguments**

`newname`

The name of positive semi-definite variable to be set.

**Example**

```
# Set the name of variable v
v.setName('v')
```

### PsdVar.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Retrieve specified information of positive semi-definite variable. Return a list.

**Arguments**

`infoname`

The name of the information. Please refer to *Information* for possible values.

**Example**

```
# Get solution values of positive semi-definite variable x
sol = x.getInfo(COPT.Info.Value)
```

### PsdVar.remove()

**Synopsis**

```
remove()
```

**Description**

Delete the positive semi-definite variable from model.

**Example**

```
# Delete variable 'x
x.remove()
```

### PsdVar.diag()

**Synopsis**

`diag(offset=0)`

**Description**

Retrieves the diagonal elements of the PSD variable.

**Arguments**

`offset`

The diagonal offset, default is 0.

If `offset` $> 0$, it represents a downward diagonal offset. If `offset` $< 0$, it represents an upward diagonal offset.

**Return value**

A *PsdExpr* object.

**Example**

```
# Retrieve the main diagonal elements of the PSD variable
v.diag(offset=0)
```

### PsdVar.pick()

**Synopsis**

`pick(indexes)`

**Description**

Retrieves a PSD expression composed of the elements at the specified indices of the PSD variable.

**Arguments**

`indexes`

The array of specified indices.

**Return value**

A *PsdExpr* object.

**Example**

```
# Retrieve a PSD expression composed of the element at index 0
barX = model.addPsdVars(3, "BAR_X")
barX.pick([0])
```

### PsdVar.sum()

**Synopsis**

`sum()`

**Description**

Retrieves a PSD expression composed of the sum of all elements in the PSD variable.

**Return value**

A *PsdExpr* object.

### PsdVar.toexpr()

**Synopsis**

```
toexpr()
```

**Description**

Retrieves the PSD expression equivalent to the PSD variable.

**Return value**

A *PsdExpr* object.

### PsdVar.shape

**Synopsis**

```
shape
```

**Description**

Shape of the `PsdVar` object.

**Return value**

Integer tuple.

### PsdVar.size

**Synopsis**

```
size
```

**Description**

Size of the `PsdVar` object.

**Return value**

Integer tuple.

### PsdVar.dim

**Synopsis**

```
dim
```

**Description**

Dimension of the `PsdVar` object.

**Return value**

Integer.

#### PsdVar.len

**Synopsis**

`len`

**Description**

Flattened length of the `PsdVar` object.

**Return value**

Integer.

### 24.2.7 PsdVarArray Class

To facilitate users to operate on multiple *PsdVar Class* objects, the Python interface of COPT provides PsdVarArray object with the following methods:

#### PsdVarArray()

**Synopsis**

`PsdVarArray(vars=None)`

**Description**

Create a *PsdVarArray Class* object.

If parameter `vars` is `None`, then create an empty *PsdVarArray Class* object, otherwise initialize the new created *PsdVarArray Class* object based on `vars`.

**Arguments**

`vars`

Positive semi-definite variables to be added. Optional, `None` by default. `vars` can be *PsdVar Class* object, *PsdVarArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty PsdVarArray object
vararr = PsdVarArray()
# Create a PsdVarArray object containing positive semi-definite variables x, y.
vararr = PsdVarArray([x, y])
```

#### PsdVarArray.pushBack()

**Synopsis**

`pushBack(var)`

**Description**

Add single or multiple *PsdVar Class* objects.

**Arguments**

`var`

Positive semi-definite variables to be applied. `vars` can be *PsdVar Class* object, *PsdVarArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add variable x to vararr
vararr.pushBack(x)
# Add variables x and y to vararr
vararr.pushBack([x, y])
```

#### PsdVarArray.getPsdVar()

**Synopsis**

```
getPsdVar(idx)
```

**Description**

Retrieve a positive semi-definite variable from an index in a *PsdVarArray Class* object. Return a *PsdVar Class* object.

**Arguments**

`idx`

Subscript of the specified positive semi-definite variable in *PsdVarArray Class* object, starting with 0.

**Example**

```
# Get the positive semi-definite variable with subscript of 1 in vararr
var = vararr.getPsdVar(1)
```

#### PsdVarArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of positive semi-definite variables in *PsdVarArray Class* object.

**Example**

```
# Retrieve the number of variables in vararr.
arrsize = vararr.getSize()
```

### 24.2.8 SymMatrix Class

SymMatrix object contains related operations of COPT symmetric matrices and provides the following methods:

#### SymMatrix.getIdx()

**Synopsis**

`getIdx()`

**Description**

Retrieve the subscript of the symmetric matrix in the model.

**Example**

```
# Retrieve the subscript of symmetric matrix mat
matidx = mat.getIdx()
```

#### SymMatrix.getDim()

**Synopsis**

`getDim()`

**Description**

Retrieve the dimension of symmetric matrix.

**Example**

```
# Retrieve the dimension of symmetric matrix "mat".
matdim = mat.getDim()
```

### 24.2.9 SymMatrixArray Class

To facilitate users to operate on multiple *SymMatrix Class* objects, the Python interface of COPT provides SymMatrixArray object with the following methods:

#### SymMatrixArray()

**Synopsis**

`SymMatrixArray(mats=None)`

**Description**

Create a *SymMatrixArray Class* object.

If parameter `mats` is `None`, then create an empty *SymMatrixArray Class* object, otherwise initialize the new created *SymMatrixArray Class* object based on `mats`.

**Arguments**

`mats`

`mats` can be *SymMatrix Class* object, *SymMatrixArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty SymMatrixArray object
matarr = SymMatrixArray()
# Create a SymMatrixArray object containing matx, maty.
matarr = SymMatrixArray([matx, maty])
```

### SymMatrixArray.pushBack()

**Synopsis**

`pushBack(mat)`

**Description**

Add single or multiple *SymMatrix Class* objects.

**Arguments**

`mat`

Symmetric matrices to be applied. `mat` can be *SymMatrix Class* object, *SymMatrixArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add symmetric matrix matx to matarr
matarr.pushBack(matx)
# Add symmetric matrices matx and maty to matarr
matarr.pushBack([matx, maty])
```

### SymMatrixArray.getMatrix()

**Synopsis**

`getMatrix(idx)`

**Description**

Retrieve a symmetric matrix from an index in a *SymMatrixArray Class* object. Return a *SymMatrix Class* object.

**Arguments**

`idx`

Subscript of the specified symmetric matrix in *SymMatrixArray Class* object, starting with 0.

**Example**

```
# Get the symmetric matrix with subscript of 1 in matarr
mat = matarr.getMatrix(1)
```

### SymMatrixArray.getSize()

**Synopsis**

`getSize()`

**Description**

Retrieve the number of symmetric matrices in *SymMatrixArray Class* object.

**Example**

```
# Retrieve the number of symmetric matrices in matarr.
arrsize = matarr.getSize()
```

### 24.2.10 Constraint Class

For easy access to information of constraints, Constraint class provides methods such as `Constraint.LB`. The supported information can be found in *Information* section. For convenience, information can be queried by names in original case or lowercase.

In addition, you can access the name of the constraint through `Constraint.name`, the dual value of the constraint in LP through `Constraint.pi`, the basis status of the constraint through `Constraint.basis`, and the index in the coefficient matrix through `Constraint.index`.

For the model-related information and constraint name, the user can also set the corresponding information in the form of `"Constraint.lb = -100"`.

Constraint object contains related operations of COPT constraints and provides the following methods:

#### Constraint.getName()

**Synopsis**

`getName()`

**Description**

Retrieve the name of linear constraint.

**Example**

```
# Retrieve the name of linear constraint 'con'.
conname = con.getName()
```

#### Constraint.getBasis()

**Synopsis**

`getBasis()`

**Description**

Retrieve the basis status of linear constraint.

**Example**

```
# Retrieve the basis status of linear constraint 'con'.
conbasis = con.getBasis()
```

#### Constraint.computeShadowPriceRange()

**Synopsis**

`computeShadowPriceRange(ifRhs)`

**Description**

Compute the shadow price range of the current linear constraint.

**Arguments**

`ifRhs`

For a range constraint, whether to use the RHS to compute the shadow price.

**Return value**

A two-element tuple consisting of the lower and upper bound of the shadow price, that is `(lower, upper)`.

**Example**

```
# Compute the shadow price range of the current linear constraint con.
lower, upper = con.computeShadowPriceRange(1)
```

### Constraint.getLowerIIS()

**Synopsis**

```
getLowerIIS()
```

**Description**

Retrieve the IIS status of lower bound of linear constraint.

**Example**

```
# Retrieve the IIS status of lower bound of linear constraint 'con'.
lowerIIS = con.getLowerIIS()
```

### Constraint.getUpperIIS()

**Synopsis**

```
getUpperIIS()
```

**Description**

Retrieve the IIS status of upper bound of linear constraint.

**Example**

```
# Retrieve the IIS status of upper bound of linear constraint 'con'.
upperIIS = con.getUpperIIS()
```

### Constraint.getIdx()

**Synopsis**

```
getIdx()
```

**Description**

Retrieve the subscript of linear constraint in coefficient matrix.

**Example**

```
# Retrieve the subscript of linear constraint con.
conidx = con.getIdx()
```

### Constraint.setName()

**Synopsis**

```
setName(newname)
```

**Description**

Set the name of linear constraint.

**Arguments**

`newname`

The name of constraint to be set.

**Example**

```
# Set the name of linear constraint 'con'.
con.setName('con')
```

### Constraint.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Retrieve specified information. Return a constant.

**Arguments**

`infoname`

Name of the information to be obtained. Please refer to *Information* section for possible values.

**Example**

```
# Get the lower bound of linear constraint con
conlb = con.getInfo(COPT.Info.LB)
```

### Constraint.setInfo()

**Synopsis**

```
setInfo(infoname, newval)
```

**Description**

Set new information value to the specified constraint.

**Arguments**

`infoname`

The name of the information to be set. Please refer to *Information* section for possible values.

`newval`

New information value to be set.

**Example**

```
# Set the lower bound of linear constraint con
con.setInfo(COPT.Info.LB, 1.0)
```

#### Constraint.remove()

**Synopsis**

`remove()`

**Description**

Delete the linear constraint from model.

**Example**

```
# Delete the linear constraint 'conx'
conx.remove()
```

### 24.2.11 ConstrArray Class

To facilitate users to operate on multiple *Constraint Class* objects, the Python interface of COPT provides ConstrArray class with the following methods:

#### ConstrArray()

**Synopsis**

`ConstrArray(constrs=None)`

**Description**

Create a *ConstrArray Class* object.

If parameter `constrs` is `None`, the create an empty *ConstrArray Class* object, otherwise initialize the newly created *ConstrArray Class* object with parameter `constrs`

**Arguments**

`constrs`

Linear constraints to be added. `None` by default.

`constrs` can be *Constraint Class* object, *ConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty ConstrArray object
conarr = ConstrArray()
# Create an ConstrArray object initialized with linear constraint conx and cony
conarr = ConstrArray([conx, cony])
```

### ConstrArray.pushBack()

**Synopsis**

`pushBack(constr)`

**Description**

Add single or multiple *Constraint Class* objects.

**Arguments**

`constr`

Linear constraints to be applied. `constr` can be *Constraint Class* object, *ConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add linear constraint r to conarr
conarr.pushBack(r)
# Add linear constraint r0 and r1 to conarr
conarr.pushBack([r0, r1])
```

### ConstrArray.getConstr()

**Synopsis**

`getConstr(idx)`

**Description**

Retrieve the linear constraint according to its subscript in *ConstrArray Class* object. Return a *Constraint Class* object.

**Arguments**

`idx`

Subscript of the desired constraint in *ConstrArray Class* object, starting with 0.

**Example**

```
# Retrieve the linear constraint with subscript 1  in conarr
conarr.getConstr(1)
```

### ConstrArray.getAll()

**Synopsis**

`getAll()`

**Description**

Retrieve all linear constraints in *ConstrArray Class* object. Returns a list object.

**Example**

```
# Get all linear constraints in 'conarr'
cons = conarr.getAll()
```

#### ConstrArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Get the number of elements in *ConstrArray Class* object.

**Example**

```
# Get the number of linear constraints in conarr
arrsize = conarr.getSize()
```

### 24.2.12 ConstrBuilder Class

ConstrBuilder object contains operations related to temporary constraints when building constraints, and provides the following methods:

#### ConstrBuilder()

**Synopsis**

```
ConstrBuilder()
```

**Description**

Create an empty *ConstrBuilder Class* object

**Example**

```
# Create an empty linear constraint builder
constrbuilder = ConstrBuilder()
```

#### ConstrBuilder.setBuilder()

**Synopsis**

```
setBuilder(expr, sense, rhs)
```

**Description**

Set expression and constraint type for linear constraint builder.

**Arguments**

`expr`

The expression to be set, which can be *Var Class* expression or *LinExpr Class* expression.

`sense`

Sense of constraint. The full list of available types can be found in *Constraint type* section.

`rhs`

Right hand side of constraint.

**Example**

```
# Set the expresson of linear constraint builder as: x+y==1
constrbuilder.setBuilder(x + y, COPT.EQUAL, 1)
```

#### ConstrBuilder.getExpr()

**Synopsis**

`getExpr()`

**Description**

Retrieve the expression of a linear constraint builder object.

**Example**

```
#  Retrieve the expression of a linear constraint builder
linexpr = constrbuilder.getExpr()
```

#### ConstrBuilder.getSense()

**Synopsis**

`getSense()`

**Description**

Retrieve the constraint sense of linear constraint builder object.

**Example**

```
# Retrieve the constraint sense of linear constraint builder object.
consense = constrbuilder.getSense()
```

### 24.2.13 ConstrBuilderArray Class

To facilitate users to operate on multiple *ConstrBuilder Class* objects, the Python interface of COPT provides ConstrArray object with the following methods:

#### ConstrBuilderArray()

**Synopsis**

`ConstrBuilderArray(constrbuilders=None)`

**Description**

Create a *ConstrBuilderArray Class* object.

If parameter `constrbuilders` is `None`, then create an empty *ConstrBuilderArray Class* object, otherwise initialize the newly created *ConstrBuilderArray Class* object by parameter `constrbuilders`.

**Arguments**

`constrbuilders`
: Linear constraint builder to be added. Optional, `None` by default. It can be *ConstrBuilder Class* object, *ConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty ConstrBuilderArray object.
conbuilderarr = ConstrBuilderArray()
# Create a ConstrBuilderArray object and initialize it with builders: conbuilderx and␣
↪conbuildery
conbuilderarr = ConstrBuilderArray([conbuilderx, conbuildery])
```

### ConstrBuilderArray.pushBack()

**Synopsis**

```
pushBack(constrbuilder)
```

**Description**

Add single or multiple *ConstrBuilder Class* objects.

**Arguments**

`constrbuilder`

Builder of linear constraint to be added, which can be *ConstrBuilder Class* object, *ConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add linear constraint builder conbuilderx to conbuilderarr
conbuilderarr.pushBack(conbuilderx)
# Add linear constraint builders conbuilderx and conbuildery to conbuilderarr
conbuilderarr.pushBack([conbuilderx, conbuildery])
```

### ConstrBuilderArray.getBuilder()

**Synopsis**

```
getBuilder(idx)
```

**Description**

Retrieve a temporary constraint from its index in *ConstrBuilderArray Class* object. Return a *ConstrBuilder Class* object.

Retrieve the corresponding builder object according to the subscript of linear constraint builder in *ConstrBuilderArray Class* object.

**Arguments**

`idx`

Subscript of the linear constraint builder in the *ConstrBuilderArray Class* object, starting with 0.

**Example**

```
# Retrieve the builder with subscript 1 in conbuilderarr
conbuilder = conbuilderarr.getBuilder(1)
```

### ConstrBuilderArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Get the number of elements in *ConstrBuilderArray Class* object.

**Example**

```
# Get the number of builders in conbuilderarr
arrsize = conbuilderarr.getSize()
```

### 24.2.14 QConstraint Class

For easy access to information of quadratic constraints, QConstraint class provides methods such as `QConstraint.index`. The supported information can be found in *Information* section. For convenience, information can be queried by names in original case or lowercase.

In addition, you can access the name of the quadratic constraint through `QConstraint.name`, and the index in the model through `QConstraint.index`.

For the model-related information and constraint name, the user can also set the corresponding information in the form of `"QConstraint.rhs = -100"`.

QConstraint object contains related operations of COPT quadratic constraints and provides the following methods:

#### QConstraint.getName()

**Synopsis**

`getName()`

**Description**

Retrieve the name of quadratic constraint.

**Example**

```
# Retrieve the name of quadratic constraint 'qcon'
qconname = qcon.getName()
```

#### QConstraint.getRhs()

**Synopsis**

`getRhs()`

**Description**

Retrieve the right hand side of quadratic constraint.

**Example**

```
# Retrieve the RHS of quadratic constraint 'qcon'
qconrhs = qcon.getRhs()
```

#### QConstraint.getSense()

**Synopsis**

`getSense()`

**Description**

Retrieve the type of quadratic constraint.

**Example**

```
# Retrieve the type of quadratic constraint 'qcon'
qconsense = qcon.getSense()
```

### QConstraint.getIdx()

**Synopsis**

`getIdx()`

**Description**

Retrieve the subscript of quadratic constraint.

**Example**

```
# Retrieve the subscript of quadratic constraint 'qcon'
qconidx = qcon.getIdx()
```

### QConstraint.setName()

**Synopsis**

`setName(newname)`

**Description**

Set the name of quadratic constraint.

**Arguments**

`newname`

The name of quadratic constraint to be set.

**Example**

```
# Set the name of quadratic constraint 'qcon'.
qcon.setName('qcon')
```

### QConstraint.setRhs()

**Synopsis**

`setRhs(rhs)`

**Description**

Set the right hand side of quadratic constraint.

**Arguments**

`rhs`

The right hand side of quadratic constraint to be set.

**Example**

```
# Set the RHS of quadratic constraint 'qcon' to 0.0
qcon.setRhs(0.0)
```

### QConstraint.setSense()

**Synopsis**

```
setSense(sense)
```

**Description**

Set the sense of quadratic constraint.

**Arguments**

`sense`

The sense of quadratic constraint to be set.

**Example**

```
# Set the sense of quadratic constraint 'qcon' to <=
qcon.setSense(COPT.LESS_EQUAL)
```

### QConstraint.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Retrieve specified information. Return a constant.

**Arguments**

`infoname`

Name of the information to be obtained.

Please refer to *Information* section for possible values.

**Example**

```
# Get the row activity of quadratic constraint 'qcon'
qconlb = qcon.getInfo(COPT.Info.Slack)
```

### QConstraint.setInfo()

**Synopsis**

```
setInfo(infoname, newval)
```

**Description**

Set new information value to the specified quadratic constraint.

**Arguments**

`infoname`

The name of the information to be set. Please refer to *Information* section for possible values.

`newval`

New information value to be set.

**Example**

```
# Set the lower bound of quadratic constraint 'qcon'
qcon.setInfo(COPT.Info.LB, 1.0)
```

#### Constraint.remove()

**Synopsis**

```
remove()
```

**Description**

Delete the quadratic constraint from model.

**Example**

```
# Delete the quadratic constraint 'qconx'
qconx.remove()
```

### 24.2.15 QConstrArray Class

To facilitate users to operate on multiple *QConstraint Class* objects, the Python interface of COPT provides QConstrArray class with the following methods:

#### QConstrArray()

**Synopsis**

```
QConstrArray(qconstrs=None)
```

**Description**

Create a *QConstrArray Class* object.

If parameter `qconstrs` is `None`, the create an empty *QConstrArray Class* object, otherwise initialize the newly created *QConstrArray Class* object with parameter `qconstrs`

**Arguments**

`qconstrs`

Quadratic constraints to be added. `None` by default.

`qconstrs` can be *QConstraint Class* object, *QConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty QConstrArray object
qconarr = QConstrArray()
# Create an QConstrArray object initialized with quadratic constraint qconx and qcony
qconarr = QConstrArray([qconx, qcony])
```

### QConstrArray.pushBack()

**Synopsis**

`pushBack(qconstrs)`

**Description**

Add single or multiple *QConstraint Class* object.

**Arguments**

`qconstrs`

Quadratic constraints to be added. `None` by default.

`qconstrs` can be *QConstraint Class* object, *QConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add quadratic constraint qr to conarr
qconarr.pushBack(qr)
# Add quadratic constraint qr0 and qr1 to qconarr
qconarr.pushBack([qr0, qr1])
```

### QConstrArray.getQConstr()

**Synopsis**

`getQConstr(idx)`

**Description**

Retrieve the quadratic constraint according to its subscript in *QConstrArray Class* object. Return a *QConstraint Class* object.

**Arguments**

`idx`

Subscript of the desired quadratic constraint in *QConstrArray Class* object, starting with 0.

**Example**

```
# Retrieve the quadratic constraint with subscript 1  in qconarr
qcon = qconarr.getQConstr(1)
```

### QConstrArray.getSize()

**Synopsis**

`getSize()`

**Description**

Get the number of elements in *QConstrArray Class* object.

**Example**

```
# Get the number of quadratic constraints in qconarr
qarrsize = qconarr.getSize()
```

### 24.2.16 QConstrBuilder Class

QConstrBuilder object contains operations related to temporary constraints when building quadratic constraints, and provides the following methods:

#### QConstrBuilder()

**Synopsis**

`QConstrBuilder()`

**Description**

Create an empty *QConstrBuilder Class* object.

**Example**

```
# Create an empty quadratic constraint builder
qconstrbuilder = QConstrBuilder()
```

#### QConstrBuilder.setBuilder()

**Synopsis**

`setBuilder(expr, sense, rhs)`

**Description**

Set expression, constraint type and RHS for quadratic constraint builder.

**Arguments**

`expr`

The expression to be set, which can be *Var Class* object, *LinExpr Class* object or *QuadExpr Class* object.

`sense`

Sense of quadratic constraint. The full list of available types can be found in *Constraint type* section.

`rhs`

Right hand side of quadratic constraint.

**Example**

```
# Set the expresson of quadratic constraint builder as: x+y, sense of constraint as␣
↪equal and RHS as 1
qconstrbuilder.setBuilder(x + y, COPT.LESS_EQUAL, 1.0)
```

#### QConstrBuilder.getQuadExpr()

**Synopsis**

`getQuadExpr()`

**Description**

Retrieve the expression of a quadratic constraint builder object.

**Example**

```
#  Retrieve the expression of a quadratic constraint builder
quadexpr = constrbuilder.getQuadExpr()
```

#### QConstrBuilder.getSense()

**Synopsis**

```
getSense()
```

**Description**

Retrieve the constraint sense of quadratic constraint builder object.

**Example**

```
# Retrieve the constraint sense of quadratic constraint builder object.
qconsense = qconstrbuilder.getSense()
```

### 24.2.17 QConstrBuilderArray Class

To facilitate users to operate on multiple *QConstrBuilder Class* objects, the Python interface of COPT provides QConstrArray object with the following methods:

#### QConstrBuilderArray()

**Synopsis**

```
QConstrBuilderArray(qconstrbuilders=None)
```

**Description**

Create a *QConstrBuilderArray Class* object.

If parameter `qconstrbuilders` is `None`, then create an empty *QConstrBuilderArray Class* object, otherwise initialize the newly created *QConstrBuilderArray Class* object by parameter `qconstrbuilders`.

**Arguments**

`qconstrbuilders`

Quadratic constraint builder to be added. Optional, `None` by default. It can be *QConstrBuilder Class* object, *QConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty QConstrBuilderArray object.
qconbuilderarr = QConstrBuilderArray()
# Create a QConstrBuilderArray object and initialize it with builders: qconbuilderx␣
↪and qconbuildery
qconbuilderarr = QConstrBuilderArray([qconbuilderx, qconbuildery])
```

### QConstrBuilderArray.pushBack()

**Synopsis**

```
pushBack(qconstrbuilder)
```

**Description**

Add single or multiple *QConstrBuilder Class* objects.

**Arguments**

`qconstrbuilder`

Builder of quadratic constraint to be added, which can be *QConstrBuilder Class* object, *QConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add quadratic constraint builder qconbuilderx to qconbuilderarr
qconbuilderarr.pushBack(qconbuilderx)
# Add quadratic constraint builders qconbuilderx and qconbuildery to qconbuilderarr
qconbuilderarr.pushBack([qconbuilderx, qconbuildery])
```

### QConstrBuilderArray.getBuilder()

**Synopsis**

```
getBuilder(idx)
```

**Description**

Retrieve the corresponding builder object according to the subscript of quadratic constraint builder in *QConstrBuilderArray Class* object.

**Arguments**

`idx`

Subscript of the quadratic constraint builder in the *QConstrBuilderArray Class* object, starting with 0.

**Example**

```
# Retrieve the builder with subscript 1 in qconbuilderarr
qconbuilder = qconbuilderarr.getBuilder(1)
```

### QConstrBuilderArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Get the number of elements in *QConstrBuilderArray Class* object.

**Example**

```
# Get the number of builders in qconbuilderarr
qarrsize = qconbuilderarr.getSize()
```

### 24.2.18 PsdConstraint Class

PsdConstraint object contains related operations of COPT positive semi-definite constraints and provides the following methods:

#### PsdConstraint.getName()

**Synopsis**

`getName()`

**Description**

Retrieve the name of positive semi-definite constraint.

**Example**

```
# Retrieve the name of positive semi-definite constraint 'con'.
conname = con.getName()
```

#### PsdConstraint.getIdx()

**Synopsis**

`getIdx()`

**Description**

Retrieve the subscript of positive semi-definite constraint in the model.

**Example**

```
# Retrieve the subscript of positive semi-definite constraint con.
conidx = con.getIdx()
```

#### PsdConstraint.setName()

**Synopsis**

`setName(newname)`

**Description**

Set the name of positive semi-definite constraint.

**Arguments**

`newname`

The name of positive semi-definite constraint to be set.

**Example**

```
# Set the name of positive semi-definite constraint 'con'.
con.setName('con')
```

### PsdConstraint.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Retrieve specified information. Return a constant.

**Arguments**

`infoname`

Name of the information to be obtained. Please refer to *Information* section for possible values.

**Example**

```
# Get the lower bound of positive semi-definite constraint con
conlb = con.getInfo(COPT.Info.LB)
```

### PsdConstraint.setInfo()

**Synopsis**

```
setInfo(infoname, newval)
```

**Description**

Set new information value to the specified positive semi-definite constraint.

**Arguments**

`infoname`

The name of the information to be set. Please refer to *Information* section for possible values.

`newval`

New information value to be set.

**Example**

```
# Set the lower bound of positive semi-definite constraint con
con.setInfo(COPT.Info.LB, 1.0)
```

### PsdConstraint.remove()

**Synopsis**

```
remove()
```

**Description**

Delete the positive semi-definite constraint from model.

**Example**

```
# Delete the positive semi-definite constraint 'conx'
conx.remove()
```

### 24.2.19 PsdConstrArray Class

To facilitate users to operate on multiple *PsdConstraint Class* objects, the Python interface of COPT provides PsdConstrArray class with the following methods:

#### PsdConstrArray()

**Synopsis**

`PsdConstrArray(constrs=None)`

**Description**

Create a *PsdConstrArray Class* object.

If parameter `constrs` is `None`, the create an empty *PsdConstrArray Class* object, otherwise initialize the newly created *PsdConstrArray Class* object with parameter `constrs` .

**Arguments**

`constrs`

Positive semi-definite constraints to be added. `None` by default.

`constrs` can be *PsdConstraint Class* object, *PsdConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty PsdConstrArray object
conarr = PsdConstrArray()
# Create an PsdConstrArray object containing positive semi-definite constraint conx␣
↪and cony
conarr = PsdConstrArray([conx, cony])
```

#### PsdConstrArray.pushBack()

**Synopsis**

`pushBack(constrs)`

**Description**

Add single or multiple *PsdConstraint Class* objects.

**Arguments**

`constrs`

Positive semi-definite constraints to be applied. `constrs` can be *PsdConstraint Class* object, *PsdConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add positive semi-definite constraint r to conarr
conarr.pushBack(r)
# Add positive semi-definite constraint r0 and r1 to conarr
conarr.pushBack([r0, r1])
```

#### PsdConstrArray.getPsdConstr()

**Synopsis**

```
getPsdConstr(idx)
```

**Description**

Retrieve the positive semi-definite constraint according to its subscript in *PsdConstrArray Class* object. Return a *PsdConstraint Class* object.

**Arguments**

`idx`

Subscript of the desired positive semi-definite constraint in *PsdConstrArray Class* object, starting with 0.

**Example**

```
# Retrieve the positive semi-definite constraint with subscript 1  in conarr
con = conarr.getPsdConstr(1)
```

#### PsdConstrArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Get the number of elements in *PsdConstrArray Class* object.

**Example**

```
# Get the number of positive semi-definite constraints in conarr
arrsize = conarr.getSize()
```

### 24.2.20 PsdConstrBuilder Class

PsdConstrBuilder object contains operations related to temporary constraints when building positive semi-definite constraints, and provides the following methods:

#### PsdConstrBuilder()

**Synopsis**

```
PsdConstrBuilder()
```

**Description**

Create an empty *PsdConstrBuilder Class* object

**Example**

```
# Create an empty positive semi-definite constraint builder
constrbuilder = PsdConstrBuilder()
```

### PsdConstrBuilder.setBuilder()

**Synopsis**

```
setBuilder(expr, sense, rhs)
```

**Description**

Set expression, constraint type and right hand side for positive semi-definite constraint builder.

**Arguments**

`expr`

The expression to be set, which can be *PsdVar Class* expression or *PsdExpr Class* expression.

`sense`

Sense of constraint. The full list of available types can be found in *Constraint type* section.

`rhs`

The right hand side of constraint.

**Example**

```
# Set the expresson of positive semi-definite constraint builder as: x + y == 1, and␣
↪sense of constraint as equal
constrbuilder.setBuilder(x + y, COPT.EQUAL, 1)
```

### PsdConstrBuilder.setRange()

**Synopsis**

```
setRange(expr, range)
```

**Description**

Set a range positive semi-definite constraint builder where `expr` is less than or equals to 0 and greater than or equals to - `range`.

**Arguments**

`expr`

The expression to be set, which can be *PsdVar Class* expression or *PsdExpr Class* expression.

`range`

Range of constraint, nonnegative constant.

**Example**

```
# Set a range positive semi-definite constraint builder: -1 <= x + y - 1 <= 0
constrbuilder.setRange(x + y - 1, 1)
```

#### PsdConstrBuilder.getPsdExpr()

**Synopsis**

getPsdExpr()

**Description**

Retrieve the expression of a positive semi-definite constraint builder object.

**Example**

```
#  Retrieve the expression of a positive semi-definite constraint builder
psdexpr = constrbuilder.getPsdExpr()
```

#### PsdConstrBuilder.getSense()

**Synopsis**

getSense()

**Description**

Retrieve the constraint sense of positive semi-definite constraint builder object.

**Example**

```
# Retrieve the constraint sense of positive semi-definite constraint builder object.
consense = constrbuilder.getSense()
```

#### PsdConstrBuilder.getRange()

**Synopsis**

getRange()

**Description**

Retrieve the range of positive semi-definite constraint builder object, i.e. length from lower bound to upper bound of the constraint

**Example**

```
# Retrieve the range of positive semi-definite constraint builder object
rngval = constrbuilder.getRange()
```

### 24.2.21 PsdConstrBuilderArray Class

To facilitate users to operate on multiple *PsdConstrBuilder Class* objects, the Python interface of COPT provides PsdConstrBuilderArray object with the following methods:

### PsdConstrBuilderArray()

#### Synopsis

```
PsdConstrBuilderArray(builders=None)
```

#### Description

Create a *PsdConstrBuilderArray Class* object.

If parameter `builders` is `None`, then create an empty *PsdConstrBuilderArray Class* object, otherwise initialize the newly created *PsdConstrBuilderArray Class* object by parameter `builders`.

#### Arguments

`builders`

Positive semi-definite constraint builder to be added. Optional, `None` by default. It can be *PsdConstrBuilder Class* object, *PsdConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

#### Example

```
# Create an empty PsdConstrBuilderArray object.
conbuilderarr = PsdConstrBuilderArray()
# Create a PsdConstrBuilderArray object containing builders: conbuilderx and␣
↪conbuildery
conbuilderarr = PsdConstrBuilderArray([conbuilderx, conbuildery])
```

### PsdConstrBuilderArray.pushBack()

#### Synopsis

```
pushBack(builder)
```

#### Description

Add single or multiple *PsdConstrBuilder Class* objects.

#### Arguments

`builder`

Builder of positive semi-definite constraint to be added, which can be *PsdConstrBuilder Class* object, *PsdConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

#### Example

```
# Add positive semi-definite constraint builder conbuilderx to conbuilderarr
conbuilderarr.pushBack(conbuilderx)
# Add positive semi-definite constraint builders conbuilderx and conbuildery to␣
↪conbuilderarr
conbuilderarr.pushBack([conbuilderx, conbuildery])
```

#### PsdConstrBuilderArray.getBuilder()

**Synopsis**

`getBuilder(idx)`

**Description**

Retrieve the corresponding builder object according to the subscript of positive semi-definite constraint builder in *PsdConstrBuilderArray Class* object.

**Arguments**

`idx`

Subscript of the positive semi-definite constraint builder in the *PsdConstrBuilderArray Class* object, starting with 0.

**Example**

```
# Retrieve the builder with subscript 1 in conbuilderarr
conbuilder = conbuilderarr.getBuilder(1)
```

#### PsdConstrBuilderArray.getSize()

**Synopsis**

`getSize()`

**Description**

Get the number of elements in *PsdConstrBuilderArray Class* object.

**Example**

```
# Get the number of builders in conbuilderarr
arrsize = conbuilderarr.getSize()
```

### 24.2.22 LmiConstraint Class

LmiConstraint object contains related operations of COPT LMI (Linear Matrix Inequality) constraints and provides the following methods:

#### LmiConstraint.getName()

**Synopsis**

`getName()`

**Description**

Retrieve the name of the *LmiConstraint Class* object.

**Example**

```
# Get name of the LmiConstraint con
conname = con.getName()
```

### LmiConstraint.getIdx()

**Synopsis**

```
getIdx()
```

**Description**

Retrieve the subscript of the LMI constraint in the model.

**Example**

```
# Retrieve the subscript of the LMI constraint
conidx = con.getIdx()
```

### LmiConstraint.getDim()

**Synopsis**

```
getDim()
```

**Description**

Retrieve the dimension of the LMI constraint.

**Example**

```
# Retrieve the dimension of the LMI constraint
conidx = con.getDim()
```

### LmiConstraint.getLen()

**Synopsis**

```
getLen()
```

**Description**

Retrieve the flattened length of the LMI constraint.

**Example**

```
# Retrieve the flattened length of the LMI constraint
conidx = con.getDim()
```

### LmiConstraint.setName()

**Synopsis**

```
setName(newname)
```

**Description**

Set the name of the LMI constraint to `newname` .

**Arguments**

`newname`

The name of the LMI constraint to be set.

**Example**

```
# Set the name of the LMI constraint
con.setName('con')
```

### LmiConstraint.setRhs()

**Synopsis**

`setRhs(mat)`

**Description**

Set the constant-term symmetric matrix of the LMI constraint.

**Arguments**

`mat`

The constant-term symmetric matrix of the LMI constraint to be set. It should be *SymMatrix Class* .

**Example**

```
# Set the constant-term symmetric of the LMI constraint
D = m.addSparseMat(2, [0, 1], [0, 1], [1.0, 1.0])
con.setRhs(D)
```

### LmiConstraint.getInfo()

**Synopsis**

`getInfo(infoname)`

**Description**

Retrieve the specified information with the name `infoname` .

**Arguments**

`infoname`

Name of the information to be obtained. Please refer to *Information* section for possible values.

**Example**

```
# Get slack of the LMI constraint con
conlb = con.getInfo(COPT.Info.Slack)
```

### LmiConstraint.remove()

**Synopsis**

`remove()`

**Description**

Remove the current LMI constraint from the model.

**Example**

```
# Remove the LMI constraint conx
conx.remove()
```

### LmiConstraint.shape

**Synopsis**

```
shape
```

**Description**

Shape of the `LmiConstraint` object.

**Return value**

Integer tuple.

### LmiConstraint.size

**Synopsis**

```
size
```

**Description**

Size of the `LmiConstraint` object.

**Return value**

Integer tuple.

### LmiConstraint.dim

**Synopsis**

```
dim
```

**Description**

Dimension of the `LmiConstraint` object.

**Return value**

Integer.

### LmiConstraint.len

**Synopsis**

```
len
```

**Description**

Flattened length of the `LmiConstraint` object.

**Return value**

Integer.

### 24.2.23 LmiConstrArray Class

To facilitate users to operate on multiple *LmiConstraint Class* objects, the Python interface of COPT provides LmiConstrArray class with the following methods:

#### LmiConstrArray()

**Synopsis**

`LmiConstrArray(constrs=None)`

**Description**

Create a *LmiConstrArray Class* object.

If parameter `constrs` is `None`, the create an empty *LmiConstrArray Class* object, otherwise initialize the newly created *LmiConstrArray Class* object with parameter `constrs` .

**Arguments**

`constrs`

Can be *LmiConstraint Class* object, *LmiConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty LmiConstrArray class object
conarr = LmiConstrArray()
# Create a LmiConstrArray class object and initialize it with the LMI constraints␣
↪conx and cony
conarr = LmiConstrArray([conx, cony])
```

#### LmiConstrArray.pushBack()

**Synopsis**

`pushBack(constr)`

**Description**

Add single or multiple *LmiConstraint Class* objects.

**Arguments**

`constr`

`constrs` can be *LmiConstraint Class* object, *LmiConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add LMI constraint r to conarr
conarr.pushBack(r)
# Add LMI constraint r0 and r1 to conarr
conarr.pushBack([r0, r1])
```

### LmiConstrArray.getLmiConstr()

**Synopsis**

```
getLmiConstr(idx)
```

**Description**

Retrieve the LMI constraint according to its subscript in *LmiConstrArray Class* object. Return a *LmiConstraint Class* object.

**Arguments**

`idx`

Subscript of the desired LMI constraint in *LmiConstrArray Class* object, starting with 0.

**Example**

```
# Get the LMI constraint with subscript 1 in conarr
con = conarr.getLmiConstr(1)
```

### LmiConstrArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Get the number of elements in *LmiConstrArray Class* object.

**Example**

```
# Get the number of LMI constraints in conarr
arrsize = conarr.getSize()
```

### LmiConstrArray.reserve()

**Synopsis**

```
reserve(n)
```

**Description**

Reserve space for *LmiConstrArray Class* objects of size n.

**Arguments**

`n`

The number of elements in the object *LmiConstrArray Class* .

### 24.2.24 NlConstraint Class

The `NlConstraint` class provides an interface for operations on nonlinear constraints in the COPT. It offers the following member functions:

#### NlConstraint.getName()

**Synopsis**

```
getName()
```

**Description**

Retrieve the name of the nonlinear constraint.

**Return value**

A string.

#### NlConstraint.getIdx()

**Synopsis**

```
getIdx()
```

**Description**

Retrieve the index of the nonlinear constraint in the model.

**Return value**

An integer.

#### NlConstraint.setName()

**Synopsis**

```
setName(newname)
```

**Description**

Set the name of the nonlinear constraint.

**Arguments**

`newname`

The new name of the constraint.

#### NlConstraint.getRhs()

**Synopsis**

```
getRhs()
```

**Description**

Retrieve the right hand side of nonlinear constraint.

**Return value**

A double value.

### NlConstraint.getSense()

**Synopsis**

```
getSense()
```

**Description**

Retrieve the type of nonlinear constraint.

**Return value**

A string.

### NlConstraint.setRhs()

**Synopsis**

```
setRhs(rhs)
```

**Description**

Set the right hand side value of the nonlinear constraint.

**Arguments**

`rhs`

The new right hand side value.

### NlConstraint.setSense()

**Synopsis**

```
setSense(sense)
```

**Description**

Set the sense of nonlinear constraint.

**Arguments**

`sense`

The sense of nonlinear constraint to be set.

### NlConstraint.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Retrieve specified information of the nonlinear constraint.

**Arguments**

`infoname`

Name of the information to be retrieved.

See *Information* for possible values.

**Return value**

A double value.

#### NlConstraint.setInfo()

**Synopsis**

```
setInfo(infoname, newval)
```

**Description**

Set the specified information value of the nonlinear constraint.

**Arguments**

`infoname`

Name of the information to be set.

See *Information* for possible values.

`newval`

The new information value.

#### NlConstraint.remove()

**Synopsis**

```
remove()
```

**Description**

Remove the current nonlinear constraint from the model.

### 24.2.25 NlConstrArray Class

The `NlConstrArray` class is used to manage and operate on a collection of nonlinear constraints. The following methods are provided:

#### NlConstrArray()

**Synopsis**

```
NlConstrArray(constrs=None)
```

**Description**

Construct a new nonlinear constraint array object.

The argument `constrs` can be a single *NlConstraint Class* object, an iterable of *NlConstraint Class* objects, or a mapping from arbitrary keys to constraint objects.

**Arguments**

`constrs`

Optional. Defaults to `None`.

#### NlConstrArray.pushBack()

**Synopsis**

```
pushBack(constrs)
```

**Description**

Append one or more *NlConstraint Class* objects to the current array.

**Arguments**

`constrs`

Could be a single *NlConstraint Class* object, an iterable of such objects, or a mapping from keys to constraints.

#### NlConstrArray.getNlConstr()

**Synopsis**

```
getNlConstr(idx)
```

**Description**

Retrieve the constraint object at the specified index.

**Arguments**

`idx`

The index of the constraint.

**Return value**

The corresponding *NlConstraint Class* object.

#### NlConstrArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Obtain the number of elements in *NlConstrArray Class* object.

**Return value**

An integer value.

### 24.2.26 NlConstrBuilder Class

The `NlConstrBuilder` class provides a builder interface for defining nonlinear constraints in the COPT. It provides the following functions:

### NlConstrBuilder()

**Synopsis**

```
NlConstrBuilder()
```

**Description**

Construct a new nonlinear constraint builder object.

### NlConstrBuilder.setBuilder()

**Synopsis**

```
setBuilder(expr, sense, rhs)
```

**Description**

Define the nonlinear expression, constraint sense, and right hand side for the builder.

**Arguments**

`expr`

The nonlinear expression.

`sense`

The constraint sense.

`rhs`

The right hand side.

### NlConstrBuilder.getNlExpr()

**Synopsis**

```
getNlExpr()
```

**Description**

Retrieve the nonlinear expression in the builder.

**Return value**

A *NlExpr Class* object.

### NlConstrBuilder.getSense()

**Synopsis**

```
getSense()
```

**Description**

Retrieve the constraint sense of the nonlinear constraint builder.

**Return value**

A string.

### 24.2.27 NlConstrBuilderArray Class

The NlConstrBuilderArray class is used to manage a collection of nonlinear constraint builders.

It provides the following member functions:

#### NlConstrBuilderArray()

**Synopsis**

```
NlConstrBuilderArray(constrs=None)
```

**Description**

Construct a new nonlinear constraint builder array.

The argument `constrs` can be a single *NlConstrBuilder Class* object, an iterable of builder objects, or a mapping from arbitrary keys to builder objects.

**Arguments**

`constrs`

Optional. Defaults to `None`.

#### NlConstrBuilderArray.pushBack()

**Synopsis**

```
pushBack(constrbuilder)
```

**Description**

Append one or more *NlConstrBuilder Class* objects to the array.

**Arguments**

`constrbuilder`

The builder(s) to append.

Can be a single *NlConstrBuilder Class* object, a *NlConstrBuilderArray Class*, a list, a dictionary, or a *tupledict Class*.

#### NlConstrBuilderArray.getBuilder()

**Synopsis**

```
getBuilder(idx)
```

**Description**

Retrieve the builder object at the specified index.

**Arguments**

`idx`

The index of the builder.

**Return value**

The corresponding *NlConstrBuilder Class* object.

#### NlConstrBuilderArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in the *NlConstrBuilderArray Class* object.

**Return value**

An integer.

### 24.2.28 SOS Class

SOS object contains related operations of COPT SOS constraints. The following methods are provided:

#### SOS.getIdx()

**Synopsis**

```
getIdx()
```

**Description**

Retrieve the subscript of SOS constraint in model.

**Example**

```
# Retrieve the subscript of SOS constraint sosx.
sosidx = sosx.getIdx()
```

#### SOS.remove()

**Synopsis**

```
remove()
```

**Description**

Delete the SOS constraint from model.

**Example**

```
# Delete the SOS constraint 'sosx'
sosx.remove()
```

### 24.2.29 SOSArray Class

To facilitate users to operate on a set of *SOS Class* objects, COPT designed SOSArray class in Python interface. The following methods are provided:

### SOSArray()

**Synopsis**

```
SOSArray(soss=None)
```

**Description**

Create a *SOSArray Class* object.

If parameter `soss` is `None`, then build an empty *SOSArray Class* object, otherwise initialize the newly created *SOSArray Class* object with `soss`.

**Arguments**

`soss`

SOS constraint to be added. Optional, `None` by default. It can be *SOS Class* object, *SOSArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create a new SOSArray object
sosarr = SOSArray()
# Create a SOSArray object, and initialize it with SOS constraints sosx and sosy.
sosarr = SOSArray([sosx, sosy])
```

### SOSArray.pushBack()

**Synopsis**

```
pushBack(sos)
```

**Description**

Add one or multiple *SOS Class* objects.

**Arguments**

`sos`

SOS constraints to be added, which can be *SOS Class* object, *SOSArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add SOS constraint sosx to sosarr
sosarr.pushBack(sosx)
# Add SOS constraints sosx and sosy to sosarr
sosarr.pushBack([sosx, sosy])
```

### SOSArray.getSOS()

**Synopsis**

```
getSOS(idx)
```

**Description**

Retrieve the corresponding SOS constraint according to its subscript in *SOSArray Class* object and return a *SOS Class* object.

**Arguments**

`idx`

Index of the SOS constraint in *SOSArray Class* object, starting with 0.

**Example**

```
# Retrieve the SOS constraint with index of 1 in sosarr
sos = sosarr.getSOS(1)
```

#### SOSArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in *SOSArray Class* object.

**Example**

```
# Retrieve the number of SOS constraints in sosarr.
arrsize = sosarr.getSize()
```

### 24.2.30 SOSBuilder Class

For easy access builders of SOS constraints, SOSBuilder class provides the following methods:

#### SOSBuilder()

**Synopsis**

```
SOSBuilder()
```

**Description**

Create an empty *SOSBuilder Class* object.

**Example**

```
# Create an empty SOSBuilder object.
sosbuilder = SOSBuilder()
```

#### SOSBuilder.setBuilder()

**Synopsis**

```
setBuilder(sostype, vars, weights=None)
```

**Description**

Set type, variable, weight of variable on *SOSBuilder Class* object.

**Arguments**

`sostype`

SOS constraint type. Full list of available types can be found in *SOS-constraint types*.

`vars`

Variables of SOS constraint, which can be *VarArray Class* object, list, dictionary or *tupledict Class* object.

`weights`
: Weights of variables in SOS constraint. Optional, `None` by default. Could be list, dictionary or *tupledict Class* object.

**Example**

```
# Set the type of SOS constraint builder as SOS1, variables x and y, weights of␣
↪variables as 1 and 2 respectively.
sosbuilder.setBuilder(COPT.SOS_TYPE1, [x, y], [1, 2])
```

### SOSBuilder.getType()

**Synopsis**

`getType()`

**Description**

Retrieve the SOS constraint type of *SOSBuilder Class* object.

**Example**

```
# Retrieve the type of SOS constraint builder sosx.
sostype = sosbuilder.getType(sosx)
```

### SOSBuilder.getVar()

**Synopsis**

`getVar(idx)`

**Description**

Retrieve the corresponding variables according to its index in *SOSBuilder Class* object, and return a *Var Class* object.

**Arguments**

`idx`

Index of the variable in *SOSBuilder Class* object, starting with 0.

**Example**

```
# Retrieve the variable in SOS constraint builder sosx with index of 1
sosvar = sosx.getVar(1)
```

### SOSBuilder.getVars()

**Synopsis**

`getVars()`

**Description**

Retrieve all variables in *SOSBuilder Class* objects, and return a *VarArray Class* object.

**Example**

```
# Retrieve all variables in SOS constraint builder sosx.
sosvars = sosx.getVars()
```

### SOSBuilder.getWeight()

**Synopsis**

```
getWeight(idx)
```

**Description**

Retrieve the corresponding weight of variable according to its index in *SOSBuilder Class* object.

**Arguments**

`idx`

Index of the variable in *SOSBuilder Class* object, starting with 0.

**Example**

```
# Retrieve the corresponding weight of variable according to its index in the SOS␣
↪constraint builder sosx.
sosweight = sosx.getWeight(1)
```

### SOSBuilder.getWeights()

**Synopsis**

```
getWeights()
```

**Description**

Retrieve weights of all the variables in *SOSBuilder Class* object.

**Example**

```
# Retrieve weights of all the variables in SOS constraint builder sosx.
sosweights = sosx.getWeights()
```

### SOSBuilder.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in *SOSBuilder Class* object.

**Example**

```
# Retrieve the number of elements in SOS constraint builder sosx.
sossize = sosx.getSize()
```

### 24.2.31 SOSBuilderArray Class

In order to facilitate users to operate on a set of *SOSBuilder Class* objects, COPT provides SOSBuilderArray class in Python interface, providing the following methods:

#### SOSBuilderArray()

**Synopsis**

`SOSBuilderArray(sosbuilders=None)`

**Description**

Create a *SOSBuilderArray Class* object.

If parameter `sosbuilders` is `None`, then create an empty *SOSBuilderArray Class* object, otherwise initialize the newly created *SOSBuilderArray Class* object with parameter `sosbuilders`.

**Arguments**

`sosbuilders`

SOS constraint builder to be added. Optional, `None` by default. Could be *SOSBuilder Class* object, *SOSBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty SOSBuilderArray object.
sosbuilderarr = SOSBuilderArray()
# Create a SOSBuilderArray object and initialize it with SOS constraint builder sosx␣
↪and sosy
sosbuilderarr = SOSBuilderArray([sosx, sosy])
```

#### SOSBuilderArray.pushBack()

**Synopsis**

`pushBack(sosbuilder)`

**Description**

Add one or multiple *SOSBuilder Class* objects.

**Arguments**

`sosbuilder`

SOS constraint builder to be added. Could be *SOSBuilder Class* object, *SOSBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add SOS constraint builder sosx to sosbuilderarr
sosbuilderarr.pushBack(sosx)
```

#### SOSBuilderArray.getBuilder()

**Synopsis**

`getBuilder(idx)`

**Description**

Retrieve the corresponding builder according to the index of SOS constraint builder in *SOSBuilderArray Class* object.

**Arguments**

`idx`

Index of the SOS constraint builder in *SOSBuilderArray Class* object, starting with 0.

**Example**

```
# Retrieve the SOS constraint builder with index of 1 in sosbuilderarr
sosbuilder = sosbuilderarr.getBuilder(1)
```

#### SOSBuilderArray.getSize()

**Synopsis**

`getSize()`

**Description**

Retrieve the number of elements in *SOSBuilderArray Class* object.

**Example**

```
# Retrieve the number of elements in sosbuilderarr
sosbuildersize = sosbuilderarr.getSize()
```

### 24.2.32 Cone Class

Cone object contains related operations of COPT Second-Order-Cone (SOC) constraints. The following methods are provided:

#### Cone.getIdx()

**Synopsis**

`getIdx()`

**Description**

Retrieve the subscript of SOC constraint in model.

**Example**

```
# Retrieve the subscript of SOC constraint cone.
coneidx = cone.getIdx()
```

#### Cone.remove()

**Synopsis**

`remove()`

**Description**

Delete the SOC constraint from model.

**Example**

```
# Delete the SOC constraint 'cone'
cone.remove()
```

### 24.2.33 ConeArray Class

To facilitate users to operate on a set of *Cone Class* objects, COPT designed ConeArray class in Python interface. The following methods are provided:

#### ConeArray()

**Synopsis**

`ConeArray(cones=None)`

**Description**

Create a *ConeArray Class* object.

If parameter `cones` is `None`, then build an empty *ConeArray Class* object, otherwise initialize the newly created *ConeArray Class* object with `cones`.

**Arguments**

`cones`

Second-Order-Cone constraint to be added. Optional, `None` by default. It can be *Cone Class* object, *ConeArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create a new ConeArray object
conearr = ConeArray()
# Create a ConeArray object, and initialize it with SOC constraints conex and coney.
conearr = ConeArray([conex, coney])
```

#### ConeArray.pushBack()

**Synopsis**

`pushBack(cone)`

**Description**

Add one or multiple *Cone Class* objects.

**Arguments**

`cone`

Second-Order-Cone constraints to be added, which can be *Cone Class* object, *ConeArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add SOC constraint conex to conearr
conearr.pushBack(conex)
# Add SOC constraints conex and coney to conearr
conearr.pushBack([conex, coney])
```

#### ConeArray.getCone()

**Synopsis**

```
getCone(idx)
```

**Description**

Retrieve the corresponding Second-Order-Cone (SOC) constraint according to its subscript in *ConeArray Class* object and return a *Cone Class* object.

**Arguments**

`idx`

Index of the SOC constraint in *ConeArray Class* object, starting with 0.

**Example**

```
# Retrieve the SOC constraint with index of 1 in conearr
cone = conearr.getCone(1)
```

#### ConeArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in *ConeArray Class* object.

**Example**

```
# Retrieve the number of SOC constraints in conearr.
arrsize = conearr.getSize()
```

### 24.2.34 ConeBuilder Class

For easy access builders of Second-Order-Cone (SOC) constraints, ConeBuilder class provides the following methods:

#### ConeBuilder()

**Synopsis**

```
ConeBuilder()
```

**Description**

Create an empty *ConeBuilder Class* object.

**Example**

```
# Create an empty ConeBuilder object.
conebuilder = ConeBuilder()
```

### ConeBuilder.setBuilder()

**Synopsis**

`setBuilder(conetype, vars)`

**Description**

Set type, variables of *ConeBuilder Class* object.

**Arguments**

`conetype`

Type of Second-Order-Cone (SOC) constraint. Full list of available types can be found in *SOC-constraint types*.

`vars`

Variables of SOC constraint, which can be *VarArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Set type as regular, variables as [z, x, y] for SOC constraint builder
conebuilder.setBuilder(COPT.CONE_QUAD, [z, x, y])
```

### ConeBuilder.getType()

**Synopsis**

`getType()`

**Description**

Retrieve the Second-Order-Cone (SOC) constraint type of *ConeBuilder Class* object.

**Example**

```
# Retrieve the type of SOC constraint builder conex.
conetype = conebuilder.getType(conex)
```

### ConeBuilder.getVar()

**Synopsis**

`getVar(idx)`

**Description**

Retrieve the corresponding variables according to its index in *ConeBuilder Class* object, and return a *Var Class* object.

**Arguments**

`idx`

Index of the variable in *ConeBuilder Class* object, starting with 0.

**Example**

```
# Retrieve the variable in SOC constraint builder conex with index of 1
conevar = conex.getVar(1)
```

#### ConeBuilder.getVars()

**Synopsis**

```
getVars()
```

**Description**

Retrieve all variables in *ConeBuilder Class* objects, and return a *VarArray Class* object.

**Example**

```
# Retrieve all variables in SOC constraint builder conex.
conevars = conex.getVars()
```

#### ConeBuilder.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in *ConeBuilder Class* object.

**Example**

```
# Retrieve the number of elements in SOC constraint builder conex.
conesize = conex.getSize()
```

### 24.2.35 ConeBuilderArray Class

In order to facilitate users to operate on a set of *ConeBuilder Class* objects, COPT provides ConeBuilderArray class in Python interface, providing the following methods:

#### ConeBuilderArray()

**Synopsis**

```
ConeBuilderArray(conebuilders=None)
```

**Description**

Create a *ConeBuilderArray Class* object.

If parameter `conebuilders` is `None`, then create an empty *ConeBuilderArray Class* object, otherwise initialize the newly created *ConeBuilderArray Class* object with parameter `conebuilders`.

**Arguments**

`conebuilders`

SOC constraint builder to be added. Optional, `None` by default. Could be *ConeBuilder Class* object, *ConeBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty ConeBuilderArray object.
conebuilderarr = ConeBuilderArray()
# Create a ConeBuilderArray object and initialize it with SOC constraint builder␣
↪conex and coney
conebuilderarr = ConeBuilderArray([conex, coney])
```

### ConeBuilderArray.pushBack()

**Synopsis**

`pushBack(conebuilder)`

**Description**

Add one or multiple *ConeBuilder Class* objects.

**Arguments**

`conebuilder`

SOC constraint builder to be added. Could be *ConeBuilder Class* object, *ConeBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add SOC constraint builder conex to conebuilderarr
conebuilderarr.pushBack(conex)
```

### ConeBuilderArray.getBuilder()

**Synopsis**

`getBuilder(idx)`

**Description**

Retrieve the corresponding builder according to the index of SOC constraint builder in *ConeBuilderArray Class* object.

**Arguments**

`idx`

Index of the SOC constraint builder in *ConeBuilderArray Class* object, starting with 0.

**Example**

```
# Retrieve the SOC constraint builder with index of 1 in conebuilderarr
ConeBuilder = conebuilderarr.getBuilder(1)
```

#### ConeBuilderArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in *ConeBuilderArray Class* object.

**Example**

```
# Retrieve the number of elements in conebuilderarr
conebuildersize = conebuilderarr.getSize()
```

### 24.2.36 ExpCone Class

ExpCone object contains related operations of COPT exponential cone constraints. The following methods are provided:

#### ExpCone.getIdx()

**Synopsis**

```
getIdx()
```

**Description**

Retrieve the subscript of exponential cone constraint in model.

**Example**

```
# Retrieve the subscript of exponential cone constraint cone.
coneidx = cone.getIdx()
```

#### ExpCone.remove()

**Synopsis**

```
remove()
```

**Description**

Delete the exponential cone constraint from model.

**Example**

```
# Delete the exponential cone constraint 'cone'
cone.remove()
```

### 24.2.37 ExpConeArray Class

To facilitate users to operate on a set of *ExpCone Class* objects, COPT designed ExpConeArray class in Python interface. The following methods are provided:

#### ExpConeArray()

**Synopsis**

`ExpConeArray(cones=None)`

**Description**

Create a *ExpConeArray Class* object.

If parameter `cones` is `None`, then build an empty *ExpConeArray Class* object, otherwise initialize the newly created *ExpConeArray Class* object with `cones`.

**Arguments**

`cones`

Exponential cone constraint to be added. Optional, `None` by default. It can be *ExpCone Class* object, *ExpConeArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create a new ExpConeArray object
conearr = ExpConeArray()
# Create a ExpConeArray object, and initialize it with exponential cone constraints␣
↪conex and coney.
conearr = ExpConeArray([conex, coney])
```

#### ExpConeArray.pushBack()

**Synopsis**

`pushBack(cone)`

**Description**

Add one or multiple *ExpCone Class* objects.

**Arguments**

`cone`

Exponential cone constraints to be added, which can be *ExpCone Class* object, *ExpConeArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Add exponential cone constraint conex to conearr
conearr.pushBack(conex)
# Add exponential cone constraints conex and coney to conearr
conearr.pushBack([conex, coney])
```

#### ExpConeArray.getCone()

**Synopsis**

```
getCone(idx)
```

**Description**

Retrieve the corresponding exponential cone constraint according to its subscript in *ExpConeArray Class* object and return a *ExpCone Class* object.

**Arguments**

`idx`

Index of the exponential cone constraint in *ExpConeArray Class* object, starting with 0.

**Example**

```
# Retrieve the exponential cone constraint with index of 1 in conearr
cone = conearr.getCone(1)
```

#### ExpConeArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in *ExpConeArray Class* object.

**Example**

```
# Retrieve the number of exponential cone constraints in conearr.
arrsize = conearr.getSize()
```

### 24.2.38 ExpConeBuilder Class

For easy access builders of exponential cone constraints, ExpConeBuilder class provides the following methods:

#### ExpConeBuilder()

**Synopsis**

```
ExpConeBuilder()
```

**Description**

Create an empty *ExpConeBuilder Class* object.

**Example**

```
# Create an empty ExpConeBuilder object.
ExpConeBuilder = ExpConeBuilder()
```

### ExpConeBuilder.setBuilder()

**Synopsis**

```
setBuilder(conetype, vars)
```

**Description**

Set type of variables of *ExpConeBuilder Class* object.

**Arguments**

`conetype`

Type of exponential cone constraint. Full list of available types can be found in *exponential cone types*.

`vars`

Variables of exponential cone constraint, which can be *VarArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Set type as regular, variables as [z, x, y] for exponential cone constraint builder
ExpConeBuilder.setBuilder(COPT.EXPCONE_PRIMAL, [z, x, y])
```

### ExpConeBuilder.getType()

**Synopsis**

```
getType()
```

**Description**

Retrieve the exponential cone constraint type of *ExpConeBuilder Class* object.

**Example**

```
# Retrieve the type of exponential cone constraint builder conex.
conetype = ExpConeBuilder.getType(conex)
```

### ExpConeBuilder.getVar()

**Synopsis**

```
getVar(idx)
```

**Description**

Retrieve the corresponding variables according to its index in *ExpConeBuilder Class* object, and return a *Var Class* object.

**Arguments**

`idx`

Index of the variable in *ExpConeBuilder Class* object, starting with 0.

**Example**

```
# Retrieve the variable in exponential cone constraint builder conex with index of 1
conevar = conex.getVar(1)
```

#### ExpConeBuilder.getVars()

**Synopsis**

```
getVars()
```

**Description**

Retrieve all variables in *ExpConeBuilder Class* objects, and return a *VarArray Class* object.

**Example**

```
# Retrieve all variables in exponential cone constraint builder conex.
conevars = conex.getVars()
```

#### ExpConeBuilder.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in *ExpConeBuilder Class* object.

**Example**

```
# Retrieve the number of elements in exponential cone constraint builder conex.
conesize = conex.getSize()
```

### 24.2.39 ExpConeBuilderArray Class

In order to facilitate users to operate on a set of *ExpConeBuilder Class* objects, COPT provides ExpConeBuilderArray class in Python interface, providing the following methods:

#### ExpConeBuilderArray()

**Synopsis**

```
ExpConeBuilderArray(ExpConeBuilders=None)
```

**Description**

Create a *ExpConeBuilderArray Class* object.

If parameter `ExpConeBuilders` is `None`, then create an empty *ExpConeBuilderArray Class* object, otherwise initialize the newly created *ExpConeBuilderArray Class* object with parameter `ExpConeBuilders`.

**Arguments**

`ExpConeBuilders`

Exponential cone constraint builder to be added. Optional, `None` by default. Could be *ExpConeBuilder Class* object, *ExpConeBuilderArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create an empty ExpConeBuilderArray object.
ExpConeBuilderarr = ExpConeBuilderArray()
# Create a ExpConeBuilderArray object and initialize it with exponential cone␣
↪constraint builder conex and coney
ExpConeBuilderarr = ExpConeBuilderArray([conex, coney])
```

### ExpConeBuilderArray.pushBack()

#### Synopsis

`pushBack(conebuilder)`

#### Description

Add one or multiple *ExpConeBuilder Class* objects.

#### Arguments

`conebuilder`

Exponential cone constraint builder to be added. Could be *ExpConeBuilder Class* object, *ExpConeBuilderArray Class* object, list, dictionary or *tupledict Class* object.

#### Example

```
# Add exponential cone constraint builder conex to ExpConeBuilderarr
ExpConeBuilderarr.pushBack(conex)
```

### ExpConeBuilderArray.getBuilder()

#### Synopsis

`getBuilder(idx)`

#### Description

Retrieve the corresponding builder according to the index of exponential cone constraint builder in *ExpConeBuilderArray Class* object.

#### Arguments

`idx`

Index of the exponential cone constraint builder in *ExpConeBuilderArray Class* object, starting with 0.

#### Example

```
# Retrieve the exponential cone constraint builder with index of 1 in␣
↪ExpConeBuilderarr
ExpExpConeBuilder = ExpConeBuilderarr.getBuilder(1)
```

#### ExpConeBuilderArray.getSize()

**Synopsis**

`getSize()`

**Description**

Retrieve the number of elements in *ExpConeBuilderArray Class* object.

**Example**

```
# Retrieve the number of elements in ExpConeBuilderarr
ExpConeBuildersize = ExpConeBuilderarr.getSize()
```

### 24.2.40 AffineCone Class

The AffineCone class encapsulates operations related to affine cone in COPT. The following methods are provided:

#### AffineCone.getIdx()

**Synopsis**

`getIdx()`

**Description**

Retrieve the index of the affine cone in the model.

**Example**

```
# Retrieve the index of the afcone
afconeidx = afcone.getIdx()
```

#### AffineCone.getName()

**Synopsis**

`getName()`

**Description**

Retrieve the name of the affine cone.

**Example**

```
# Retrieve the name of the afcone
afconename = afcone.getName()
```

#### AffineCone.setName()

**Synopsis**

`setName(newname)`

**Description**

Set the name of the affine cone.

**Arguments**

`newname`

The name to set for the affine cone.

**Example**

```
# Set the name of the afcone
afcone.setName("afcone")
```

#### AffineCone.remove()

**Synopsis**

`remove()`

**Description**

Remove the current affine cone from the model.

**Example**

```
# Remove the current afcone from the model
afcone.remove()
```

### 24.2.41 AffineConeArray Class

To facilitate user operations on a group of *AffineCone Class* objects, COPT introduces the AffineConeArray class. The following methods are provided:

#### AffineConeArray()

**Synopsis**

`AffineConeArray(cones=None)`

**Description**

Creates an *AffineConeArray Class* object.

If the argument `cones` is `None`, an empty *AffineConeArray Class* object is created. Otherwise, the new *AffineConeArray Class* object is initialized with the argument `cones`.

**Arguments**

`cones`

Affine cone constraints to be added. This is an optional parameter, the default value is `None`.

Acceptable values include *AffineCone Class* objects, *AffineConeArray Class* objects, lists, dictionaries, or *tupledict Class* objects.

**Example**

```
# Create an empty AffineConeArray object
afconearr = AffineConeArray()
# Create an AffineConeArray object initialized with afconex and afconey
afconearr = AffineConeArray([afconex, afconey])
```

### AffineConeArray.pushBack()

**Synopsis**

pushBack(cone)

**Description**

Adds one or more *AffineCone Class* objects.

**Arguments**

cone

Affine cone constraints to be added. Possible values include *AffineCone Class* objects, *AffineConeArray Class* objects, lists, dictionaries, or *tupledict Class* objects.

**Example**

```
# Add afconex to afconearr
afconearr.pushBack(afconex)
# Add afconex and afconey to afconearr
afconearr.pushBack([afconex, afconey])

dict_cone = {'AC': afcone1, 'CA': afcone2}
afconearr2 = AffineConeArray({'AC': afcone1, 'CA': afcone2})
```

### AffineConeArray.getCone()

**Synopsis**

getCone(idx)

**Description**

Retrieves the affine cone at the specified index in the *AffineConeArray Class* object. Returns an *AffineCone Class* object.

**Arguments**

idx

Index of the affine cone constraint in the *AffineConeArray Class* object. Indexing starts at 0.

**Example**

```
# Retrieve the affine cone at index 1 in afconearr
afcone = afconearr.getCone(1)
```

#### AffineConeArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieves the number of elements in the *AffineConeArray Class* object.

**Example**

```
# Get the number of affine cone in afconearr
arrsize = afconearr.getSize()
```

### 24.2.42 AffineConeBuilder Class

The *AffineConeBuilder* class is an encapsulation of the builder for constructing affine cone in COPT. The following methods are provided:

#### AffineConeBuilder()

**Synopsis**

```
AffineConeBuilder()
```

**Description**

Creates an empty *AffineConeBuilder Class* object.

**Example**

```
# Create an empty AffineConeBuilder object
afconebuilder = AffineConeBuilder()
```

#### AffineConeBuilder.setBuilder()

**Synopsis**

```
setBuilder(conetype, exprs)
```

**Description**

Sets up the affine cone builder according to `conetype` and `exprs` .

**Arguments**

`conetype`

The type of affine cone. For possible values, refer to *Second-Order Cone Constraint Types* and *Exponential Cone Constraint Types*.

`exprs`

Mathematical expressions forming the affine cone. Possible values include linear expressions or semidefinite expressions.

**Example**

```
# Set the affine cone builder's type to standard second-order cone,
# composed of z, x+y, y
afconebuilder = AffineConeBuilder()
afconebuilder.setBuilder(COPT.CONE_QUAD, [z, x+y, y])
```

### AffineConeBuilder.hasPsdTerm()

**Synopsis**

```
hasPsdTerm()
```

**Description**

Checks if the affine cone builder contains any semidefinite terms.

**Return value**

Bool. If `False`, the affine cone contains no semidefinite terms, and linear expressions can be directly retrieved.

### AffineConeBuilder.getType()

**Synopsis**

```
getType()
```

**Description**

Retrieves the affine cone type of the *AffineConeBuilder Class* object.

**Example**

```
# Retrieve the type of afconex
afconetype = afconebuilder.getType(afconex)
```

### AffineConeBuilder.getExpr()

**Synopsis**

```
getExpr(idx)
```

**Description**

Retrieves the expression at the specified index in the affine cone builder.

**Arguments**

`idx`

The index of the expression in the affine cone builder, starting with `0` .

**Return value**

If semidefinite terms exist, returns a *PsdExpr* object; otherwise, returns a *LinExpr* object.

**Example**

```
# Retrieve the expression at index 0 in the affine cone builder
expr = afconstrbuilder.getExpr(0)
```

### AffineConeBuilder.getPsdExpr()

**Synopsis**

`getPsdExpr(idx)`

**Description**

Retrieves the semidefinite expression at the specified index in the affine cone builder.

**Arguments**

`idx`

The index of the semidefinite expression in the affine cone builder, , starting with `0` .

**Return value**

Returns a *PsdExpr* object.

**Example**

```
# Retrieve the semidefinite expression at index 0 in the afconstrbuilder
psdexpr = afconstrbuilder.getPsdExpr(0)
```

### AffineConeBuilder.getExprs()

**Synopsis**

`getExprs(idx=None)`

**Description**

Retrieves a group of expressions from the affine cone builder at the specified indices.

**Arguments**

`idx`

The indices of the expressions in the affine cone builder.

If `idx` is `None`, all expressions in the affine cone are returned. If `idx` is a Python list, the expressions at the specified indices are returned.

**Return value**

If semidefinite terms exist, returns *PsdExpr* objects; otherwise, returns *LinExpr* objects.

**Example**

```
# Retrieve all expressions in the affine cone builder
allexprs = afconstrbuilder.getExprs()
```

#### AffineConeBuilder.getPsdExprs()

**Synopsis**

```
getPsdExprs(idx=None)
```

**Description**

Retrieves a group of semidefinite expressions from the affine cone builder at the specified indices.

**Arguments**

`idx`

The indices of the semidefinite expressions in the affine cone builder.

If `idx` is `None`, all semidefinite expressions in the affine cone are returned. If `idx` is a Python list, the semidefinite expressions at the specified indices are returned.

**Return value**

Returns *PsdExpr* objects.

**Example**

```
# Retrieve all semidefinite expressions in the afconstrbuilder
allpsdexprs = afconstrbuilder.getPsdExprs()
```

#### AffineConeBuilder.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieves the number of expressions in the affine cone builder.

**Example**

```
# Retrieve the number of expressions in the afconebuilder
afconesize = afconebuilder.getSize()
```

### 24.2.43 AffineConeBuilderArray Class

To facilitate user operations on a group of *AffineConeBuilder Class* objects, COPT introduces the AffineConeBuilderArray class, providing the following methods:

#### AffineConeBuilderArray()

**Synopsis**

```
AffineConeBuilderArray(conebuilders=None)
```

**Description**

Creates an *AffineConeBuilderArray Class* object.

If the argument `conebuilders` is `None`, an empty *AffineConeBuilderArray Class* object is created. Otherwise, the new *AffineConeBuilderArray Class* object is initialized with the argument `conebuilders`.

**Arguments**

`conebuilders`

Affine cone constraint builders to add. This is an optional argument, defaulting to `None`. Acceptable values include *AffineConeBuilder Class* objects, *AffineConeBuilderArray Class* objects, lists, dictionaries, or *tupledict Class* objects.

**Example**

```
# Create an empty AffineConeBuilderArray object
conebuilderarr = AffineConeBuilderArray()
# Create an AffineConeBuilderArray object initialized with conex and coney
conebuilderarr = AffineConeBuilderArray([conex, coney])
```

### AffineConeBuilderArray.pushBack()

**Synopsis**

`pushBack(conebuilder)`

**Description**

Adds one or more *AffineConeBuilder Class* objects.

**Arguments**

`conebuilder`

Affine cone builders to add. Acceptable values include *AffineConeBuilder Class* objects, *AffineConeBuilderArray Class* objects, lists, dictionaries, or *tupledict Class* objects.

**Example**

```
# Add affine cone builder afconex to afconebuilderarr
afconebuilderarr.pushBack(afconex)
```

### AffineConeBuilderArray.getBuilder()

**Synopsis**

`getBuilder(idx)`

**Description**

Retrieves the affine cone builder at the specified index in the *AffineConeBuilderArray Class* object.

**Arguments**

`idx`

The index of the affine cone builder in the *AffineConeBuilderArray Class* object. Indexing starts at 0.

**Example**

```
# Retrieve the second-order affine cone builder at index 1 in afconebuilderarr
affineConeBuilder = afconebuilderarr.getBuilder(1)
```

#### AffineConeBuilderArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieves the number of elements in the *AffineConeBuilderArray Class* object.

**Example**

```
# Get the number of elements in afconebuilderarr
afconebuildersize = afconebuilderarr.getSize()
```

### 24.2.44 GenConstr Class

For easy access to indicator constraints, COPT provides GenConstr class which containing the following methods:

#### GenConstr.getName()

**Synopsis**

```
getName()
```

**Description**

Retrieve the name of the indicator constraint in model.

**Example**

```
# Retrieve the name of indicator constraint indicx
indiname = indicx.getName()
```

#### GenConstr.setName()

**Synopsis**

```
setName(newname)
```

**Description**

Set the name of the indicator constraint in model with `newname` .

**Example**

```
# Set the name of the indicator constraint indicx with "if"
indicx.setName("if")
```

#### GenConstr.getIdx()

**Synopsis**

```
getIdx()
```

**Description**

Retrieve the subscript of indicator constraint in model.

**Example**

```
# Retrieve the index of indicator constraint indicx
indidx = indicx.getIdx()
```

#### GenConstr.remove()

**Synopsis**

```
remove()
```

**Description**

Delete the indicator constraint from model.

**Example**

```
# Delete indicator constraint 'indx'
indx.remove()
```

### 24.2.45 GenConstrArray Class

In order to facilitate users to operate on a set of *GenConstr Class* objects, COPT provides GenConstrArray class in Python interface, providing the following methods:

#### GenConstrArray()

**Synopsis**

```
GenConstrArray(genconstrs=None)
```

**Description**

Create a *GenConstrArray Class* object.

If parameter `genconstrs` is `None`, then create an empty *GenConstrArray Class* object, otherwise initialize the newly created *GenConstrArray Class* object with parameter `genconstrs`.

**Arguments**

`genconstrs`

Indicator constraint to be added. Optional, `None` by default. Could be *GenConstr Class* object, *GenConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Create a new GenConstrArray object
genconstrarr = GenConstrArray()
# Create a GenConstrArray object and user indicator constraints genx and geny to␣
↪initialize it.
genconstrarr = GenConstrArray([genx, geny])
```

### GenConstrArray.pushBack()

**Synopsis**

pushBack(genconstr)

**Description**

Add one or multiple *GenConstr Class* objects.

**Arguments**

constrs
: The indicator constraint to be added. Could be *GenConstr Class* object, *GenConstrArray Class* object, list, dictionary or *tupledict Class* object.

**Example**

```
# Aff indicator constraint genx to genconarr
genconarr.pushBack(genx)
# Add indicator constraint genx and geny to genconarr
genconarr.pushBack([genx, geny])
```

### GenConstrArray.getGenConstr()

**Synopsis**

getGenConstr(idx)

**Description**

Retrieve the corresponding indicator constraint according to the index of indicator constraint in *GenConstrArray Class* object, and return a *GenConstr Class* object.

**Arguments**

idx
: Index of the indicator constraint in *GenConstrArray Class*, starting with 0.

**Example**

```
# Retrieve the indicator constraint with index of 1 in genconarr
genconstr = genconarr.getGenConstr(1)
```

#### GenConstrArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in *GenConstrArray Class* object.

**Example**

```
# Retrieve the number of elements in genconarr
genconsize = genconarr.getSize()
```

### 24.2.46 GenConstrBuilder Class

GenConstrBuilder object contains operations for building indicator constraints, and provides the following methods:

#### GenConstrBuilder()

**Synopsis**

```
GenConstrBuilder()
```

**Description**

Create an empty *GenConstrBuilder Class* object.

**Example**

```
# Create an empty GenConstrBuilder object
genconbuilder = GenConstrBuilder()
```

#### GenConstrBuilder.setBuilder()

**Synopsis**

```
setBuilder(var, val, expr, sense, type=COPT.INDICATOR_IF)
```

**Description**

Set indicator variable, the value of indicator variable, the expression/sense of constraint, and the type of indicator constraint of the *GenConstrBuilder Class* object.

**Arguments**

`var`

Indicator variable.

`val`

Value of an indicator variable.

`expr`

Expression of linear constraint, which can be *Var Class* object or *LinExpr Class* object.

`sense`

Sense for the linear constraint. Please refer to *Constraint type* for possible values.

type

Type of the indicator constraint. The default value is COPT.INDICATOR_IF (If-Then). Please refer to *Indicator constraint type* for possible values.

**Example**

```
# Set indicator variable of indicator constraint builder to x. When x is true, the␣
↪linear constraint x + y == 1 holds
genconbuilder.setBuilder(x, True, x + y - 1, COPT.EQUAL)
```

### GenConstrBuilder.getBinVar()

**Synopsis**

getBinVar()

**Description**

Retrieve the indicator variable of a *GenConstrBuilder Class* object.

**Example**

```
# Retrieve the indicator variable of indicator constraint builder genbuilderx
indvar = genbuilderx.getBinVar()
```

### GenConstrBuilder.getBinVal()

**Synopsis**

getBinVal()

**Description**

Retrieve the value of indicator variable of a *GenConstrBuilder Class* object.

**Example**

```
# Retrieve the value when the indicator variable of indicator constraint builder␣
↪genbuilderx is valid
indval = genbuilderx.getBinVal()
```

### GenConstrBuilder.getExpr()

**Synopsis**

getExpr()

**Description**

Retrieve the linear expression of a *GenConstrBuilder Class* object.

**Example**

```
# Retrieve the linear expression of indicator constraint builder genbuilderx
linexpr = genbuilderx.getExpr()
```

#### GenConstrBuilder.getSense()

**Synopsis**

```
getSense()
```

**Description**

Retrieve the sense for the linear constraint of a *GenConstrBuilder Class* object.

**Example**

```
# Retrieve the sense for the linear constraint of indicator constraint builder␣
↪genbuilderx
linsense = genbuilderx.getSense()
```

#### GenConstrBuilder.getIndType()

**Synopsis**

```
getIndType()
```

**Description**

Retrieve the type for the indicator constraint of a *GenConstrBuilder Class* object.

**Example**

```
# Retrieve the type for the indicator constraint of indicator constraint builder␣
↪genbuilderx
linsense = genbuilderx.getIndType()
```

### 24.2.47 GenConstrBuilderArray Class

To facilitate users to operate on multiple *GenConstrBuilder Class* objects, the Python interface of COPT provides GenConstrBuilderArray object with the following methods:

#### GenConstrBuilderArray()

**Synopsis**

```
GenConstrBuilderArray(genconstrbuilders=None)
```

**Description**

Create a *GenConstrBuilderArray Class* object.

If argument `genconstrbuilders` is `None`, then create an empty *GenConstrBuilderArray Class* object; otherwise use the argument `genconstrbuilders` to initialize the newly created *GenConstrBuilderArray Class* object.

**Arguments**

`genconstrbuilders`

Indicator constraint builder to add. Optional, `None` by default. It can be *GenConstrBuilder Class* object, *GenConstrBuilderArray Class* object, list, dict, or *tupledict Class* object.

**Example**

```
# Create an empty GenConstrBuilderArray object
genbuilderarr = GenConstrBuilderArray()
# Create a GenConstrBuilderArray object and use indicator constraint builder␣
↪genbuilderx and genbuildery to initialize it.
genbuilderarr = GenConstrBuilderArray([genbuilderx, genbuildery])
```

### GenConstrBuilderArray.pushBack()

#### Synopsis

pushBack(genconstrbuilder)

#### Description

Add single or multiple *GenConstrBuilder Class* objects.

#### Arguments

genconstrbuilder

Indicator constraint builders to add, which can be *GenConstrBuilder Class* object, *GenConstrBuilderArray Class* object, list, dict, or *tupledict Class* object.

#### Example

```
# Add an indicator constraint builder to genbuilderarr
genbuilderarr.pushBack(genbuilderx)
# Add indicator constraint builders genbuilderx and genbuildery to genbuilderarr
genbuilderarr.pushBack([genbuilderx, genbuildery])
```

### GenConstrBuilderArray.getBuilder()

#### Synopsis

getBuilder(idx)

#### Description

Retrieve the indicator constraint builder according to its index in the *GenConstrBuilderArray Class* object, and return a *GenConstrBuilder Class* object.

#### Arguments

idx

Index of the indicator constraint builder in the *GenConstrBuilderArray Class* object, starting with 0.

#### Example

```
# Retrieve the indicator constraint builder whose index in genbuilderarr is 1
genbuilder = genbuilderarr.getBuilder(1)
```

#### GenConstrBuilderArray.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in the *GenConstrBuilderArray Class* object.

**Example**

```
# Retrieve the number of elements in genbuilderarr
genbuildersize = genbuilderarr.getSize()
```

### 24.2.48 Column Class

To facilitate users to model by column, the Python interface of COPT provides Column object with the following methods:

#### Column()

**Synopsis**

```
Column(constrs=0.0, coeffs=None)
```

**Description**

Create an *Column Class* object.

If argument `constrs` is `None` and argument `coeffs` is `None`, then create an empty *Column Class* object; otherwise use the argument `constrs` and `coeffs` to initialize the newly created *Column Class* object. If argument `constrs` is a *Constraint Class* or *Column Class* object, then argument `coeffs` is a constant. If argument `coeffs` is `None`, then it is considered to be constant 1.0; If argument `constrs` is a list and argument `coeffs` is `None`, then the elements of argument `constrs` are constraint-coefficient pairs; For other forms of arguments, call method `addTerms` to initialize the newly created *Column Class* object.

**Arguments**

`constrs`

Linear constraint.

`coeffs`

Coefficient for variables in the linear constraint.

**Example**

```
# Create an empty Column object
col = Column()
# Create a Column object and add two terms: coefficient is 2 in constraint conx and 3␣
↪in constraint cony
col = Column([(conx, 2), (cony, 3)])
# Create a Column object and add two terms: coefficient is 1 in constraint conxx and␣
↪2 in constraint conyy
col = Column([conxx, conyy], [1, 2])
```

### Column.getCoeff()

**Synopsis**

```
getCoeff(idx)
```

**Description**

Retrieve the coefficient according to its index in the *Column Class* object.

**Arguments**

`idx`

Index for the element, starting with 0.

**Example**

```
# Retrieve the coefficient whose index is 0 in col
coeff = col.getCoeff(0)
```

### Column.getConstr()

**Synopsis**

```
getConstr(idx)
```

**Description**

Retrieve the linear constraint according to its index in the *Column Class* object.

**Arguments**

`idx`

Index for the element, starting with 0.

**Example**

```
# Retrieve the linear constraint whose index is 1 in col
constr = col.getConstr(1)
```

### Column.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of elements in the *Column Class* object.

**Example**

```
# Retrieve the number of elements in col
colsize = col.getSize()
```

### Column.addTerm()

**Synopsis**

```
addTerm(constr, coeff=1.0)
```

**Description**

Add a new term.

**Arguments**

`constr`

The linear constraint for the term to add.

`coeff`

The coefficient for the term to add. Optional, 1.0 by default.

**Example**

```
# Add an term to col, whose constraint is cony and coefficient is 2.0
col.addTerm(cony, 2.0)
# Add an term to col, whose constraint is conx and coefficient is 1.0
col.addTerm(conx)
```

### Column.addTerms()

**Synopsis**

```
addTerms(constrs, coeffs)
```

**Description**

Add single or multiple terms.

If argument `constrs` is *Constraint Class* object, then argument `coeffs` is constant; If argument `constrs` is *ConstrArray Class* object or list, then argument `coeffs` is constant or list; If argument `constrs` is dictionary or *tupledict Class* object, then argument `coeffs` is constant, dict, or *tupledict Class* object.

**Arguments**

`constrs`

The linear constraints for terms to add.

`coeffs`

The coefficients for terms to add.

**Example**

```
# Add two terms: constraint conx with coefficient 2.0, constraint cony with␣
↪coefficient 3.0
col.addTerms([conx, cony], [2.0, 3.0])
```

### Column.addColumn()

**Synopsis**

```
addColumn(col, mult=1.0)
```

**Description**

Add a new column to current column.

**Arguments**

`col`

Column to add.

`mult`

Magnification coefficient for added column. Optional, 1.0 by default.

**Example**

```
# Add column coly to column colx. The magnification coefficient for coly is 2.0
colx.addColumn(coly, 2.0)
```

### Column.clone()

**Synopsis**

```
clone()
```

**Description**

Create a deep copy of a column.

**Example**

```
# Create a deep copy of column col
colcopy = col.clone()
```

### Column.remove()

**Synopsis**

```
remove(item)
```

**Description**

Remove a term from a column.

If argument `item` is a constant, then remove the term by its index; otherwise argument `item` is a *Constraint Class* object.

**Arguments**

`item`

Constant index or the linear constraint for the term to be removed.

**Example**

```
# Remove the term whose index is 2 from column col
col.remove(2)
# Remove the term of the linear constraint conx from col
col.remove(conx)
```

#### Column.clear()

**Synopsis**

```
clear()
```

**Description**

Remove all terms from a column.

**Example**

```
# Remove all terms from column col
col.clear()
```

### 24.2.49 ColumnArray Class

To facilitate users to operate on multiple *Column Class* objects, the Python interface of COPT provides ColumnArray object with the following methods:

#### ColumnArray()

**Synopsis**

```
ColumnArray(columns=None)
```

**Description**

Create a *ColumnArray Class* object.

If argument `columns` is `None`, then create an empty *ColumnArray Class* object; otherwise use argument `columns` to initialize the newly created *ColumnArray Class* object.

**Arguments**

`columns`

Columns to add. Optional, `None` by default. It can be *Column Class* object, *ColumnArray Class* object, list, dict, or *tupledict Class* object.

**Example**

```
# Create an empty ColumnArray object
colarr = ColumnArray()
# Create a ColumnArray object and use columns colx and coly to initialize it
colarr = ColumnArray([colx, coly])
```

#### ColumnArray.pushBack()

**Synopsis**

```
pushBack(column)
```

**Description**

Add single or multiple *Column Class* objects.

**Arguments**

`column`

Columns to add, which can be *Column Class* object, *ColumnArray Class* object, list, dict, or *tupledict Class* object.

**Example**

```
# Add column colx to colarr
colarr.pushBack(colx)
# Add columns colx and coly to colarr
colarr.pushBack([colx, coly])
```

### ColumnArray.getColumn()

**Synopsis**

`getColumn(idx)`

**Description**

Retrieve the column according to its index in a *ColumnArray Class* object. Return a *Column Class* object.

**Arguments**

`idx`

Index of the column in the *ColumnArray Class* object, starting with 0.

**Example**

```
# Retrieve the column whose index is 1 in colarr
col = colarr.getColumn(1)
```

### ColumnArray.getSize()

**Synopsis**

`getSize()`

**Description**

Retrieve the number of elements in a *ColumnArray Class* object.

**Example**

```
# Retrieve the number of element in colarr
colsize = colarr.getSize()
```

### ColumnArray.clear()

**Synopsis**

`clear()`

**Description**

Remove all terms from a *ColumnArray Class* object.

**Example**

```
# Remove all terms from colarr
colarr.clear()
```

### 24.2.50 MVar Class

The MVar class is used in COPT to build multi-dimensional variables and supports NumPy's multi-dimensional array operations. We recommend to generate it through the method `addVars` or `addMVar` of the model class, although it can also be converted and generated through the two built-in class methods `fromlist` and `fromvar` . It supports a unified set of indexing and slicing rules. For detailed usage, see *Matrix Modeling: Indexing and Slicing*.

The following member methods are provided:

#### MVar.fromlist()

**Synopsis**

`fromlist(vars)`

**Description**

Generate a *MVar Class* object from a set of *Var Class* objects. This is the class generation method and can be called directly without MVar object.

**Arguments**

`vars`

A set of Var objects, which can be a multi-dimensional list or ndarray.

**Return value**

new MVar object whose dimensions depend on the dimensions of the arguments vars.

**Example**

```
vars = model.addVars(4)
mx_1d = MVar.fromlist(vars)
mx_2d = MVar.fromlist([vars[0], vars[1]], [vars[2], vars[3]])
```

#### MVar.fromvar()

**Synopsis**

`fromvar(var)`

**Description**

Generate a 0-dimensional *MVar Class* object from a *Var Class* object. This is the class generation method and can be called directly without MVar object.

**Arguments**

`var`

A Var object.

**Return value**

The new 0-dimensional MVar object.

**Example**

```
x = model.addVar()
mx_0d = MVar.fromvar(x)
```

### MVar.clone()

**Synopsis**

`clone()`

**Description**

Deep-copy a *MVar Class* object.

**Return value**

new MVar object

**Example**

```
# Create a 2-D variable and make a copy. Note that the actual variable is␣
↪not incremented.
mx = model.addMVar((3, 2), nameprefix="mx")
mx_copy = mx.clone()
```

### MVar.diagonal()

**Synopsis**

`diagonal(offset=0, axis1=0, axis2=1)`

**Description**

Generate a *MVar Class* object whose elements are the elements on the diagonal of the original MVar object.

**Arguments**

`offset`

Optional parameter, indicating the offset of the diagonal, the default value is 0. If the value is greater than 0, it means the diagonal upward offset; if the value is less than 0, it means the diagonal downward offset.

`axis1`

Optional parameter, the axis to use as the first axis of the 2D sub MVar, from where the diagonal should start. The default first axis is 0.

`axis2`

Optional parameter, the axis to use as the second axis of the 2D sub MVar, from where the diagonal should start. The default second axis is 1.

**Return value**

new MVar object

**Example**

```
mx = model.addMVar((5, 5), nameprefix="mx")
diag_m0 = mx.diagonal()
diag_a1 = mx.diagonal(1)
diag_b1 = mx.diagonal(-1)
```

### MVar.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Get the information value of each variable inside MVar.

**Arguments**

`infoname`

The name of the information being queried. Please refer to *Information Section* for possible values.

**Return value**

Returns a *NdArray Class* object with the same dimension as the `MVar` object, whose elements are the information values of the corresponding variable. In legacy matrix modeling mode ( `Model.matrixmodelmode = "legacy"` ) a NumPy ndarray is returned instead, see *Matrix Modeling*.

**Example**

```
mx = model.addMVar(3)
print(mx.getInfo("LB"))
```

### MVar.item()

**Synopsis**

```
item()
```

**Description**

Get the Var variable inside the 0-dimensional MVar. Raises a ValueError exception if the MVar object is not 0-dimensional.

**Return value**

Returns the Var object.

**Example**

```
mx = model.addMVar(3)
var = mx[0].item()
```

### MVar.reshape()

**Synopsis**

```
reshape(shape, order='C')
```

**Description**

Returns a new MVar object whose elements remain unchanged but whose shape is transformed by the parameter shape.

**Arguments**

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MVar object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language, that is, it is stored in rows; it can also be set to the character 'F', that is, it is stored in columns, and it is compatible with the Fortune language.

**Return value**

Returns a new MVar object with the same elements as the original MVar object but with a different shape.

**Example**

```
mx = model.addMVar(6)
mx_2x3 = mx.reshape((2, 3))
```

### MVar.setInfo()

**Synopsis**

`setInfo(infoname, newval)`

**Description**

Sets the information value of each variable inside the MVar.

**Arguments**

`infoname`

The name of the information being queried. Please refer to *Information Section* for possible values.

`newval`

The information value to be set.

**Example**

```
mx = model.addMVar(3)
mx.setInfo("ub", 9.0)
mx.setInfo(COPT.Info.LB, 0.0)
```

### MVar.sum()

**Synopsis**

`sum(axis=None)`

**Description**

Sum the variables in the MVar, returning a new *MLinExpr Class* object.

**Arguments**

`axis`

Optional integer parameter, the default value is None, that is, to sum up variables one by one. Otherwise, sum over the given axis.

**Return value**

Returns an MLinExpr object representing the sum of the corresponding variables.

**Example**

```
mx = model.addMVar((3, 5))
sum_all = mx.sum() #Return 0-dimensional MLinExpr object
sum_row = mx.sum(axis = 0) #Return a 1-dimensional MLinExpr object with a␣
↪shape of (5, )
```

### MVar.tolist()

**Synopsis**

```
tolist()
```

**Description**

Convert an MVar object to a one-dimensional list of Var objects.

**Return value**

Returns a one-dimensional list containing Var objects.

**Example**

```
mx = model.addMVar((3, 5))
print(mx.tolist())
```

### MVar.transpose()

**Synopsis**

```
transpose()
```

**Description**

Generates a new MVar object that is the transpose of the original MVar object.

**Return value**

Return the new MVar object.

**Example**

```
mx = model.addMVar((3, 5))
print(mx.transpose().shape) #its shape is (5, 3)
```

### MVar.ndim

**Synopsis**

```
ndim
```

**Description**

Dimensions of the MVar object.

**Return value**

Integer value.

**Example**

```
mx = model.addMVar((3, 5))
print(mx.ndim) #ndim = 2
```

### MVar.shape

**Synopsis**

shape

**Description**

The shape of the MVar object.

**Return value**

Integer tuple.

**Example**

```
mx = model.addMVar((3,))
print(mx.shape) # shape = (3, )
```

### MVar.size

**Synopsis**

size

**Description**

The number of Var variables in the MVar object.

**Return value**

Integer value.

**Example**

```
mx = model.addMVar((3, 4))
print(mx.size) # size = 12
```

### MVar.T

**Synopsis**

T

**Description**

Transpose of the MVar object. Similar to the class method transpose().

**Return value**

Returns the transposed MVar object.

**Example**

```
mx = model.addMVar((3, 4))
print(mx.T.shape) # shape = (4, 3)
```

### 24.2.51 MConstr Class

The MConstr class holds multi-dimensional linear constraints in COPT and supports NumPy's multi-dimensional array operations. It is generated by the method `addConstrs` or `addMConstr` of the model class. It supports a unified set of indexing and slicing rules. For detailed usage, see *Matrix Modeling: Indexing and Slicing*.

The following member methods are provided:

#### MConstr.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Get the information value of each constraint within MConstr.

**Arguments**

`infoname`

The name of the information being queried. Please refer to *Information Section* for possible values.

**Return value**

Returns a *NdArray Class* object with the same dimension as the MConstr object, whose elements are the information values of the corresponding constraint. In legacy matrix modeling mode ( `Model.matrixmodelmode = "legacy"` ) a NumPy ndarray is returned instead, see *Matrix Modeling*.

**Example**

```
a = np.random.rand(4)
mx = m.addMVar((4, 3), nameprefix="mx")
b = np.random.rand(3)
mc = m.addConstrs(a @ mx <= b)
print(mc.getInfo("pi"))
```

#### MConstr.item()

**Synopsis**

```
item()
```

**Description**

Get the constraint object in 0-dimensional MConstr. If the MConstr object is not 0-dimensional, a ValueError exception is raised.

**Return value**

Returns the linear constraint object.

**Example**

```
mc = m.addConstrs(a @ mx <= b)
print(mc[0].item())
```

### MConstr.reshape()

#### Synopsis

```
reshape(shape, order='C')
```

#### Description

Returns a new MConstr object whose elements remain unchanged but whose shape is transformed by the argument shape.

#### Arguments

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MConstr object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language, that is, it is stored in rows; it can also be set to the character 'F', that is, it is stored in columns, and it is compatible with the Fortune language.

#### Return value

Returns a new MConstr object with the same elements as the original MConstr object but with a different shape.

#### Example

```
mc = m.addConstrs(a @ mx <= b)
mc_2x2 = mc.reshape((2, 2))
```

### MConstr.setInfo()

#### Synopsis

```
setInfo(infoname, newval)
```

#### Description

Set information values for each constraint within MConstr.

#### Arguments

`infoname`

The name of the information to be set. Please refer to *Information Section* for possible values.

`newval`

The new value to be set.

#### Example

```
mc = model.addConstrs(a @ mx <= b)
mc.setInfo("obj", 9.0)
```

### MConstr.tolist()

**Synopsis**

```
tolist()
```

**Description**

Convert the MConstr object to a one-dimensional list of constraints.

**Return value**

Returns a one-dimensional list containing Constraint objects.

**Example**

```
mc = m.addConstrs(a @ mx <= b)
print(mc.tolist())
```

### MConstr.transpose()

**Synopsis**

```
transpose()
```

**Description**

Generates a new MConstr object that is the transpose of the original MConstr object.

**Return value**

Returns the transposed MConstr object.

**Example**

```
mc = m.addConstrs(a @ mx <= b)
print(mc.transpose())
```

### MConstr.ndim

**Synopsis**

```
ndim
```

**Description**

Dimensions of the MConstr object.

**Return value**

Integer value.

**Example**

```
mc = m.addConstrs(a @ mx <= b)
print(mc.ndim)
```

### MConstr.shape

**Synopsis**

shape

**Description**

The shape of the MConstr object.

**Return value**

Integer tuple.

**Example**

```
mc = m.addConstrs(a @ mx <= b)
print(mc.shape)
```

### MConstr.size

**Synopsis**

size

**Description**

The number of constraints of the MConstr object.

**Return value**

Integer value.

**Example**

```
mc = m.addConstrs(a @ mx <= b)
print(mc.size)
```

### MConstr.T

**Synopsis**

T

**Description**

Transpose of the MConstr object. Similar to the class method transpose().

**Return value**

Returns the transposed MConstr object.

**Example**

```
A = np.ones([2, 4])
mx = m.addMVar((4, 3), nameprefix="mx")
mc = m.addConstrs(A @ X == 0.0)
print(mc.T.shape) # shape = (3, 2)
```

### 24.2.52 MConstrBuilder Class

The MConstrBuilder class is used to build multi-dimensional linear constraints in COPT and supports NumPy's multi-dimensional array operations. Users might create a MConstrBuilder object by its constructor with a list of *ConstrBuilder Class* objects, or simply by overloaded comparison operator of *MLinExpr Class*. The following member methods are provided:

#### MConstrBuilder()

**Synopsis**

`MConstrBuilder(args, shape=None)`

**Description**

constructor of MConstrBuilder.

**Arguments**

`args`

one or a set of *ConstrBuilder Class* objects, in form of Python list or NumPy ndarray.

`shape`

an integer, or tuple of integers, which is the shape of new MConstrBuilder object.

**Example**

```
vars = m.addVars(4)
builders = [x <= 1.0 for x in vars]
mcb = MConstrBuilder(builders, (2, 2))

# or by overloaded comparison operator of MVar
mx = m.addMVar((3, 2))
mcb = mx >= 1.0

# or immediately passed to addConstrs()
model.addConstrs(mx >= 1.0)
```

### 24.2.53 MQConstr Class

The MQConstr class holds multi-dimensional quadratic constraints in COPT and supports NumPy's multi-dimensional array operations. In experimental version of matrix modeling, it is generated by the method `addQConstr` or `addMQConstr` of the model class. It supports a unified set of indexing and slicing rules. For detailed usage, see *Matrix Modeling: Indexing and Slicing*.

The following member methods are provided:

### MQConstr.getInfo()

**Synopsis**

```
getInfo(infoname)
```

**Description**

Get the information value of each quadratic constraint within MQConstr.

**Arguments**

`infoname`

The name of the information being queried. Please refer to *Information Section* for possible values.

**Return value**

Returns a built-in NdArray with the same shape as the MQConstr object, whose elements are the information values of the corresponding constraints.

**Example**

```
mx = m.addMVar((1, 3), nameprefix="mx")
mc = m.addQConstr(mx @ mx.T <= 1.0)
print(mc.getInfo("x"))
```

### MQConstr.item()

**Synopsis**

```
item()
```

**Description**

Get the quadratic constraint object in MQConstr object. If the MQConstr object has more than one item, an exception of ValueError is raised.

**Return value**

Returns the quadratic constraint object.

**Example**

```
mc = m.addQConstr(mx @ mx.T <= 1.0)
print(mc.item())
```

### MQConstr.reshape()

**Synopsis**

```
reshape(shape, order='C')
```

**Description**

Returns a new MQConstr object whose elements remain unchanged but whose shape is transformed by the argument shape.

**Arguments**

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MQConstr object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language. The other option 'F' is not implemented yet.

**Return value**

Returns a new MQConstr object with the same elements as the original MQConstr object but with a different shape.

**Example**

```
mc = m.addQConstr(mx.T @ mx <= 1.0)
mc_1x9 = mc.reshape((1, 9))
```

### MQConstr.setInfo()

**Synopsis**

`setInfo(infoname, newval)`

**Description**

Set information values for each quadratic constraint within MQConstr.

**Arguments**

`infoname`

The name of the information to be set. Please refer to *Information Section* for possible values.

`newval`

The new value to be set.

**Example**

```
mc = model.addQConstr(mx @ mx.T <= 1.0)
mc.setInfo("LB", 9.0)
```

### MQConstr.tolist()

**Synopsis**

`tolist()`

**Description**

Convert the MQConstr object to a one-dimensional list of quadratic constraints.

**Return value**

Returns a one-dimensional list containing *QConstraint Class* objects.

**Example**

```
mc = m.addQConstr(mx.T @ mx <= 1.0)
print(mc.tolist())
```

### MQConstr.transpose()

**Synopsis**

`transpose()`

**Description**

Generates a new MQConstr object that is the transpose of the original MQConstr object.

**Return value**

Returns the transposed MQConstr object.

**Example**

```
mc = m.addQConstr(mx.T @ mx <= 1.0)
print(mc.transpose())
```

### MQConstr.ndim

**Synopsis**

`ndim`

**Description**

Dimensions of the MQConstr object.

**Return value**

Integer value.

**Example**

```
mc = m.addQConstr(mx.T @ mx <= 1.0)
print(mc.ndim)
```

### MQConstr.shape

**Synopsis**

`shape`

**Description**

The shape of the MQConstr object.

**Return value**

Integer tuple.

**Example**

```
mc = m.addQConstr(mx.T @ mx <= 1.0)
print(mc.shape)
```

#### MQConstr.size

**Synopsis**

`size`

**Description**

The number of constraints of the MQConstr object.

**Return value**

Integer value.

**Example**

```
mc = m.addQConstr(mx.T @ mx <= 1.0)
print(mc.size)
```

#### MQConstr.T

**Synopsis**

`T`

**Description**

Transpose of the MQConstr object. Similar to the class method `transpose()` .

**Return value**

Returns the transposed MQConstr object.

**Example**

```
A = np.ones([4, 3])
mx = model.addMVar((3, 4), nameprefix="mx")
mc = model.addQConstr(mx @ A @ mx == 0.0)
print(mc.shape) # shape = (3, 4)
print(mc.T.shape) # shape = (4, 3)
```

### 24.2.54 MQConstrBuilder Class

The MQConstrBuilder class is used to build multi-dimensional quadratic constraints in COPT and supports NumPy's multi-dimensional array operations. Users might create a MQConstrBuilder object by its constructor with a list of *QConstrBuilder Class* objects, or simply by overloaded comparison operator of *MQuadExpr Class*. The following member methods are provided:

#### MQConstrBuilder()

**Synopsis**

`MQConstrBuilder(args, shape=None)`

**Description**

constructor of MQConstrBuilder.

**Arguments**

`args`

one or a set of *QConstrBuilder Class* objects, in form of Python list or NumPy ndarray.

`shape`

an integer, or tuple of integers, which is the shape of new MQConstrBuilder object.

**Example**

```
x = model.addVar()
mqcb = MQConstrBuilder(x * x <= 9.0)

# or by overloaded comparison operator of MQuadExpr
mx = model.addMVar(3, 3)
mqcb = mx @ mx >= 1.0

# or immediately passed to addConstrs()
ma = model.addMVar(2)
A = np.full((2,3), 1)
mb = model.addMVar(3)
model.addQConstr(ma @ A @ mb <= 1.0)
```

### 24.2.55 MPsdConstr Class

The *MPsdConstr* class in the COPT is used to construct multi-dimensional semidefinite constraints. It is created via the *addConstr* or *addConstrs* methods of the model class. It supports a unified set of indexing and slicing rules. For detailed usage, see *Matrix Modeling: Indexing and Slicing*.

The following member methods are provided:

#### MPsdConstr.getInfo()

**Synopsis**

`getInfo(infoname)`

**Description**

Retrieves the information value for each semidefinite constraint in the *MPsdConstr*.

**Arguments**

`infoname`

The name of the information to retrieve. Possible values are detailed in *Information Section*.

**Return value**

Returns a *NdArray Class* object with the same shape as the MPsdConstr object, whose elements are the information values of the corresponding semidefinite constraints.

**Example**

```
mpsdcon = model.addConstr(barX[:-1, :-1].sum() == 1)
print(mpsdcon.UB)
print(mpsdcon.getInfo("UB"))
```

### MPsdConstr.setInfo()

#### Synopsis

```
setInfo(infoname, newval)
```

#### Description

Sets the information value for each semidefinite constraint in the *MPsdConstr*.

#### Arguments

`infoname`

The name of the information to set. Possible values are detailed in *Information Section*.

`newval`

The new value to set for the information.

#### Example

```
barX = model.addPsdVars(3, "BAR_X")
mY = model.addMVar(2, nameprefix="M_X")
mpsdCon = model.addConstrs(barX[:-1, :-1].diagonal() == mY)
mpsdCon.UB = 10
mpsdCon.setInfo("UB", 10)
```

### MPsdConstr.item()

#### Synopsis

```
item()
```

#### Description

Retrieves the semidefinite constraint object in a 0-dimensional *MPsdConstr*. If the *MPsdConstr* object is not 0-dimensional, raises a *ValueError*.

#### Return value

Returns a *PsdConstraint* object.

#### Example

```
barX = model.addPsdVars(3, "BAR_X")
mY = model.addMVar(2, nameprefix="M_X")
mpsdCon = model.addConstrs(barX[:-1, :-1].diagonal() == mY)
psdCon = mpsdCon[1].item()
```

### MPsdConstr.clone()

#### Synopsis

```
clone()
```

#### Description

Creates a deep copy of an *MPsdConstr Class* object.

#### Return value

Returns a new *MPsdConstr* object.

#### Example

```
barX = model.addPsdVars(3, "BAR_X")
mY = model.addMVar(2, nameprefix="M_X")
mpsdCon = model.addConstrs(barX[:-1, :-1].diagonal() == mY)
mpsdCon_copy = mpsdCon.clone()
```

### 24.2.56 MPsdConstrBuilder Class

The *MPsdConstrBuilder* class in COPT is used as a generator for multi-dimensional semidefinite constraints. It can be generated from a group of *PsdConstrBuilder Class* objects or by overloading the comparison operators of *MVar Class* and *MPsdExpr Class* objects. The following methods are provided:

#### MPsdConstrBuilder()

**Synopsis**

`MPsdConstrBuilder()`

**Description**

Creates an empty *MPsdConstrBuilder Class* object.

**Example**

```
# Create an empty semidefinite constraint builder
constrbuilder = MPsdConstrBuilder()
```

#### MPsdConstrBuilder.setBuilder()

**Synopsis**

`setBuilder(expr, sense, rhs)`

**Description**

Sets the expression and constraint type for the multi-dimensional semidefinite constraint builder.

**Arguments**

`expr`

The expression to set. Possible values include constants, *MVar Class* objects, *MLinExpr Class* objects, or *MPsdExpr Class* objects.

`sense`

The constraint type. Possible values are detailed in *Constraint Types*.

`rhs`

The right-hand side of the constraint. Possible values include constants, *MVar Class* objects, *MLinExpr Class* objects, or *MPsdExpr Class* objects.

**Example**

```
# Set the expression for the multi-dimensional PSD constraint builder: x + y == 1
constrbuilder.setBuilder(x + y, COPT.EQUAL, 1)
```

### MPsdConstrBuilder.setRange()

**Synopsis**

```
setRange(expr, range)
```

**Description**

Sets the expression and the range for the multi-dimensional PSD constraint builder. The format is *expr* less than or equal to 0, and greater than or equal to *-range*.

**Arguments**

`expr`

The expression to set. Possible values include *MPsdExpr Class* objects.

`range`

The right-hand side of the range constraint. Must be non-negative.

**Example**

```
# Set the expression for the multi-dimensional PSD constraint builder: x + y - 1,␣
↪with range 1
constrbuilder.setRange(x + y - 1, 1)
```

### MPsdConstrBuilder.getPsdExpr()

**Synopsis**

```
getPsdExpr()
```

**Description**

Retrieves the expression of the multi-dimensional PSD constraint builder.

**Example**

```
# Retrieve the expression of the multi-dimensional PSD constraint builder
psdexpr = constrbuilder.getPsdExpr()
```

### MPsdConstrBuilder.getSense()

**Synopsis**

```
getSense()
```

**Description**

Retrieves the constraint type of the multi-dimensional PSD constraint builder.

**Example**

```
# Retrieve the constraint type of the multi-dimensional PSD constraint builder
consense = constrbuilder.getSense()
```

#### MPsdConstrBuilder.getRange()

**Synopsis**

`getRange()`

**Description**

Retrieves the range right-hand side of the multi-dimensional PSD constraint builder, representing the range between the upper and lower bounds.

**Example**

```
# Retrieve the range right-hand side of the multi-dimensional PSD constraint builder
rngval = constrbuilder.getRange()
```

### 24.2.57 MLinExpr Class

The MLinExpr class is used in COPT to build multi-dimensional linear expressions and supports NumPy's multi-dimensional array operations. The MLinExpr object with an initial value of 0.0 can be generated by the class generation method `zeros()` , or by *MVar Class* object to generate a linear combination. It supports a unified set of indexing and slicing rules. For detailed usage, see *Matrix Modeling: Indexing and Slicing*.

The following member methods are provided:

#### MLinExpr.zeros()

**Synopsis**

`zeros(shape)`

**Description**

This is the class generation method and can be called directly without the MLinExpr object.

**Arguments**

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MLinExpr object.

**Return value**

new MLinExpr object.

**Example**

```
mexpr = MLinExpr.zeros((2,3))
x = model.addVar()
mexpr += x
```

### MLinExpr.clear()

**Synopsis**

```
clear()
```

**Description**

Resets each element of the *MLinExpr Class* object to 0.0.

**Example**

```
mexpr = 2.0 * model.addMVar(3)
mexpr.clear()
```

### MLinExpr.clone()

**Synopsis**

```
clone()
```

**Description**

Deep-copy a *MLinExpr Class* object.

**Return value**

new MLinExpr object

**Example**

```
mexpr = 2.0 * model.addMVar(3)
mexpr_copy = mexpr.clone()
```

### MLinExpr.getValue()

**Synopsis**

```
getValue()
```

**Description**

Get the evaluation of each linear expression within the *MLinExpr Class* object.

**Return value**

Returns a *NdArray Class* object of the same dimensions as the MLinExpr object whose elements are the evaluations of the corresponding expression. In legacy matrix modeling mode ( `Model.matrixmodelmode = "legacy"` ) a NumPy ndarray is returned instead, see *Matrix Modeling*.

**Example**

```
a = np.random.rand(4)
mx = m.addMVar((4, 3), nameprefix="mx")
mexpr = a @ mx
mc = m.addConstrs(mexpr <= 1.0)
m.solve()
print(mexpr.getValue())
```

### MLinExpr.item()

**Synopsis**

```
item()
```

**Description**

Get the constraint object in 0-dimensional MLinExpr. Raises a ValueError exception if the MLinExpr object is not 0-dimensional.

**Return value**

Returns the linear constraint object.

**Example**

```
mexpr = 2.0 * model.addMVar(3)
print(mexpr[0].item())
```

### MLinExpr.reshape()

**Synopsis**

```
reshape(shape, order='C')
```

**Description**

Returns a new MLinExpr object whose elements remain unchanged but whose shape is transformed by the argument shape.

**Arguments**

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MLinExpr object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language, that is, it is stored in rows; it can also be set to the character 'F', that is, it is stored in columns, and it is compatible with the Fortune language.

**Return value**

Returns a new MLinExpr object with the same elements as the original MLinExpr object but with a different shape.

**Example**

```
mc = m.addConstrs(a @ mx <= b)
mc_2x2 = mc.reshape((2, 2))
```

### MLinExpr.sum()

**Synopsis**

```
sum(axis=None)
```

**Description**

Sum the variables in the MLinExpr object, returning a new *MLinExpr Class* object.

**Arguments**

`axis`

Optional integer parameter, the default value is None, that is, to sum up variables one by one. Otherwise, sum over the given axis.

**Return value**

Returns an MLinExpr object representing the sum of the corresponding linear expressions.

**Example**

```
mexpr = 2.0 * model.addMVar((3, 5))
sum_all = mexpr.sum() #return 0-dimensional MLinExpr object
sum_row = mexpr.sum(axis = 0) #Return a 1-dimensional MLinExpr object with␣
↪a shape of (5, )
```

### MLinExpr.tolist()

**Synopsis**

```
tolist()
```

**Description**

Converts an MLinExpr object to a one-dimensional list whose elements are linear expressions.

**Return value**

Return a 1D list containing *LinExpr Class*.

**Example**

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.tolist())
```

### MLinExpr.transpose()

**Synopsis**

```
transpose()
```

**Description**

Generates a new MLinExpr object that is the transpose of the original MLinExpr object.

**Return value**

Returns the transposed MLinExpr object.

**Example**

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.transpose())
```

### MLinExpr.ndim

**Synopsis**

ndim

**Description**

*MLinExpr Class* Dimensions of the object.

**Return value**

Integer value.

**Example**

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.ndim)
```

### MLinExpr.shape

**Synopsis**

shape

**Description**

*MLinExpr Class* The shape of the object.

**Return value**

Integer tuple.

**Example**

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.shape)
```

### MLinExpr.size

**Synopsis**

size

**Description**

The number of elements of *MLinExpr Class* object.

**Return value**

Integer value.

**Example**

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.size)
```

### MLinExpr.T

**Synopsis**

T

**Description**

Transpose of *MLinExpr Class* object. Similar to the class method transpose().

**Return value**

Returns the transposed MLinExpr object.

**Example**

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.T.shape) # The transposed shape is (5, 3)
```

### MLinExpr.__eq__()

**Synopsis**

__eq__()

**Description**

Overload the == operator to build a *MConstrBuilder Class* object, which can be passed as the first argument to *Model.addConstrs*.

**Return value**

a *MConstrBuilder Class* object.

**Example**

```
model.addConstrs(A @ x == 1.0)
```

### MLinExpr.__ge__()

**Synopsis**

__ge__()

**Description**

Overload the >= operator to build a *MConstrBuilder Class* object, which can be passed as the first argument to *Model.addConstrs*.

**Return value**

a *MConstrBuilder Class* object.

**Example**

```
model.addConstrs(A @ x >= 1.0)
```

#### MLinExpr.__le__()

**Synopsis**

`__le__()`

**Description**

Overload the <= operator to build a *MConstrBuilder Class* object, which can be passed as the first argument to *Model.addConstrs*.

**Return value**

a *MConstrBuilder Class* object.

**Example**

```
model.addConstrs(A @ x <= 1.0)
```

### 24.2.58 MQuadExpr Class

The MQuadExpr class is used in COPT to construct multi-dimensional quadratic expressions and supports NumPy's multi-dimensional array operations. The MQuadExpr object with an initial value of 0.0 can be generated by the class generation method `zeros()` , or by pairing two *MVar Class* objects are generated by (matrix) multiplication. It supports a unified set of indexing and slicing rules. For detailed usage, see *Matrix Modeling: Indexing and Slicing*.

The following member methods are provided:

#### MQuadExpr.zeros()

**Synopsis**

`zeros(shape)`

**Description**

This is the class generation method and can be called directly without an MQuadExpr object.

**Arguments**

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MQuadExpr object.

**Return value**

new MQuadExpr object.

**Example**

```
mqx = MQuadExpr.zeros((2,3)) # shape = (2, 3)
x = model.addVar()
mqx += 2.0 * x * x           # broadcast scalar
mqx += model.addMVar(3)      # broadcast MVar of shape (3,)
```

### MQuadExpr.clear()

**Synopsis**

```
clear()
```

**Description**

Resets each element of the *MQuadExpr Class* object to 0.0.

**Example**

```
ma = model.addMVar(3, nameprefix='a')
mb = model.addMVar(3, nameprefix='b')
mqx = ma * mb     # elementwise multiply, shape = (3,)
mqx.clear()
print(mqx)        # result is [0.0, 0.0, 0.0]
```

### MQuadExpr.clone()

**Synopsis**

```
clone()
```

**Description**

Deep-copy a *MQuadExpr Class* object.

**Return value**

new MQuadExpr object

**Example**

```
mx = model.addMVar((3, 3), nameprefix='mx')
mqx = 2.0 * mx @ mx       # matrix multiply, shape = (3, 3)
mqx_copy = mqx.clone()
mqx_copy.clear()
print(mqx)                # mqx is untouched
```

### MQuadExpr.getValue()

**Synopsis**

```
getValue()
```

**Description**

Get the evaluation of each linear expression within the *MQuadExpr Class* object.

**Return value**

Returns a *NdArray Class* object of the same dimensions as the MQuadExpr object whose elements are the evaluations of the corresponding expression. In legacy matrix modeling mode ( `Model.matrixmodelmode = "legacy"` ) a NumPy ndarray is returned instead, see *Matrix Modeling*.

**Example**

```
A = np.eye(3)
mx = m.addMVar(3, nameprefix="mx")
mqx = mx @ A @ mx         # 0-D MQuadExpr, shape = ()
m.addQConstr(mqx <= 9.0)
```

(continues on next page)

(continued from previous page)

```
m.solve()
print(mqx.getValue())
```

### MQuadExpr.item()

**Synopsis**

```
item()
```

**Description**

Get the constraint object in the 0-dimensional MQuadExpr. Raises a ValueError exception if the MQuadExpr object is not 0-dimensional.

**Return value**

Returns the linear constraint object.

**Example**

```
x = m.addVar()
mqx = MQuadExpr.zeros(3) + x * x
print(mqx[1].item()) # Return QuadExpr(x * x)
```

### MQuadExpr.reshape()

**Synopsis**

```
reshape(shape, order='C')
```

**Description**

Returns a new MQuadExpr object whose elements are unchanged but whose shape is transformed by the argument shape.

**Arguments**

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MQuadExpr object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language, that is, it is stored in rows; it can also be set to the character 'F', that is, it is stored in columns, and it is compatible with the Fortune language.

**Return value**

Returns a new MQuadExpr object with the same elements as the original MQuadExpr object but with a different shape.

**Example**

```
mqx = MQuadExpr.zeros(6)
mqx_2x3 = mqx.reshape((2, 3))
```

### MQuadExpr.sum()

**Synopsis**

```
sum(axis=None)
```

**Description**

Sums the variables in the MQuadExpr object, returning a new *MQuadExpr Class* object.

**Arguments**

`axis`

Optional integer parameter, the default value is None, that is, to sum up variables one by one. Otherwise, sum over the given axis.

**Return value**

Returns an MQuadExpr object representing the sum of the corresponding linear expressions.

**Example**

```
ma = model.addMVar((2, 3), nameprefix='ma')
mb = model.addMVar((3, 2), nameprefix='mb')
mqx = ma @ mb
sum_all = mqx.sum() # Return 0-dimensional MQuadExpr object
sum_row = mqx.sum(axis = 0) # Return a 1-dimensional MQuadExpr object with␣
↪a shape of (2, )
```

### MQuadExpr.tolist()

**Synopsis**

```
tolist()
```

**Description**

Converts an MQuadExpr object to a one-dimensional list whose elements are linear expressions.

**Return value**

Return a 1D list containing *LinExpr Class*.

**Example**

```
print(MQuadExpr.zeros((2,3)).tolist()) # a list of length 6
```

### MQuadExpr.transpose()

**Synopsis**

```
transpose()
```

**Description**

Generates a new MQuadExpr object that is the transpose of the original MQuadExpr object.

**Return value**

Returns the transposed MQuadExpr object.

**Example**

```
mqx = MQuadExpr.zeros((2,3))
print(mqx.transpose().shape) # shape = (3, 2)
```

### MQuadExpr.ndim

**Synopsis**

ndim

**Description**

*MQuadExpr Class* Dimensions of the object.

**Return value**

Integer value.

**Example**

```
mqx = MQuadExpr.zeros((2,3))
print(mqx.ndim)  # ndim = 2
```

### MQuadExpr.shape

**Synopsis**

shape

**Description**

*MQuadExpr Class* The shape of the object.

**Return value**

Integer tuple.

**Example**

```
print(MQuadExpr.zeros((2,3)).shape) # shape = (2, 3)
```

### MQuadExpr.size

**Synopsis**

size

**Description**

The number of elements of *MQuadExpr Class* object.

**Return value**

Integer value.

**Example**

```
mqx = MQuadExpr.zeros((2,3))
print(mqx.size)  # size= 6
```

### MQuadExpr.T

**Synopsis**

T

**Description**

Transpose of *MQuadExpr Class* object. Similar to the class method transpose().

**Return value**

Returns the transposed MQuadExpr object.

**Example**

```
mqx = MQuadExpr.zeros((2,3))
print(mqx.T.shape) # shape = (3, 2)
```

### MQuadExpr.__eq__()

**Synopsis**

__eq__()

**Description**

Overload the == operator to build a *MQConstrBuilder Class* object, which can be passed as the first argument to *Model.addQConstr*.

**Return value**

a *MQConstrBuilder Class* object.

**Example**

```
model.addQConstr(x @ Q @ y == 1.0)
```

### MQuadExpr.__ge__()

**Synopsis**

__ge__()

**Description**

Overload the >= operator to build a *MQConstrBuilder Class* object, which can be passed as the first argument to *Model.addQConstr*.

**Return value**

a *MQConstrBuilder Class* object.

**Example**

```
model.addQConstr(x @ Q @ y >= 1.0)
```

#### MQuadExpr.__le__()

**Synopsis**

```
__le__()
```

**Description**

Overload the <= operator to build a *MQConstrBuilder Class* object, which can be passed as the first argument to *Model.addQConstr*.

**Return value**

a *MQConstrBuilder Class* object.

**Example**

```
model.addQConstr(x @ Q @ y <= 1.0)
```

### 24.2.59 NdArray Class

The NdArray class is a built-in multi-dimensional array class in COPT. It represents a table of elements of the same type, indexed by a tuple of integers. It supports a unified set of indexing and slicing rules. For detailed usage, see *Matrix Modeling: Indexing and Slicing*.

The following methods are provided:

#### NdArray()

**Synopsis**

```
NdArray(args=None, dtype=None, shape=None)
```

**Description**

Create a *NdArray Class* object.

**Return value**

Returns a NdArray object.

**Example**

```
# Create a NdArray object with a shape of 3x3 and initialize its elements␣
↪to 0
ndmat = NdArray(shape=(3, 3))
```

#### NdArray.item()

**Synopsis**

```
item()
```

**Description**

Gets the single element within a 0-dimensional NdArray object. If the NdArray object is not 0-dimensional, a `ValueError` exception will be triggered.

**Return value**

Returns the type of the elements in the 0-dimensional NdArray object. (For example: `"float"` or `"int"`, etc.)

**Example**

```
ndmat = NdArray(args=1.1, shape=(1,))
# Type of value is "float"
value = ndmat.item()
```

### NdArray.reshape()

**Synopsis**

```
reshape(shape, order='C')
```

**Description**

Returns a new NdArray object whose elements remain unchanged where shape is transformed according to the shape in the arguments.

**Arguments**

`shape`

The value could be an integer or a tuple of integers, representing the shape of the new NdArray object.

`order`

Optional. The default is the character 'C', which means it is compatible with C language (stored in rows). The current version does not yet support the character 'F' (stored in columns).

**Return value**

Returns a new NdArray object with the same elements as the original one but a different shape.

**Example**

```
ndmat = NdArray(shape=(6,))
ndmat_2x3 = ndmat.reshape((2, 3))
```

### NdArray.sum()

**Synopsis**

```
sum(axis=None)
```

**Description**

Compute the sum of elements in NdArray along the given axis.

If `axis` is not specified, all elements are summed; otherwise, the summation is performed along the specified axis.

**Arguments**

`axis`

Optional integer argument specifying the axis.

The default value is `None`.

**Return value**

If `axis` is `None`, returns a scalar result representing the sum of all elements. The numeric type of the returned value is consistent with the element type of the NdArray.

If `axis` is specified, returns a new NdArray object whose dimension is reduced by one. The elements represent the sums computed along the given axis.

**Example**

```
ndmat = NdArray(args=1.1, shape=(2, 2))
sum_all = ndmat.sum()         # Returns a scalar result
sum_row = ndmat.sum(axis=0)   # Returns a 1-D NdArray
```

### NdArray.prod()

**Synopsis**

`prod(axis=None)`

**Description**

Compute the product of elements in NdArray along the given axis.

If `axis` is not specified, the product is taken over all elements; otherwise, the product is computed along the specified axis.

**Arguments**

`axis`

Optional integer argument specifying the axis.

The default value is `None`.

**Return value**

If `axis` is `None`, returns a scalar result representing the product of all elements. The numeric type of the returned value is consistent with the element type of the NdArray.

If `axis` is specified, returns a new NdArray object whose dimension is reduced by one. The elements represent the products computed along the given axis.

**Example**

```
ndmat = NdArray(args=2.0, shape=(2, 2))
prod_all = ndmat.prod()         # Returns a scalar result
prod_row = ndmat.prod(axis=0)   # Returns a 1-D NdArray
```

### NdArray.dot()

**Synopsis**

`dot(other)`

**Description**

Compute the dot product between the current NdArray and the given object.

**Arguments**

`other`

The object used for the dot product computation.

Acceptable values include an iterable object or an NdArray. The length must match that of the current NdArray.

**Return value**

Returns a scalar result representing the dot product computation. The numeric type of the returned value is consistent with the element type of the NdArray.

**Example**

```
a = NdArray(args=[1.0, 2.0, 3.0], shape=(3, ))
b = NdArray(args=[4.0, 5.0, 6.0], shape=(3, ))

val = a.dot(b)  # Returns a scalar result
```

### NdArray.tolist()

**Synopsis**

`tolist()`

**Description**

Convert a NdArray object into a list.

**Return value**

Returns a list object.

**Example**

```
# Type of object mat_tolist is "list"
mat_tolist = ndmat.tolist()
```

### NdArray.tonumpy()

**Synopsis**

`tonumpy()`

**Description**

Convert a NdArray object to a NumPy ndarray object.

**Return value**

Returns a numpy ndarray object.

**Example**

```
# Type of object mat_tolist is "numpy.ndarray"
mat_tonumpy = ndmat.tonumpy()
```

### NdArray.fill()

**Synopsis**

`fill(value)`

**Description**

Fills each element in the NdArray object with the specified value.

**Arguments**

`value`

The new value of each element in the NdArray object.

**Return value**

Returns a NdArray object.

**Example**

```
mat_fillvalue = ndmat.fill(100.0)
```

### NdArray.expand()

**Synopsis**

`expand(axis=0)`

**Description**

Expand the NdArray object into an N+1 dimensional shape on the axis.

**Arguments**

`axis`

The specified dimension, which defaults to 0 (the first dimension).

**Return value**

Returns a N+1 dimensional NdArray object.

**Example**

```
mat_1 = ndmat.expand()
```

### NdArray.squeeze()

**Synopsis**

`squeeze(axis=0)`

**Description**

Reduce the NdArray object to an N-1 dimensional shape on the axis.

**Arguments**

`axis`

The specified dimension, default to 0 (the first dimension).

**Return value**

Returns a N-1 dimensional NdArray object.

**Example**

```
mat_1 = ndmat.squeeze()
```

### NdArray.flatten()

**Synopsis**

`flatten()`

**Description**

Expand a NdArray object into a one-dimensional shape.

**Return value**

Returns a new one-dimensional NdArray object.

**Example**

```
ndmat = NdArray(shape=(2, 2))
# The shape of mat_1 is (4,)
mat_1 = ndmat.flatten()
```

### NdArray.setItem()

**Synopsis**

`setItem(idx, value)`

**Description**

Sets the value of the element according to the given index in the NdArray object.

**Arguments**

`idx`

The specified one-dimensional index is the corresponding one after flattening NdArray into one dimension.

`value`

The new value of the specified element.

**Return value**

Returns a NdArray object in which the element value of `idx` is set to `value`.

**Example**

```
# Set the value of the element at index 0 to 100
mat_1 = ndmat.setItem(0, 100)
```

### NdArray.transpose()

**Synopsis**

`transpose()`

**Description**

Generates a new NdArray object, which is the transpose of the original one.

**Return value**

Returns a new NdArray object.

**Example**

```
ndmat = NdArray(shape=(3, 5))
print(ndmat.transpose().shape) #shape=(5, 3)
```

### NdArray.clone()

**Synopsis**

```
clone()
```

**Description**

Creates a deep copy of the NdArray object.

**Return value**

Returns a new NdArray object with its own copy of the data.

**Example**

```
ndmat = NdArray(shape=(3, 5))
ndmat_copy = ndmat.clone()
```

### NdArray.diagonal()

**Synopsis**

```
diagonal(offset=0, axis1=0, axis2=1)
```

**Description**

Generate a *NdArray Class* object, in which elements are the ones on the diagonal of the original NdArray object.

**Arguments**

`offset`

Optional argument, indicating the offset of the diagonal, the default value is 0. If the value is greater than 0, it represents the upward offset of the diagonal. If the value is less than 0, it represents the downward offset of the diagonal.

`axis1`

Optional argument. Axis to be used as the first axis of the 2-D sub-NdArrays from which the diagonals should be taken. Defaults to first axis (0).

`axis2`

Optional argument. Axis to be used as the second axis of the 2-D sub-NdArrays from which the diagonals should be taken. Defaults to second axis (1).

**Return value**

A new NdArray object.

**Example**

```
ndmat = NdArray(shape=(3, 3),args=[[1,1,1],[2,2,2],[3,3,3]])

diag_m0 = ndmat.diagonal(0)
diag_a1 = ndmat.diagonal(1)
diag_b1 = ndmat.diagonal(-1)
```

### NdArray.ndim

**Synopsis**

`ndim`

**Description**

The dimensions of the NdArray object.

**Return value**

An integer Value

**Example**

```
ndmat = NdArray((3, 5))
print(ndmat.ndim) # ndim = 2
```

### NdArray.shape

**Synopsis**

`shape`

**Description**

The shape of the NdArray object.

**Return value**

An integer tuple.

**Example**

```
ndmat = NdArray((3, 5))
print(ndmat.shape) # shape = (3, 5)
```

### NdArray.pick()

**Synopsis**

`pick(indexes)`

**Description**

Retrieves a new multi-dimensional array from the *NdArray* object according to the specified indices.

**Arguments**

`indexes`

The specified indices.

**Return value**

Returns a new *NdArray* object.

### NdArray.hstack()

**Synopsis**

```
hstack(other)
```

**Description**

Stacks another *NdArray* object along the horizontal dimension (last dimension) to form a new *NdArray* object.

**Arguments**

`other`

Another *NdArray* object.

**Return value**

Returns a new *NdArray* object.

### NdArray.vstack()

**Synopsis**

```
vstack(other)
```

**Description**

Stacks another *NdArray* object along the vertical dimension to form a new *NdArray* object.

**Arguments**

`other`

Another *NdArray* object.

**Return value**

Returns a new *NdArray* object.

### NdArray.stack()

**Synopsis**

```
stack(other, axis)
```

**Description**

Stacks another *NdArray* object along the specified axis to form a new *NdArray* object.

**Arguments**

`other`

Another *NdArray* object.

`axis`

The specified axis index.

**Return value**

Returns a new *NdArray* object.

#### NdArray.size

**Synopsis**

```
size
```

**Description**

The number of elements in the NdArray object.

**Return value**

An integer value.

**Example**

```
ndmat = NdArray((3, 5))
print(ndmat.size) # size = 15
```

#### NdArray.T

**Synopsis**

```
T
```

**Description**

Transpose of NdArray objects, similar to the method `NdArray.transpose()` .

**Return value**

Returns the transposed NdArray object.

**Example**

```
ndmat = NdArray(shape=(3, 5))
print(ndmat.T.shape) # shape = (5, 3)
```

### 24.2.60 ExprBuilder Class

ExprBuilder object contains operations related to building linear expressions, and provides the following methods:

#### ExprBuilder()

**Synopsis**

```
ExprBuilder(arg1=0.0, arg2=None)
```

**Description**

Create a *ExprBuilder Class* object.

If argument `arg1` is constant, argument `arg2` is `None`, then create a *ExprBuilder Class* object and initialize it using argument `arg1`. If argument `arg1` is *Var Class* or *ExprBuilder Class* object, and argument `arg2` is constant or considered to be constant 1.0 when argument `arg2` is `None`, then initialize the newly created *ExprBuilder Class* object using arguments `arg1` and `arg2`. If argument `arg1` and `arg2` are list objects, then they are variables and coefficients used to initialize the newly created *ExprBuilder Class* object.

**Arguments**

`arg1`

Optional, 0.0 by default.

`arg2`

Optional, `None` by default.

**Example**

```
# Create a new ExprBuilder object and initialize it to 0.0
expr0 = ExprBuilder()
# Create a ExprBuilder object and initialize it to x + 2*y
expr2 = ExprBuilder([x, y], [1, 2])
```

### ExprBuilder.getSize()

**Synopsis**

`getSize()`

**Description**

Retrieve the number of terms in an expression builder.

**Example**

```
# Retrieve the number of terms in expression builder 'expr'
exprsize = expr.getSize()
```

### ExprBuilder.getCoeff()

**Synopsis**

`getCoeff(idx)`

**Description**

Retrieve the coefficient of a variable by its index from an expression builder.

**Arguments**

`idx`

Index of the variable in the expression builder, starting with 0.

**Example**

```
# Retrieve the coefficient for the term at index 1 from expression builder 'expr'
coeff = expr.getCoeff(1)
```

### ExprBuilder.getVar()

**Synopsis**

`getVar(idx)`

**Description**

Retrieve the variable by its index from an expression builder. Return a *Var Class* object.

**Arguments**

`idx`

Index of the variable in the expression builder, starting with 0.

**Example**

```
# Retrieve the variable for the term at index 1 from expression builder 'expr'
x = expr.getVar(1)
```

### ExprBuilder.getConstant()

**Synopsis**

```
getConstant()
```

**Description**

Retrieve the constant term from an expression builder.

**Example**

```
# Retrieve the constant term from linear expression builder 'expr'
constant = expr.getConstant()
```

### ExprBuilder.addTerm()

**Synopsis**

```
addTerm(var, coeff=1.0)
```

**Description**

Add a new term to current expression builder.

**Arguments**

`var`

Variable to add.

`coeff`

Magnification coefficient for added term. Optional, 1.0 by default.

**Example**

```
# Add term 2*x to linear expression builder 'expr'
expr.addTerm(x, 2.0)
```

### ExprBuilder.addExpr()

**Synopsis**

```
addExpr(expr, mult=1.0)
```

**Description**

Add new expression builder to the current one.

**Arguments**

`expr`

Expression builder to add.

`mult`

Magnification coefficients for the added expression builder. Optional, 1.0 by default.

**Example**

```
# Add linear expression builder 2*x + 2*y to 'expr'
expr.addExpr(x + y, 2.0)
```

#### ExprBuilder.clone()

**Synopsis**

```
clone()
```

**Description**

Create a deep copy of the expression builder.

**Example**

```
# Create a deep copy of expression builder 'expr'
exprcopy = expr.clone()
```

#### ExprBuilder.getExpr()

**Synopsis**

```
getExpr()
```

**Description**

Create a linear expression related to the expression builder. Returns a *LinExpr Class* object.

**Example**

```
# Get the linear expression object related to expression builder 'exprbuilder'
expr = exprbuilder.getExpr()
```

### 24.2.61 LinExpr Class

LinExpr object contains operations related to variables for building linear constraints, and provides the following methods:

#### LinExpr()

**Synopsis**

```
LinExpr(arg1=0.0, arg2=None)
```

**Description**

Create a *LinExpr Class* object.

If argument `arg1` is constant, argument `arg2` is `None`, then create a *LinExpr Class* object and initialize it using argument `arg1`. If argument `arg1` is *Var Class* or *LinExpr Class* object, and argument `arg2` is constant or considered to be constant 1.0 when argument `arg2` is `None`, then initialize the newly created *LinExpr Class* object using arguments `arg1` and `arg2`. If argument `arg1` is list object and argument `arg2` is `None`, then argument `arg1` contains a list of variable-coefficient pairs and initialize the newly created *LinExpr Class* object using arguments `arg1` and `arg2`. For other forms of arguments, call method `addTerms` to initialize the newly created *LinExpr Class* object.

**Arguments**

`arg1`

Optional, 0.0 by default.

`arg2`

Optional, `None` by default.

**Example**

```
# Create a new LinExpr object and initialize it to 0.0
expr0 = LinExpr()
# Create a LinExpr object and initialize it to 2*x + 3*y
expr1 = LinExpr([(x, 2), (y, 3)])
# Create a LinExpr object and initialize it to x + 2*y
expr2 = LinExpr([x, y], [1, 2])
```

### LinExpr.setCoeff()

**Synopsis**

`setCoeff(idx, newval)`

**Description**

Set new coefficient value of a variable based on its index in an expression.

**Arguments**

`idx`

Index of the variable in the expression, starting with 0.

`newval`

New coefficient value of the variable.

**Example**

```
# Set the coefficient for the term at index 0 in expression expr to 1.0
expr.setCoeff(0, 1.0)
```

### LinExpr.getCoeff()

**Synopsis**

`getCoeff(idx)`

**Description**

Retrieve the coefficient of a variable by its index from an expression.

**Arguments**

`idx`

Index of the variable in the expression, starting with 0.

**Example**

```
# Retrieve the coefficient for the term at index 1 from expression expr
coeff = expr.getCoeff(1)
```

### LinExpr.getVar()

**Synopsis**

```
getVar(idx)
```

**Description**

Retrieve the variable by its index from an expression. Return a *Var Class* object.

**Arguments**

`idx`

Index of the variable in the expression, starting with 0.

**Example**

```
# Retrieve the variable for the term at index 1 from expression expr
x = expr.getVar(1)
```

### LinExpr.getConstant()

**Synopsis**

```
getConstant()
```

**Description**

Retrieve the constant term from an expression.

**Example**

```
# Retrieve the constant term from linear expression expr
constant = expr.getConstant()
```

### LinExpr.getValue()

**Synopsis**

```
getValue()
```

**Description**

Retrieve the value of an expression computed using the current solution.

**Example**

```
# Retrieve the value of expression expr for the current solution.
val = expr.getValue()
```

### LinExpr.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of terms in an expression.

**Example**

```
# Retrieve the number of terms in expression expr
exprsize = expr.getSize()
```

### LinExpr.setConstant()

**Synopsis**

`setConstant(newval)`

**Description**

Set the constant term of linear expression.

**Arguments**

`newval`

Constant term to be set.

**Example**

```
# Set constant term of linear expression 'expr' to 2.0
expr.setConstant(2.0)
```

### LinExpr.addConstant()

**Synopsis**

`addConstant(newval)`

**Description**

Add a constant to an expression.

**Arguments**

`newval`

Constant to add.

**Example**

```
# Add constant 2.0 to linear expression 'expr'
expr.addConstant(2.0)
```

### LinExpr.addTerm()

**Synopsis**

`addTerm(var, coeff=1.0)`

**Description**

Add a new term to current expression.

**Arguments**

`var`

Variable to add.

`coeff`

Magnification coefficient for added term. Optional, 1.0 by default.

**Example**

```
# Add term x to linear expression 'expr'
expr.addTerm(x)
```

### LinExpr.addTerms()

**Synopsis**

```
addTerms(vars, coeffs)
```

**Description**

Add a single term or multiple terms into an expression.

If argument `vars` is *Var Class* object, then argument `coeffs` is constant; If argument `vars` is *VarArray Class* object or list, then argument `coeffs` is constant or list; If argument `vars` is dictionary or *tupledict Class* object, then argument `coeffs` is constant, dict, or *tupledict Class* object.

**Arguments**

`vars`

Variables to add.

`coeffs`

Coefficients for variables.

**Example**

```
# Add term 2*x + 2*y to linear expression 'expr'
expr.addTerms([x, y], [2.0, 3.0])
```

### LinExpr.addExpr()

**Synopsis**

```
addExpr(expr, mult=1.0)
```

**Description**

Add new expression to the current one.

**Arguments**

`expr`

Expression or expression builder to add.

`mult`

Magnification coefficients for the added expression. Optional, 1.0 by default.

**Example**

```
# Add linear expression 2*x + 2*y to 'expr'
expr.addExpr(x + y, 2.0)
```

### LinExpr.clone()

**Synopsis**

```
clone()
```

**Description**

Create a deep copy of the expression.

**Example**

```
# Create a deep copy of expression expr
exprcopy = expr.clone()
```

### LinExpr.reserve()

**Synopsis**

```
reserve(n)
```

**Description**

Pre-allocate space for linear expression object.

**Arguments**

`n`

Number of terms to be allocated.

**Example**

```
# Allocate 100 terms for linear expression 'expr'
expr.reserve(100)
```

### LinExpr.remove()

**Synopsis**

```
remove(item)
```

**Description**

Remove a term from a linear expression.

If argument `item` is constant, then remove the term stored at index i of the expression; otherwise argument `item` is *Var Class* object.

**Arguments**

`item`

Constant index or variable of the term to be removed.

**Example**

```
# Remove the term whose index is 2 from linear expression expr
expr.remove(2)
# Remove the term whose variable is x from linear expression expr
expr.remove(x)
```

### 24.2.62 QuadExpr Class

QuadExpr object contains operations related to variables for building quadratic constraints, and provides the following methods:

#### QuadExpr()

**Synopsis**

```
QuadExpr(expr=0.0)
```

**Description**

Create a *QuadExpr Class* object.

Argument `expr` is constant, *Var Class*, *LinExpr Class* object or *QuadExpr Class* object.

**Arguments**

`expr`

Optional, 0.0 by default.

**Example**

```
# Create a new QuadExpr object and initialize it to 0.0
quadexpr0 = QuadExpr()
# Create a QuadExpr object and initialize it to 2*x + 3*y
quadexpr1 = QuadExpr([(x, 2), (y, 3)])
# Create a QuadExpr object and initialize it to x*x + 2*x*y
quadexpr2 = QuadExpr(x*x + 2*x*y)
```

#### QuadExpr.setCoeff()

**Synopsis**

```
setCoeff(idx, newval)
```

**Description**

Set new coefficient value of a term based on its index in a quadratic expression.

**Arguments**

`idx`

Index of the term in the quadratic expression, starting with 0.

`newval`

New coefficient value of the term.

**Example**

```
# Set the coefficient for the term at index 0 in quadratic expression quadexpr to 1.0
quadexpr.setCoeff(0, 1.0)
```

### QuadExpr.getCoeff()

**Synopsis**

```
getCoeff(idx)
```

**Description**

Retrieve the coefficient of a term by its index from a quadratic expression.

**Arguments**

`idx`

Index of the term in the quadratic expression, starting with 0.

**Example**

```
# Retrieve the coefficient for the term at index 1 from quadratic expression quadexpr
coeff = quadexpr.getCoeff(1)
```

### QuadExpr.getVar1()

**Synopsis**

```
getVar1(idx)
```

**Description**

Retrieve the first variable of a quadratic term by its index from an expression. Return a *Var Class* object.

**Arguments**

`idx`

Index of the quadratic term in the expression, starting with 0.

**Example**

```
# Retrieve the first variable of a quadratic term at index 1 from quadratic␣
↪expression quadexpr
x = quadexpr.getVar1(1)
```

### QuadExpr.getVar2()

**Synopsis**

```
getVar2(idx)
```

**Description**

Retrieve the second variable of a quadratic term by its index from an expression. Return a *Var Class* object.

**Arguments**

`idx`

Index of the quadratic term in the expression, starting with 0.

**Example**

```
# Retrieve the first variable of a quadratic term at index 1 from quadratic␣
↪expression quadexpr
y = quadexpr.getVar2(1)
```

### QuadExpr.getLinExpr()

**Synopsis**

```
getLinExpr()
```

**Description**

Retrieve the linear terms (if exist) from quadratic expression. Return a *LinExpr Class* object.

**Example**

```
# Retrieve the linear terms from a quadratic expression quadexpr
linexpr = quadexpr.getLinExpr()
```

### QuadExpr.getConstant()

**Synopsis**

```
getConstant()
```

**Description**

Retrieve the constant term from a quadratic expression.

**Example**

```
# Retrieve the constant term from quadratic expression quadexpr
constant = quadexpr.getConstant()
```

### QuadExpr.getValue()

**Synopsis**

```
getValue()
```

**Description**

Retrieve the value of a quadratic expression computed using the current solution.

**Example**

```
# Retrieve the value of quadratic expression quadexpr for the current solution.
val = quadexpr.getValue()
```

### QuadExpr.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of terms in a quadratic expression.

**Example**

```
# Retrieve the number of terms in quadratic expression quadexpr
exprsize = quadexpr.getSize()
```

### QuadExpr.setConstant()

**Synopsis**

`setConstant(newval)`

**Description**

Set the constant term of quadratic expression.

**Arguments**

`newval`

Constant to set.

**Example**

```
# Set constant term of quadratic expression 'quadexpr' to 2.0
quadexpr.setConstant(2.0)
```

### QuadExpr.addConstant()

**Synopsis**

`addConstant(newval)`

**Description**

Add a constant to a quadratic expression.

**Arguments**

`newval`

Constant to add.

**Example**

```
# Add constant 2.0 to quadratic expression 'quadexpr'
quadexpr.addConstant(2.0)
```

### QuadExpr.addTerm()

**Synopsis**

`addTerm(coeff, var1, var2=None)`

**Description**

Add a new term to current quadratic expression.

**Arguments**

`coeff`

Magnification coefficient for added term. Optional, 1.0 by default.

`var1`

The first variable for added term.

`var2`

The second variable for added term, defaults to `None`, i.e. add a linear term.

**Example**

```
# Add term x to quadratic expression 'quadexpr'
quadexpr.addTerm(1.0, x)
```

### QuadExpr.addTerms()

**Synopsis**

```
addTerms(coeffs, vars1, vars2=None)
```

**Description**

Add a single term or multiple terms into a quadratic expression.

If argument `vars` is *Var Class* object, then argument `vars2` is *Var Class* object or `None`, argument `coeffs` is constant; If argument `vars` is *VarArray Class* object or list, then argument `vars2` is *VarArray Class* object, list or `None`, argument `coeffs` is constant or list; If argument `vars` is dictionary or *tupledict Class* object, then argument `vars2` is dictionary, *tupledict Class* object or `None`, argument `coeffs` is constant, dictionary, or *tupledict Class* object.

**Arguments**

`coeffs`

Coefficients for terms.

`vars1`

The first variable of each term.

`vars2`

The second variable of each term, defaults to `None`, i.e. add a linear term.

**Example**

```
# Add term 2*x + 3y + 2*x*x + 3*x*y to quadratic expression 'quadexpr'
# Note: Mixed format is supported by addTerms yet.
quadexpr.addTerms([2.0, 3.0], [x, y])
quadexpr.addTerms([2.0, 3.0], [x, x], [x, y])
```

### QuadExpr.addLinExpr()

**Synopsis**

```
addLinExpr(expr, mult=1.0)
```

**Description**

Add new linear expression to the current quadratic expression.

**Arguments**

`expr`

Linear expression or linear expression builder to add.

`mult`

Magnification coefficient for the added expression. Optional, 1.0 by default.

**Example**

```
# Add linear expression 2*x + 2*y to 'quadexpr'
quadexpr.addLinExpr(x + y, 2.0)
```

### QuadExpr.addQuadExpr()

**Synopsis**

```
addQuadExpr(expr, mult=1.0)
```

**Description**

Add new quadratic expression to the current one.

**Arguments**

`expr`

Expression or expression builder to add.

`mult`

Magnification coefficients for the added expression. Optional, 1.0 by default.

**Example**

```
# Add quadratic expression x*x + 2*y to 'quadexpr'
quadexpr.addQuadExpr(x*x + 2*y, 2.0)
```

### QuadExpr.clone()

**Synopsis**

```
clone()
```

**Description**

Create a deep copy of the expression.

**Example**

```
# Create a deep copy of quadratic expression quadexpr
exprcopy = quadexpr.clone()
```

### QuadExpr.reserve()

**Synopsis**

```
reserve(n)
```

**Description**

Pre-allocate space for quadratic expression object.

**Arguments**

`n`

Number of terms to be allocated.

**Example**

```
# Allocate 100 terms for quadratic expression 'expr'
expr.reserve(100)
```

#### QuadExpr.remove()

**Synopsis**

```
remove(item)
```

**Description**

Remove a term from a quadratic expression.

If argument `item` is constant, then remove the term stored at index i of the expression; otherwise argument `item` is *Var Class* object.

**Arguments**

`item`

Constant index or variable of the term to be removed.

**Example**

```
# Remove the term whose index is 2 from quadratic expression quadexpr
quadexpr.remove(2)
# Remove the terms one of which variable is x from quadratic expression quadexpr
quadexpr.remove(x)
```

### 24.2.63 PsdExpr Class

PsdExpr object contains operations related to variables for building positive semi-definite constraints, and provides the following methods:

#### PsdExpr()

**Synopsis**

```
PsdExpr(expr=0.0)
```

**Description**

Create a *PsdExpr Class* object.

**Arguments**

`expr`

Optional, 0.0 by default, which can be a constant, *Var Class*, *LinExpr Class* object or *PsdExpr Class* object.

**Example**

```
# Create a new PsdExpr object and initialize it to 0.0
expr0 = PsdExpr()
# Create a PsdExpr object and initialize it to 2*x + 3*y
expr1 = PsdExpr(2*x + 3*y)
```

### PsdExpr.setCoeff()

**Synopsis**

```
setCoeff(idx, mat)
```

**Description**

Set the coefficient symmetric matrix corresponding to the specified index value `idx` in the LMI expression.

**Arguments**

`idx`

Index of the positive semi-definite variable in the expression, starting with 0.

`mat`

New symmetric matrix coefficient of the positive semi-definite variable.

**Example**

```
# Set symmetric matrix for the positive semi-definite variable at index 0 in␣
↪expression "expr" to mat
expr.setCoeff(0, mat)
```

### PsdExpr.getCoeff()

**Synopsis**

```
getCoeff(idx)
```

**Description**

Retrieve the symmetric matrix coefficient of a positive semi-definite variable by its index from the expression.

**Arguments**

`idx`

Index of the positive semi-definite variable in the expression, starting with 0.

**Example**

```
# Retrieve the symmetric matrix coefficient for the positive semi-definite variable␣
↪at index 1 from expression expr
mat = expr.getCoeff(1)
```

### PsdExpr.getPsdVar()

**Synopsis**

```
getPsdVar(idx)
```

**Description**

Retrieve a positive semi-definite variable by its index from the expression. Return a *PsdVar Class* object.

**Arguments**

`idx`

Index of the positive semi-definite variable in the expression, starting with 0.

**Example**

```
# Retrieve the positive semi-definite variable at index 1 from expression expr
x = expr.getPsdVar(1)
```

### PsdExpr.getLinExpr()

**Synopsis**

```
getLinExpr()
```

**Description**

Retrieve the linear terms (if exist) from positive semi-definite expression. Return a *LinExpr Class* object.

**Example**

```
# Retrieve the linear terms from a positive semi-definite expression expr
linexpr = expr.getLinExpr()
```

### PsdExpr.getConstant()

**Synopsis**

```
getConstant()
```

**Description**

Retrieve the constant term from a positive semi-definite expression.

**Example**

```
# Retrieve the constant term from expression expr
constant = expr.getConstant()
```

### PsdExpr.getValue()

**Synopsis**

```
getValue()
```

**Description**

Retrieve the value of a positive semi-definite expression computed using the current solution.

**Example**

```
# Retrieve the value of positive semi-definite expression expr for the current␣
↪solution.
val = expr.getValue()
```

### PsdExpr.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of terms in a positive semi-definite expression.

**Example**

```
# Retrieve the number of terms in expression expr
exprsize = expr.getSize()
```

### PsdExpr.setConstant()

**Synopsis**

```
setConstant(newval)
```

**Description**

Set the constant term of positive semi-definite expression.

**Arguments**

`newval`

Constant to set.

**Example**

```
# Set constant term of expression 'expr' to 2.0
expr.setConstant(2.0)
```

### PsdExpr.addConstant()

**Synopsis**

```
addConstant(newval)
```

**Description**

Add a constant to a positive semi-definite expression.

**Arguments**

`newval`

Constant to add.

**Example**

```
# Add constant 2.0 to expression 'expr'
expr.addConstant(2.0)
```

### PsdExpr.addTerm()

**Synopsis**

```
addTerm(var, mat)
```

**Description**

Add a new term to current positive semi-definite expression.

**Arguments**

`var`

The positive semi-definite variable to add.

`mat`

The symmetric matrix coefficient for the positive semi-definite variable.

**Example**

```
# Add positive semi-definite term C1 * X to expression 'expr'
expr.addTerm(X, C1)
```

### PsdExpr.addTerms()

**Synopsis**

```
addTerms(vars, mats)
```

**Description**

Add a single term or multiple positive semi-definite terms into a positive semi-definite expression.

If argument `vars` is *PsdVar Class* object, then argument `mats` is *SymMatrix Class* object; If argument `vars` is *PsdVarArray Class* object or list, then argument `mats` is *SymMatrixArray Class* or list;

**Arguments**

`vars`

The positive semi-definite variables to add.

`mats`

The symmetric matrices of the positive semi-definite terms.

**Example**

```
# Add terms C1 * X1 + C2 * X2 to expression 'expr'
expr.addTerms([X1, X2], [C1, C2])
```

### PsdExpr.addLinExpr()

**Synopsis**

```
addLinExpr(expr, mult=1.0)
```

**Description**

Add new linear expression to the current positive semi-definite expression.

**Arguments**

`expr`

Linear expression or linear expression builder to add.

`mult`

Magnification coefficient for the added expression. Optional, 1.0 by default.

**Example**

```
# Add linear expression 2*x + 2*y to 'expr'
expr.addLinExpr(x + y, 2.0)
```

### PsdExpr.addPsdExpr()

**Synopsis**

```
addPsdExpr(expr, mult=1.0)
```

**Description**

Add new positive semi-definite expression to the current one.

**Arguments**

`expr`

Positive semi-definite expression or positive semi-definite expression builder to add.

`mult`

Magnification coefficient for the added positive semi-definite expression. Optional, 1.0 by default.

**Example**

```
# Add positive semi-definite expression C * X to 'expr'
expr.addPsdExpr(C*X)
```

### PsdExpr.addMExpr()

**Synopsis**

```
addMExpr(expr, mult=1.0)
```

**Description**

Add a new multi-dimensional expression to the current PSD expression.

**Arguments**

`expr`

The multi-dimensional array expression or expression builder object to add.

`mult`

The scaling factor for the multi-dimensional array expression. Optional, defaulting to 1.0.

**Example**

```
# Add a multi-dimensional linear expression: 2.0 * A @ x to the PSD expression expr
expr.addMExpr(A @ x, 2.0)
```

### PsdExpr.clone()

**Synopsis**

`clone()`

**Description**

Create a deep copy of the expression.

**Example**

```
# Create a deep copy of expression expr
exprcopy = expr.clone()
```

### PsdExpr.reserve()

**Synopsis**

`reserve(n)`

**Description**

Pre-allocate space for positive semi-definite expression object.

**Arguments**

`n`

Number of terms to be allocated.

**Example**

```
# Allocate 100 terms for positive semi-definite expression 'expr'
expr.reserve(100)
```

### PsdExpr.remove()

**Synopsis**

`remove(item)`

**Description**

Remove a term from a positive semi-definite expression.

If argument `item` is constant, then remove the term stored at index i of the expression; otherwise argument `item` is *PsdVar Class* object.

**Arguments**

`item`

Constant index or *PsdVar Class* variable of the term to be removed.

**Example**

```
# Remove the term whose index is 2 from positive semi-definite expression expr
expr.remove(2)
# Remove the terms one of which variable is x from positive semi-definite expression␣
↪expr
expr.remove(x)
```

### 24.2.64 MPsdExpr Class

The *MPsdExpr* class in COPT is used for operations that combine PSD variables into multi-dimensional PSD expressions. The following methods are provided:

#### MPsdExpr.addTerm()

**Synopsis**

```
addTerm(var, mat)
```

**Description**

Adds a new PSD term to the current multi-dimensional PSD expression.

**Arguments**

`var`

The semidefinite variable in the term to be added.

`mat`

The symmetric matrix in the term to be added.

**Example**

```
# Add the PSD term C1 * X to mexpr
mexpr.addTerm(X, C1)
```

#### MPsdExpr.addTerms()

**Synopsis**

```
addTerms(vars, coeffs)
```

**Description**

Adds new terms to the multi-dimensional PSD expression object.

**Arguments**

`vars`

Multi-dimensional array variable objects. Possible value could be *MVar*.

`coeffs`

Coefficient matrices for the terms. Possible values could be a floating number or *NdArray*.

**Example**

```
# Add terms: mA @ mX to mpsdexpr
mX = model.addMVar((3,3), nameprefix="M_X")
mA = cp.NdArray(np.ones(shape=(3,3)))
mpsdexpr.addTerms(mX, mA)
```

### MPsdExpr.addLinExpr()

**Synopsis**

```
addLinExpr(expr, mult=1.0)
```

**Description**

Adds a new linear expression to the current multi-dimensional PSD expression.

**Arguments**

`expr`

The linear expression or expression builder object to add. Possible values could be *LinExpr* or *ExprBuilder*.

`mult`

The scaling factor for the linear expression. Optional, defaulting to 1.0.

**Example**

```
# Add a linear expression: 2*x + 2*y to mexpr
mexpr.addLinExpr(x + y, 2.0)
```

### MPsdExpr.addPsdExpr()

**Synopsis**

```
addPsdExpr(expr, mult=1.0)
```

**Description**

Adds a new semidefinite expression to the current multi-dimensional PSD expression.

**Arguments**

`expr`

The semidefinite expression to add.

`mult`

The scaling factor for the PSD expression. Optional, defaulting to 1.0.

**Example**

```
# Add a PSD expression: C * X to mexpr
mexpr.addPsdExpr(C * X)
```

### MPsdExpr.addMExpr()

**Synopsis**

```
addMExpr(expr, mult=1.0)
```

**Description**

Add a new multi-dimensional expression to the current multi-dimensional PSD expression.

**Arguments**

`expr`

The multi-dimensional expression or expression builder object to add.

`mult`

The scaling factor for the multi-dimensional expression. Optional, defaulting to 1.0.

**Example**

```
# Add a multi-dimensional linear expression: 2.0 * A @ x to mexpr
mexpr.addMExpr(A @ x, 2.0)
```

### MPsdExpr.addMLinExpr()

**Synopsis**

`addMLinExpr(exprs, mult=1.0)`

**Description**

Add corresponding linear expressions to each PSD expression in the `MPsdExpr` object.

**Arguments**

`exprs`

The multi-dimensional linear expressions or expression builder objects to add.

`mult`

The same scaling factor for all linear expressions to add. Optional, defaulting to 1.0.

**Example**

```
# Add a linear expression A @ x to each PSD expression in mexpr
mexpr.addMLinExpr(A @ x)
```

### MPsdExpr.addMPsdExpr()

**Synopsis**

`addMPsdExpr(exprs, mult=1.0)`

**Description**

Adds corresponding new PSD expressions to each PSD expression in the `MPsdExpr` object.

**Arguments**

`exprs`

The new semidefinite expressions to add.

`mult`

The same scaling factor for all PSD expressions to add. Optional, defaulting to 1.0.

**Example**

```
# Add a PSD expression: C * X to each PSD expression in mexpr
mexpr.addMPsdExpr(C * X)
```

### MPsdExpr.item()

#### Synopsis

```
item()
```

#### Description

Retrieves the *PsdExpr* in a 0-dimensional semidefinite expression. If the *MPsdExpr* object is not 0-dimensional, raises a *ValueError*.

#### Return value

Returns a *PsdExpr* object.

#### Example

```
barX = model.addPsdVars(3, "BAR_X")
mpsdexpr = barX[:-1, :-1]
psdexpr1 = mpsdexpr[0,0].item()
psdexpr2 = mpsdexpr.sum().item()
```

### MPsdExpr.sum()

#### Synopsis

```
sum(axis=None)
```

#### Description

Computes the sum of the semidefinite terms along the specified axis in the *MPsdExpr* object.

#### Arguments

`axis`

Optional. Defaults to *None*, which sums over all variables. Otherwise, sums along the specified axis.

#### Return value

Returns an *MPsdExpr Class* object representing the sum of the corresponding multi-dimensional PSD expressions.

### MPsdExpr.clear()

#### Synopsis

```
clear()
```

#### Description

Resets every element of the *MPsdExpr Class* object to 0.0.

#### Example

```
barX = model.addPsdVars(3, "BAR_X")
mpsdexpr = barX[:-1, :-1]
mpsdexpr.clear()
```

#### MPsdExpr.clone()

**Synopsis**

```
clone()
```

**Description**

Creates a deep copy of an *MPsdExpr Class* object.

**Return value**

Returns a new *MPsdExpr* object.

**Example**

```
barX = model.addPsdVars(3, "BAR_X")
mpsdexpr = barX[:-1, :-1]
newmpsdexpr = mpsdexpr.clone()
```

#### MPsdExpr.getValue()

**Synopsis**

```
getValue()
```

**Description**

Get the evaluation of each PSD expression within the *MPsdExpr Class* object after solving.

**Return value**

Returns a *NdArray Class* object of the same dimensions as the MPsdExpr object whose elements are the evaluations of the corresponding expression.

**Example**

```
barX = model.addPsdVars(3, "BAR_X")
mpsdexpr = barX[:-1, :-1]        # 2-D MPsdExpr, shape = (2, 2)
model.addConstrs(mpsdexpr <= 1.0)
model.solve()
print(mpsdexpr.getValue())
```

### 24.2.65 LmiExpr Class

LmiExpr object contains operations related to variables for building LMI constraints, and provides the following methods:

#### LmiExpr()

**Synopsis**

```
LmiExpr(arg1=None, arg2=None)
```

**Description**

Create a *LmiExpr Class* object.

**Arguments**

The default value of `arg1` is `None`, and the Possible values: *Var Class* object, or *SymMatrix Class* object.

If the argument `arg1` is a *Var Class* object, then the argument `arg2` is a *SymMatrix Class* object.

### LmiExpr.setCoeff()

**Synopsis**

```
setCoeff(idx, mat)
```

**Description**

Set the coefficient matrix for the entry corresponding to the specified index `idx` in the LMI expression.

**Arguments**

`idx`

The specified the index value. Starts with 0.

`mat`

The new coefficient symmetric matrix of the variable to be set, which must be a *SymMatrix Class* class object.

**Example**

```
# Set the coefficient of the 0-th term of the LMI expression expr to the symmetric␣
↪matrix mat
expr.setCoeff(0, mat)
```

### LmiExpr.getCoeff()

**Synopsis**

```
getCoeff(idx)
```

**Description**

Get the coefficient matrix for the entry corresponding to the specified index `idx` in the LMI expression.

**Arguments**

`idx`

The specified the index value. Starts with 0.

**Example**

```
# Get the symmetric matrix coefficient of the 1st term of the LMI expression expr
mat = expr.getCoeff(1)
```

### LmiExpr.getVar()

**Synopsis**

```
getVar(idx)
```

**Description**

Get the variable in the entry corresponding to the specified index `idx` in the LMI expression.

**Arguments**

`idx`

The specified the index value. Starts with 0.

**Example**

```
# Get the variable of the 1st item of the LMI expression expr
mat = expr.getVar(1)
```

### LmiExpr.getConstant()

**Synopsis**

```
getConstant()
```

**Description**

Get the constant-term symmetric matrix in the LMI expression.

**Example**

```
# Get the constant term of the LMI expression expr
constant = expr.getConstant()
```

### LmiExpr.getSize()

**Synopsis**

```
getSize()
```

**Description**

Retrieve the number of terms in the LMI expression.

**Example**

```
# Retrieve the number of terms in the LMI expression expr.
val = expr.getSize()
```

### LmiExpr.setConstant()

**Synopsis**

```
setConstant(mat)
```

**Description**

Set the constant-term symmetric matrix of the LMI expression.

**Arguments**

`mat`

The symmetric matrix corresponding to the constant item, must be a *SymMatrix Class* object.

**Example**

```
# Sets the constant term of the LMI expression expr to the symmetric matrix D1
expr.setConstant(D1)
```

### LmiExpr.addConstant()

**Synopsis**

```
addConstant(mat)
```

**Description**

Add the symmetric matrix to constant term of the LMI expression.

**Arguments**

`mat`

Matrix expression object added to constant term.

**Example**

```
# Add to the constant term of the LMI expression expr with symmetric matrix D2
expr.addConstant(D2)
```

### LmiExpr.addTerm()

**Synopsis**

```
addTerm(var, mat)
```

**Description**

Add a new term to current LMI expression.

**Arguments**

`var`

The variable in the new item.

`mat`

The symmetric matrix as variable coefficients in the new term.

**Example**

```
# Add the term C1 * X to expression 'expr'
expr.addTerm(x, C1)
```

### PsdExpr.addTerms()

**Synopsis**

```
addTerms(vars, mats)
```

**Description**

Adds multiple new terms to the LMI expression.

If argument `vars` is *PsdVar Class* object, then argument `mats` is *SymMatrix Class* object; If argument `vars` is *PsdVarArray Class* object or list, then argument `mats` is *SymMatrixArray Class* or list;

**Arguments**

`vars`

Array of variables to add new items to.

`mats`

The symmetric array of matrices to add new items to.

**Example**

```
# Add terms x1 * C1 + x2 * C2 to expression 'expr'
expr.addTerms([x1, x2], [C1, C2])
```

### LmiExpr.addLmiExpr()

**Synopsis**

```
addLmiExpr(expr, mult=1.0)
```

**Description**

Add a new LMI expression to the current LMI expression.

**Arguments**

`expr`

LMI expression to add.

`mult`

Magnification coefficient for the added LMI expression. Optional, 1.0 by default.

**Example**

```
# Add linear expression 2 * x * C to 'expr'
expr.addLinExpr(x * C, 2.0)
```

### LmiExpr.clone()

**Synopsis**

```
clone()
```

**Description**

Create a deep copy of the expression.

**Example**

```
# Create a deep copy of expression expr
exprcopy = expr.clone()
```

#### LmiExpr.reserve()

**Synopsis**

`reserve(n)`

**Description**

Pre-allocate space for LMI expression object.

**Arguments**

`n`

Number of terms to be allocated.

**Example**

```
# Allocate 100 terms for LMI expression 'expr'
expr.reserve(100)
```

#### LmiExpr.remove()

**Synopsis**

`remove(item)`

**Description**

Remove a term from the LMI expression.

If argument `item` is constant, then remove the term stored at index i of the expression; otherwise argument `item` is *Var Class* object.

**Arguments**

`item`

Constant index or *Var Class* variable of the term to be removed.

**Example**

```
# Remove the term whose index is 2 from positive semi-definite expression expr
expr.remove(2)
# Remove the terms one of which variable is x from LMI expression expr
expr.remove(x)
```

### 24.2.66 NlExpr Class

The `NlExpr` class provides interfaces for constructing nonlinear expressions in the COPT. It supports variable combination operations and offers the following member functions:

### NlExpr()

**Synopsis**

```
NlExpr(arg1=0.0, arg2=None)
```

**Description**

Creates a *NlExpr Class* object.

If `arg1` is a float and `arg2` is `None`, a constant nonlinear expression is created with value `arg1`.

If `arg1` is a *Var Class*, *LinExpr Class*, *QuadExpr Class*, or *ExprBuilder Class*, then `arg2` can be a float or `None`, indicating that the expression is scaled by `arg2` (default is 1.0).

**Arguments**

`arg1`

Optional argument. Default is 0.0.

Acceptable types include float, *Var Class*, *LinExpr Class*, *QuadExpr Class*, or *ExprBuilder Class*.

`arg2`

Coefficient of the expression.

Optional argument. Default is `None`, which is interpreted as 1.0.

**Example**

```
# Variable term with coefficient: 2 * x
expr1 = NlExpr(x, 2)
# Linear expression: 2 * x + 3 * y
lin = 2 * x + 3 * y
expr2 = NlExpr(lin)
```

### NlExpr.getConstant()

**Synopsis**

```
getConstant()
```

**Description**

Returns the constant term in the nonlinear expression.

**Example**

```
# Get the constant term of nexpr
constant = nexpr.getConstant()
```

### NlExpr.getSize()

**Synopsis**

```
getSize()
```

**Description**

Returns the number of terms in the nonlinear expression.

**Example**

```
# Get the number of elements in nexpr
exprsize = nexpr.getSize()
```

### NlExpr.getValue()

**Synopsis**

```
getValue()
```

**Description**

Returns the value of the nonlinear expression based on the current values of the variables.

**Example**

```
# Get the current value of the nonlinear expression nlexpr
val = nlexpr.getValue()
```

### NlExpr.setConstant()

**Synopsis**

```
setConstant(newval)
```

**Description**

Sets the constant term of the nonlinear expression.

**Arguments**

`newval`

The constant value to set.

**Example**

```
# Set constant of nexpr to 2.0
nexpr.setConstant(2.0)
```

### NlExpr.addConstant()

**Synopsis**

```
addConstant(newval)
```

**Description**

Adds a constant term to the nonlinear expression.

**Arguments**

`newval`

The constant value to add.

**Example**

```
# Add constant 2.0 to nexpr
nexpr.addConstant(2.0)
```

### NlExpr.addTerm()

**Synopsis**

```
addTerm(var, coeff=1.0)
```

**Description**

Adds a single linear term to the nonlinear expression.

**Arguments**

`var`

Variable in the new term.

`coeff`

Coefficient of the new term. Optional, default is 1.0.

**Example**

```
# Add term 2*x to nexpr
nexpr.addTerm(x, 2)
```

### NlExpr.addTerms()

**Synopsis**

```
addTerms(vars, coeffs)
```

**Description**

Adds one or more linear terms to the nonlinear expression.

If `vars` is a *Var Class* object, `coeffs` may be a single constant.

If `vars` is a *VarArray Class* or an iterable of *Var Class* objects, `coeffs` may be a single constant or a list of constants corresponding to each variable.

If `vars` is a mapping from arbitrary keys to variables, `coeffs` may be a single constant or a mapping with the same keys specifying individual coefficients.

**Arguments**

`vars`

Variables to add.

`coeffs`

Coefficients for the terms.

**Example**

```
# Add terms 2*x + 3*y to nexpr
nexpr.addTerms([x, y], [2.0, 3.0])
```

### NlExpr.addLinExpr()

**Synopsis**

```
addLinExpr(expr, mult=1.0)
```

**Description**

Adds a linear expression `expr` to the current nonlinear expression, scaling all terms by the given multiplier `mult`.

**Arguments**

`expr`

Linear expression to be added.

Must be a *ExprBuilder Class* or *LinExpr Class* object.

`mult`

Scaling factor applied to all terms in `expr`.

Optional argument. Default is 1.0.

**Example**

```
# Add a scaled linear expression to nexpr
lexpr = LinExpr(x, 2.0) + y * 3.0
nexpr = NlExpr()
nexpr.addLinExpr(lexpr)
nexpr.addLinExpr(lexpr, mult=0.5)
```

### NlExpr.addQuadExpr()

**Synopsis**

```
addQuadExpr(expr, mult=1.0)
```

**Description**

Adds a quadratic expression `expr` to the current nonlinear expression, with all terms scaled by `mult`.

**Arguments**

`expr`

Quadratic expression to add.

Must be a *QuadExpr Class* object.

`mult`

Scaling factor for all terms.

Optional. Default is 1.0.

### NlExpr.addNlExpr()

**Synopsis**

```
addNlExpr(expr, mult=1.0)
```

**Description**

Adds another nonlinear expression `expr` to the current expression, with all terms scaled by `mult`.

**Arguments**

`expr`

Nonlinear expression to add.

Must be a *NlExpr Class* object.

`mult`

Scaling factor.

Optional. Default is 1.0.

### NlExpr.clear()

**Synopsis**

```
clear()
```

**Description**

Clears all contents of the nonlinear expression.

### NlExpr.clone()

**Synopsis**

```
clone()
```

**Description**

Creates a deep copy of the current nonlinear expression.

**Return value**

A new *NlExpr Class* object.

### NlExpr.negate()

**Synopsis**

```
negate()
```

**Description**

Negates the current nonlinear expression by flipping the sign of each term.

#### NlExpr.reserve()

**Synopsis**

```
reserve(n)
```

**Description**

Preallocates space for `n` terms in the expression.

**Arguments**

`n`

Number of terms to reserve.

#### NlExpr.size()

**Synopsis**

```
size()
```

**Description**

Returns the number of nonlinear terms in the expression.

**Return value**

An integer.

### 24.2.67 nl Namespace

#### nl.abs()

**Synopsis**

```
NlExpr abs(expr)
```

**Description**

Calculate the absolute value of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.acos()

**Synopsis**

```
NlExpr acos(expr)
```

**Description**

Calculate the arc cosine of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.acosh()

**Synopsis**

```
NlExpr acosh(expr)
```

**Description**

Calculate the inverse hyperbolic cosine of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.asin()

**Synopsis**

```
NlExpr asin(expr)
```

**Description**

Calculate the arc sine of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.asinh()

**Synopsis**

```
NlExpr asinh(expr)
```

**Description**

Calculate the inverse hyperbolic sine of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.atan()

**Synopsis**

```
NlExpr atan(expr)
```

**Description**

Calculate the arc tangent of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.atan2()

**Synopsis**

```
NlExpr atan2(y, x)
```

**Description**

Calculate the arc tangent of `y/x`, considering the quadrant of the point (x, y).

**Arguments**

`y`

The numerator (y-coordinate) object.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

`x`

The denominator (x-coordinate) object.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.atanh()

**Synopsis**

```
NlExpr atanh(expr)
```

**Description**

Calculate the inverse hyperbolic tangent of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.ceil()

**Synopsis**

```
NlExpr ceil(expr)
```

**Description**

Calculate the ceiling of the input object, i.e. round it up to the nearest integer.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.cos()

**Synopsis**

```
NlExpr cos(expr)
```

**Description**

Calculate the cosine of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.cosh()

**Synopsis**

```
NlExpr cosh(expr)
```

**Description**

Calculate the hyperbolic cosine of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.exp()

**Synopsis**

```
NlExpr exp(expr)
```

**Description**

Calculate the natural exponential of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.floor()

**Synopsis**

```
NlExpr floor(expr)
```

**Description**

Calculate the floor of the input object, i.e. round it down to the nearest integer.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.log10()

**Synopsis**

```
NlExpr log10(expr)
```

**Description**

Calculate the base-10 logarithm of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.log()

**Synopsis**

```
NlExpr log(expr)
```

**Description**

Calculate the natural logarithm of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.neg()

**Synopsis**

```
NlExpr neg(expr)
```

**Description**

Calculate the negation of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.pow()

**Synopsis**

```
NlExpr pow(base, expo)
```

**Description**

Calculate the exponentiation of the base raised to the power of the exponent.

**Arguments**

`base`

The expression object to be used as the base.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

`expo`

The expression object to be used as the exponent.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.sin()

**Synopsis**

```
NlExpr sin(expr)
```

**Description**

Calculate the sine of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.sinh()

**Synopsis**

```
NlExpr sinh(expr)
```

**Description**

Calculate the hyperbolic sine of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.sqrt()

**Synopsis**

```
NlExpr sqrt(expr)
```

**Description**

Calculate the square root of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.sum()

**Synopsis**

```
NlExpr sum(op1, op2=None, op3=None, op4=None)
```

**Description**

Calculate the sum of nonlinear expression objects.

**Arguments**

`op1`

Required argument.

The expression object(s) to be summed.

Possible values include a *NlExpr Class* object or an iterable/mapping of *NlExpr Class* objects.

`op2`, `op3`, `op4`

Optional arguments.

Additional expression objects to be summed.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.tan()

**Synopsis**

```
NlExpr tan(expr)
```

**Description**

Calculate the tangent of the input object.

**Arguments**

`expr`

The object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### nl.tanh()

**Synopsis**

```
NlExpr tanh(expr)
```

**Description**

Calculate the hyperbolic tangent of the input object.

**Arguments**

`expr`

The expression object to be calculated.

Possible values include: constant, *Var Class* object, *LinExpr Class*, *QuadExpr Class*, *NlExpr Class* object, or *ExprBuilder Class* object.

**Return value**

Returns a *NlExpr Class* object.

### 24.2.68 CallbackBase Class

COPT CallbackBase class. This is an abstract class, the user needs to implement the function *CallbackBase.callback()* to create an instance. The instance is passed in as the first argument of the method *Model.setCallback()*

#### CallbackBase.where()

**Synopsis**

`where()`

**Description**

Get context in callback.

**Return value**

Returns an integer value.

#### CallbackBase.callback()

**Synopsis**

`callback()`

**Description**

The callback function is a pure virtual function which needs to be implemented by the user. The user can describe the information that needs to be obtained or the operation that needs to be performed during the solution process.

**Example**

```
class CoptCallback(CallbackBase):
    def __init__(self):
        super().__init__()
    def callback(self):
        # Get the objective value when finding a feasible MIP solution
        if self.where() == COPT.CBCONTEXT_MIPSOL:
            db = self.getInfo(COPT.CBInfo.MipCandObj)
```

#### CallbackBase.interrupt()

**Synopsis**

`interrupt()`

**Description**

Interrupt the callback process.

### CallbackBase.addUserCut()

**Synopsis**

```
addUserCut(lhs, sense = None, rhs = None)
```

**Description**

Add a user cut to the MIP model from within the callback function.

**Arguments**

`lhs`

Left-hand side expression for the new user cut. It can take the value of *Var Class* object, *LinExpr Class* object, or *ConstrBuilder Class* .

`sense`

The sense of the new user cut. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

Optional. None by default.

The user cut added from within callback can only have a single comparison operator.

`rhs`

Right-hand side expression for the new user cut.

Optional. None by default.

It can be a constant, or *Var Class* object, or *LinExpr Class* object.

**Example**

```
self.addUserCut(x+y <= 1)
```

### CallbackBase.addUserCuts()

**Synopsis**

```
addUserCuts(generator)
```

**Description**

Add a set of user cuts to the MIP model from within the callback function.

**Arguments**

`generator`

Array of builders for user cuts. It can be *ConstrBuilderArray Class* object or *MConstrBuilder Class* object.

**Example**

```
self.addUserCuts(x[i]+y[i] <= 1 for i in range(10))
```

### CallbackBase.addLazyConstr()

**Synopsis**

```
addLazyConstr(lhs, sense = None, rhs = None)
```

**Description**

Add a lazy constraint to the MIP model from within the callback function.

**Arguments**

`lhs`

Left-hand side expression for the new lazy constraint. It can take the value of *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object.

`sense`

The sense of the lazy constraint. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

Optional. None by default.

The lazy constraint added from within callback can only have a single comparison operator.

`rhs`

Right-hand side expression for the new lazy constraint.

Optional. None by default.

It can be a constant, or *Var Class* object, or *LinExpr Class* object.

**Example**

```
self.addLazyConstr(x+y <= 1)
```

### CallbackBase.addLazyConstrs()

**Synopsis**

```
addLazyConstrs(generator)
```

**Description**

Add a set of lazy constraints to the MIP model from within the callback function.

**Arguments**

`generator`

Array of builders for lazy constraints. It can be *ConstrBuilderArray Class* object or *MConstrBuilder Class* object.

**Example**

```
self.addLazyConstrs(x[i] + y[i] <= 1 for i in range(10))
```

### CallbackBase.getInfo()

**Synopsis**

`getInfo(cbinfo)`

**Description**

Retrieve the value of the specified callback information.

**Arguments**

`cbinfo`

The name of the callback information. Please refer to *Callback Information* for possible values.

**Return value**

Returns a constant(int-valued or double-valued).

**Example**

```
db = self.getInfo(COPT.CBInfo.BestBnd)
```

### CallbackBase.getRelaxSol()

**Synopsis**

`getRelaxSol(vars)`

**Description**

Retrieve the LP-relaxation solution of the specified variables at the current node.

Note that this method can only be invoked if `CallbackBase.where() == COPT.CBCONTEXT_MIPRELAX`.

**Arguments**

`vars`

The variables to retrieve the LP-relaxation solution values.

**Return value**

When parameter `vars` is *Var Class* object, it returns a constant, which is the LP-relaxation solution value of the specified variable.

When parameter `vars` is list or *VarArray Class* object, it returns a list of constants, consisting of the solution of the specified variables.

When parameter `args` is dictionary or *tupledict Class* object, it returns *tupledict Class* object(the indices of specified variables as key, the solutions of the specified variables as value).

When parameter `args` is `None` , it returns the LP-relaxation solution values of all variables.

**Example**

```
vals = self.getRelaxSol(vars)
```

### CallbackBase.getIncumbent()

**Synopsis**

```
getIncumbent(vars)
```

**Description**

Retrieve values from the best feasible solution of the specified variables.

**Arguments**

`vars`

The variables whose solution values to retrieve.

**Return value**

When parameter `vars` is *Var Class* object, it returns a constant, which is the solution value of the specified variable.

When parameter `vars` is list or *VarArray Class* object, it returns a list of constants, consisting of the solution of the specified variables.

When parameter `args` is dictionary or *tupledict Class* object, it returns *tupledict Class* object(the indices of specified variables as key, the solutions of the specified variables as value).

When parameter `args` is `None` , it returns the incumbent values of all variables.

**Example**

```
vals = self.getIncumbent(vars)
```

### CallbackBase.getSolution()

**Synopsis**

```
getSolution(vars)
```

**Description**

Retrieve values from the current solution of the specified variables.

Note that this method can only be invoked if `CallbackBase.where() == COPT.CBCONTEXT_MIPSOL`.

**Arguments**

`vars`

The variables whose solution values to retrieve.

**Return value**

When parameter `vars` is *Var Class* object, it returns a constant, which is the solution value of the specified variable.

When parameter `vars` is list or *VarArray Class* object, it returns a list of constants, consisting of the solution of the specified variables.

When parameter `args` is dictionary or *tupledict Class* object, it returns *tupledict Class* object(the indices of specified variables as key, the solutions of the specified variables as value).

When parameter `args` is `None` , it returns the solution values of all variables.

**Example**

```
vals = self.getSolution(vars)
```

#### CallbackBase.setSolution()

**Synopsis**

```
setSolution(vars, vals)
```

**Description**

Set feasible solution values for the specified variables.

When parameter `vars` is *Var Class* object, parameter `vals` is constant;

When parameter `vars` is dictionary or *tupledict Class* object, parameter `vals` can be constant, dictionary or *tupledict Class* object;

When parameter `vars` is list, *VarArray Class* object, parameter `vals` can be constant or list.

**Arguments**

`vars`

The variables to be set value.

`vals`

The values of the variables in the solution.

**Example**

```
self.setSolution(x, 1)
```

#### CallbackBase.loadSolution()

**Synopsis**

```
loadSolution()
```

**Description**

Load the currently feasible solution into model.

Note that a complete solution is required here.

**Example**

```
self.loadSolution()
```

### 24.2.69 NlpCallbackBase Class

The NlpCallbackBase class provides an interface for defining evaluation callbacks for nonlinear optimization models in COPT.

By inheriting this class and implementing the corresponding member methods, users provide objective and constraint value evaluations, together with their first-order and second-order derivative information to COPT during nonlinear optimization.

### NlpCallbackBase.EvalObj()

**Synopsis**

```
EvalObj(xdata, outdata)
```

**Description**

Evaluate the objective function value of the nonlinear model.

**Arguments**

`xdata`

A one-dimensional array containing the values of the decision variables.

Acceptable values: *NdArray Class*.

`outdata`

An output array used to store the objective function value (this array contains exactly one element).

Acceptable values: *NdArray Class*.

**Example**

```
class MyCallback(NlpCallbackBase):
    def EvalObj(self, xdata, outdata):
        x = NdArray(xdata)
        outval = NdArray(outdata)
        outval[0] = x[0] * x[0] + x[1] * x[1]
        return 0
```

### NlpCallbackBase.EvalGrad()

**Synopsis**

```
EvalGrad(xdata, outdata)
```

**Description**

Evaluate the gradient of the objective function of the nonlinear model.

**Arguments**

`xdata`

A one-dimensional array containing the current values of the decision variables.

Acceptable values: *NdArray Class*.

`outdata`

An output array used to store the objective function gradient. The length of this array equals the number of variables, and the element order corresponds to the variable order.

Acceptable values: *NdArray Class*.

**Return value**

Returns an integer status code. A value of `0` indicates successful evaluation.

**Example**

```
class MyCallback(NlpCallbackBase):
    def EvalGrad(self, xdata, outdata):
        x = NdArray(xdata)
        outval = NdArray(outdata)
        outval[0] = 2.0 * x[0]
        outval[1] = 2.0 * x[1]
        return 0
```

### NlpCallbackBase.EvalCon()

#### Synopsis

`EvalCon(xdata, outdata)`

#### Description

Evaluate the constraint function values of the nonlinear model.

#### Arguments

`xdata`

A one-dimensional array containing the values of the decision variables.

Acceptable values: *NdArray Class*.

`outdata`

An output array used to store the constraint function values. The length of this array equals the number of constraints, and the element order corresponds to the constraint definition order in the model.

Acceptable values: *NdArray Class*.

#### Example

```
class MyCallback(NlpCallbackBase):
    def EvalCon(self, xdata, outdata):
        x = NdArray(xdata)
        outval = NdArray(outdata)
        outval[0] = x[0] * x[0] + x[1] - 1.0
        return 0
```

### NlpCallbackBase.EvalJac()

#### Synopsis

`EvalJac(xdata, outdata)`

#### Description

Evaluate the Jacobian matrix values of the nonlinear constraint functions.

#### Arguments

`xdata`

A one-dimensional array containing the values of the decision variables.

Acceptable values: *NdArray Class*.

`outdata`

An output array used to store the nonzero elements of the constraint Jacobian matrix.

Acceptable values: *NdArray Class*.

The length of this array equals the number of nonzero elements in the Jacobian matrix. Each element corresponds to the position specified by `idxJacRow` and `idxJacCol` provided in `Model.loadNlData`.

**Example**

```
class MyCallback(NlpCallbackBase):
    def EvalJac(self, xdata, outdata):
        x = NdArray(xdata)
        outval = NdArray(outdata)
        outval[0] = 2.0 * x[0]
        outval[1] = 1.0
        return 0
```

### NlpCallbackBase.EvalHess()

**Synopsis**

```
EvalHess(xdata, sigma, lambdata, outdata)
```

**Description**

Evaluate the Hessian matrix values of the Lagrangian function of the nonlinear model.

**Arguments**

`xdata`

A one-dimensional array containing the values of the decision variables.

Acceptable values: *NdArray Class*.

`sigma`

The weight coefficient of the objective in the Lagrangian function.

`lambdata`

A one-dimensional array containing the current Lagrange multiplier values associated with the constraint functions.

Acceptable values: *NdArray Class*.

`outdata`

An output array used to store the nonzero elements of the Hessian matrix of the Lagrangian function.

Acceptable values: *NdArray Class*.

The length of this array equals the number of nonzero elements in the Hessian matrix. Each element corresponds to the position specified by `idxHessRow` and `idxHessCol` provided in `Model.loadNlData`.

**Example**

```
class MyCallback(NlpCallbackBase):
    def EvalHess(self, xdata, sigma, lambdata, outdata):
        x = NdArray(xdata)
        lam = NdArray(lambdata)
```

(continues on next page)

(continued from previous page)

```
        outval = NdArray(outdata)
        outval[0] = 2.0 * sigma + 2.0 * lam[0]
        return 0
```

### 24.2.70 GenConstrX Class

In the `Model` class, the constraints added by `addGenConstrXXX` (such as: `addGenConstrMax`) will return a `GenConstrX` object.

#### GenConstrX.getAttr()

**Synopsis**

`getAttr(attrname)`

**Description**

Get the attribute value of the `GenConstrX` class object, support to get the type and name of the `GenConstrX` class object.

**Example**

```
# Get the name of con_max
con_max.getAttr("name")
# Get the type of con_max
con_max.getAttr("type")
```

#### GenConstrX.setAttr()

**Synopsis**

`setAttr(attrname, newvalue)`

**Description**

Set the attribute value of the `GenConstrX` class object, and support setting the name of the `GenConstrX` class object.

**Arguments**

`attrname`

The name of the attribute to set, e.g. `"name"` .

`newvalue`

The new value of the attribute.

**Example**

```
# Set the name of con_max
con_max.setAttr("name", "con_max")
```

### 24.2.71 CoptError Class

CoptError Class provides operations on error. An exception of the CoptError is thrown when error occurs in a method call corresponding to the underlying interface of solver. The following attributes are provided to retrieve error information:

- CoptError.retcode

  Error code.

- CoptError.message

  Error message.

## 24.3 Helper Functions and Utilities

Helper functions and utilities are encapsulated based on Python's basic data types, providing easy-to-use data types to facilitate the rapid construction of complex optimization models. This section will explain its functions and usages.

### 24.3.1 Helper Functions

#### multidict()

**Synopsis**

`multidict(data)`

**Description**

Split a single dictionary into keys and multiple dictionaries. Return keys and dictionaries.

**Arguments**

`data`

A Python dictionary to be applied. Each key should map to a list of $n$ values.

**Example**

```
keys, dict1, dict2 = multidict({
  "hello": [0, 1],
  "world": [2, 3]})
```

#### quicksum()

**Synopsis**

`quicksum(data)`

**Description**

Build expressions efficiently. Return a *LinExpr Class* object.

**Arguments**

`data`

Terms to add.

**Example**

```
expr = quicksum(m.getVars())
```

### 24.3.2 tuplelist Class

The tuplelist object is an encapsulation based on Python lists, and provides the following methods:

#### tuplelist()

**Synopsis**

`tuplelist(list)`

**Description**

Create and return a *tuplelist Class* object.

**Arguments**

`list`

A Python list.

**Example**

```
tl = tuplelist([(0, 1), (1, 2)])
tl = tuplelist([('a', 'b'), ('b', 'c')])
```

#### tuplelist.add()

**Synopsis**

`add(key)`

**Description**

Add an item to a *tuplelist Class* object

**Arguments**

`key`

Item to add, which can be a Python tuple.

**Example**

```
tl = tuplelist([(0, 1), (1, 2)])
tl.add((2, 3))
```

#### tuplelist.select()

**Synopsis**

`select(pattern)`

**Description**

Get all terms that match the specified pattern. Return a *tuplelist Class* object.

**Arguments**

`pattern`

Specified pattern.

**Example**

```
tl = tuplelist([(0, 1), (0, 2), (1, 2)])
tl.select(0, '*')
```

### repeat()

**Synopsis**

`repeat(obj, repeats)`

**Description**

Repeats the specified object a given number of times.

**Arguments**

`obj`

A Python object.

`repeats`

The number of repetitions.

### hstack()

**Synopsis**

`hstack(left, right)`

**Description**

Horizontally stacks the input objects.

**Arguments**

`left`

A Python object.

`right`

A Python object.

### vstack()

**Synopsis**

`vstack(left, right)`

**Description**

Vertically stacks the input objects.

**Arguments**

`left`

A Python object.

`right`

A Python object.

#### stack()

**Synopsis**

```
stack(left, right, axis)
```

**Description**

Stacks the input objects along the specified axis.

**Arguments**

`left`

A Python object.

`right`

A Python object.

`axis`

The specified axis index.

### 24.3.3 tupledict Class

The tupledict class is an encapsulation based on Python dictionaries, and provides the following methods:

#### tupledict()

**Synopsis**

```
tupledict(args, kwargs)
```

**Description**

Create and return a *tupledict Class* object.

**Arguments**

`args`

Positional arguments.

`kwargs`

Named arguments.

**Example**

```
d = tupledict([(0, "hello"), (1, "world")])
```

#### tupledict.select()

**Synopsis**

```
select(pattern)
```

**Description**

Get all terms that match the specified pattern. Return a *tupledict Class* object.

**Arguments**

`pattern`

Specified pattern.

**Example**

```
d = tupledict([(0, "hello"), (1, "world")])
d.select()
```

#### tupledict.sum()

**Synopsis**

`sum(pattern)`

**Description**

Sum all terms that match the specified pattern. Return a *LinExpr Class* object.

**Arguments**

`pattern`

Specified pattern.

**Example**

```
expr = x.sum()
```

#### tupledict.prod()

**Synopsis**

`prod(coeff, pattern)`

**Description**

Filter terms that match the specified pattern and multiply by coefficients. Return a *LinExpr Class* object.

**Arguments**

`coeff`

Coefficients, which can be a dict or a *tupledict Class* object.

`pattern`

Specified pattern.

**Example**

```
coeff = dict([(1, 0.1), (2, 0.2)])
expr  = x.prod(coeff)
```

### 24.3.4 ProbBuffer Class

The ProbBuffer is an encapsulation of buffer of string stream, and provides the following methods:

### ProbBuffer()

**Synopsis**

`ProbBuffer(buff)`

**Description**

Create and return a *ProbBuffer Class* object.

**Arguments**

`buff`

Size of buffer, defaults to `None`, i.e. the buffer size is 0.

**Example**

```
# Create a buffer of size 100
buff = ProbBuffer(100)
```

### ProbBuffer.getData()

**Synopsis**

`getData()`

**Description**

Get the contents of buffer.

**Example**

```
# Print the contents in buffer
print(buff.getData())
```

### ProbBuffer.getSize()

**Synopsis**

`getSize()`

**Description**

Get the size of the buffer.

**Example**

```
# Get the size of the buffer
print(buff.getSize())
```

### ProbBuffer.resize()

**Synopsis**

`resize(sz)`

**Description**

Resize the size of the buffer.

**Arguments**

`sz`

New size of buffer.

**Example**

```
# Resize the size of buffer to 100
buff.resize(100)
```

# Chapter 25

# C++ API Reference

The **Cardinal Optimizer** provides C++ API library. This chapter documents all COPT constants, including parameters and attributes, and API functions for C++ applications.

## 25.1 Constants

All C++ constants are the same as C constants. Please refer to *C API Reference: Constants* for more details.

## 25.2 Attributes

All C++ attributes are the same as C attributes. Please refer to *C API Reference: Attributes* for more details.

In the C++ API, user can get the attribute value by specifying the attribute name. The provided functions are as follows, please refer to *C++ API: Model Class* for details.

- `Model::GetIntAttr()` : Get value of a COPT integer attribute.
- `Model::GetDblAttr()` : Get value of a COPT double attribute.

## 25.3 Information

All C++ information is the same as C information. Please refer to *C API Reference: Information* .

In the C++ API, user can get or set the information value by specifying the information name. The functions provided are as follows, please refer to *C++ Model class* for details. Information of variables or constraints can also be obtained/set through the `Get()/Set()` function of themselves.

- `Model::Get()` : Get the value of information related to the variables or constraints.
- `Model::Set()` : Set the value of information related to the variables or constraints.

## 25.4 Parameters

All C++ parameters are the same as C parameters. Please refer to *C API Reference: Parameters* for more details.

In the C++ API, user can get and set the parameter value by specifying the parameter name. The provided functions are as follows, please refer to *C++ Model class* for details.

- Get detailed information of the specified parameter (current value/max/min): `Model::GetParamInfo()`
- Get the current value of the specified integer/double parameter: `Model::GetIntParam()` / `Model::GetDblParam()`
- Set the specified integer/double parameter value: `Model::SetIntParam()` / `Model::SetDblParam()`

## 25.5 C++ Modeling Classes

This chapter documents COPT C++ interface. Users may refer to C++ classes described below for details of how to construct and solve C++ models.

### 25.5.1 Envr

Essentially, any C++ application using Cardinal Optimizer should start with a COPT environment. COPT models are always associated with a COPT environment. User must create an environment object before populating models. User generally only need a single environment object in program.

#### Envr::Envr()

Constructor of COPT Envr object.

**Synopsis**

```
Envr()
```

#### Envr::Envr()

Constructor of COPT Envr object, given a license folder.

**Synopsis**

```
Envr(const char *szLicDir)
```

**Arguments**

`szLicDir`: directory having local license or client config file.

### Envr::Envr()

Constructor of COPT Envr object, given an Envr config object.

**Synopsis**

```
Envr(const EnvrConfig &config)
```

**Arguments**

`config`: Envr config object holding settings for remote connection.

### Envr::BindNumaCpu()

Bind the CPUs for the current process to a NUMA node.

**Synopsis**

```
void BindNumaCpu(int numaNode)
```

**Arguments**

`numaNode`: ID of a NUMA node.

### Envr::BindNumaMem()

Bind memory for the current process to a NUMA node (Linux only).

**Synopsis**

```
void BindNumaMem(int numaNode)
```

**Arguments**

`numaNode`: the ID of a NUMA node.

### Envr::Close()

Close remote connection and token becomes invalid for all problems in current envr.

**Synopsis**

```
void Close()
```

### Envr::CreateModel()

Create a COPT model object.

**Synopsis**

```
Model CreateModel(const char *szName)
```

**Arguments**

`szName`: customized model name.

**Return**

a COPT model object.

#### Envr::GetCpuAffinity()

Get CPU affinity for the current process, which is saved in an integer array.

**Synopsis**

```
int GetCpuAffinity(int *cpuList, int len)
```

**Arguments**

`cpuList`: a list of CPU IDs.

`len`: length of the CPU list.

**Return**

actual size of binding CPUs.

#### Envr::GetNumaNodeCount()

Get count of NUMA nodes.

**Synopsis**

```
int GetNumaNodeCount()
```

**Return**

count of NUMA nodes.

#### Envr::SetCpuAffinity()

Set CPU affinity with given mask string.

**Synopsis**

```
void SetCpuAffinity(const char *hexMask)
```

**Arguments**

`hexMask`: CPU mask string of hexadecimal characters.

### 25.5.2 EnvrConfig

If user connects to COPT remote services, such as floating token server or compute cluster, it is necessary to add config settings with EnvrConfig object.

#### EnvrConfig::EnvrConfig()

Constructor of COPT environment config object.

**Synopsis**

```
EnvrConfig()
```

#### EnvrConfig::Set()

Set config settings in terms of name-value pair.

**Synopsis**

```
void Set(const char *szName, const char *szValue)
```

**Arguments**

`szName`: keyword of a config setting

`szValue`: value of a config setting

### 25.5.3 Model

In general, a COPT model consists of a set of variables, a (linear) objective function on these variables, a set of constraints on these variables, etc. COPT model class encapsulates all required methods for constructing a COPT model.

#### Model::AddAffineCone()

Add an affine cone constraint to a model.

**Synopsis**

```
AffineCone AddAffineCone(
    const MPsdExpr<1> &exprs,
    int type,
    const char *szName)
```

**Arguments**

`exprs`: 1-dimensional array of PSD expressions.

`type`: type of an affine cone.

`szName`: name of new affine cone constraint.

**Return**

new affine cone constraint object.

#### Model::AddAffineCone()

Add an affine cone constraint to model.

**Synopsis**

```
AffineCone AddAffineCone(const AffineConeBuilder &builder, const
char *szName)
```

**Arguments**

`builder`: builder for new affine cone constraint.

`szName`: optional, name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model::AddAffineCone()

Add an affine cone constraint to a model.

**Synopsis**

```
AffineCone AddAffineCone(
    const MLinExpr<1> &exprs,
    int type,
    const char *szName)
```

**Arguments**

`exprs`: 1-dimensional array of linear expressions.

`type`: type of an affine cone.

`szName`: name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model::AddAffineCones()

Add a list of affine cone constraints to a model.

**Synopsis**

```
AffineConeArray AddAffineCones(
    const MPsdExpr<2> &exprs,
    int type,
    const char *szPrefix)
```

**Arguments**

`exprs`: 2-dimensional array of PSD expressions.

`type`: type of an affine cone.

`szPrefix`: name prefix for new affine cone constraints.

**Return**

array of new affine cone constraint objects.

### Model::AddAffineCones()

Add a list of affine cone constraints to a model.

**Synopsis**

```
AffineConeArray AddAffineCones(
    const MLinExpr<2> &exprs,
    int type,
    const char *szPrefix)
```

**Arguments**

`exprs`: 2-dimensional array of linear expressions.

`type`: type of an affine cone.

`szPrefix`: name prefix for new affine cone constraints.

**Return**

array of new affine cone constraint objects.

### Model::AddCol()

With column data, add variable to model via advanced interface.

**Synopsis**

```
Var AddCol(
    double obj,
    int colCnt,
    const int *colIdx,
    const double *colElem,
    char vtype,
    double lb,
    double ub,
    const char *szName)
```

**Arguments**

`obj`: coefficient of variable in objective function.

`colCnt`: number of terms in the column data.

`colIdx`: array of constraint indexes in the column data.

`colElem`: array of coefficients in the column data.

`vtype`: variable type.

`lb`: lower bound of new variable.

`ub`: upper bound of new variable.

`szName`: name of new vaiable, with default value of empty.

**Return**

new variable object.

### Model::AddCols()

Add variables to model via advanced interface.

**Synopsis**

```
VarArray AddCols(
    int count,
    double *pobj,
    char *pvtype,
    double *plb,
    double *pub,
```

```
    char const* const *names)
```

**Arguments**

`count`: number of variables added to model.

`pobj`: coefficients of new variables in objective function.

`pvtype`: variable types. If empty, all types are continuous.

`plb`: lower bounds for new variables. If empty, lower bounds are set to zero.

`pub`: upper bounds for new variables. If empty, upper bounds are set to infinity.

`names`: array of names of new variables, with default value of empty.

**Return**

new variable array objects.

### Model::AddCols()

With coefficient matrix data in CSC format, add variables to model via advanced interface.

**Synopsis**

```
VarArray AddCols(
    int count,
    double *pobj,
    int *colBeg,
    int *colCnt,
    int *colIdx,
    double *colElem,
    char *pvtype,
    double *plb,
    double *pub,
    char const* const *names)
```

**Arguments**

`count`: number of variables added to model.

`pobj`: coefficients of new variables in objective function.

`colBeg`: indexes of begin elements in CSC format. If empty, variables are added without column data.

`colCnt`: count of nonzero elements in each column. If empty, use colBeg to calculate count.

`colIdx`: constraint indexes of columns in CSC format.

`colElem`: corresponding constraint coefficients of columns in CSC format.

`pvtype`: variable types. If empty, all types are continuous.

`plb`: lower bounds for new variables. If empty, lower bounds are set to zero.

`pub`: upper bounds for new variables. If empty, upper bounds are set to infinity.

`names`: array of names of new variables, with default value of empty.

**Return**

new variable array objects.

### Model::AddCone()

Add a cone constraint to a model, given its dimension.

**Synopsis**

```
Cone AddCone(
    int dim,
    int type,
    char *pvtype,
    const char *szPrefix)
```

**Arguments**

`dim`: dimension of the cone constraint.

`type`: type of the cone constraint.

`pvtype`: types of variables in the cone.

`szPrefix`: name prefix of variables in the cone.

**Return**

object of new cone constraint.

### Model::AddCone()

Add a cone constraint to model.

**Synopsis**

```
Cone AddCone(const ConeBuilder &builder)
```

**Arguments**

`builder`: builder for new cone constraint.

**Return**

new cone constraint object.

### Model::AddCone()

Add a cone constraint to model.

**Synopsis**

```
Cone AddCone(const VarArray &vars, int type)
```

**Arguments**

`vars`: variables that participate in the cone constraint.

`type`: type of the cone constraint.

**Return**

object of new cone constraint.

### Model::AddCone()

Add a cone constraint to model.

**Synopsis**

```
Cone AddCone(const MVar<1> &vars, int type)
```

**Arguments**

`vars`: one-dimensional variables in the cone constraint.

`type`: type of the cone constraint.

**Return**

object of new cone constraint.

### Model::AddCones()

Add a list of cone constraints to a model.

**Synopsis**

```
ConeArray AddCones(const MVar<2> &vars, int type)
```

**Arguments**

`vars`: 2-dimensional array of variables.

`type`: type of a cone.

**Return**

array of new cone constraint objects.

### Model::AddConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint AddConstr(
    const Expr &expr,
    char sense,
    double rhs,
    const char *szName)
```

**Arguments**

`expr`: expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`szName`: optional, name of new constraint.

**Return**

new constraint object.

### Model::AddConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint AddConstr(
    const Expr &lhs,
    char sense,
    const Expr &rhs,
    const char *szName)
```

**Arguments**

`lhs`: left hand side expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side expression for the new constraint.

`szName`: optional, name of new constraint.

**Return**

new constraint object.

### Model::AddConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint AddConstr(
    const Expr &expr,
    double lb,
    double ub,
    const char *szName)
```

**Arguments**

`expr`: expression for the new constraint.

`lb`: lower bound for the new constraint.

`ub`: upper bound for the new constraint

`szName`: optional, name of new constraint.

**Return**

new constraint object.

### Model::AddConstr()

Add a linear constraint to a model.

**Synopsis**

```
Constraint AddConstr(const ConstrBuilder &builder, const char
*szName)
```

**Arguments**

`builder`: builder for the new constraint.

`szName`: optional, name of new constraint.

**Return**

new constraint object.

### Model::AddConstrs()

Add linear constraints to model.

**Synopsis**

```
ConstrArray AddConstrs(
    int count,
    char *pSense,
    double *pRhs,
    const char *szPrefix)
```

**Arguments**

`count`: number of constraints added to model.

`pSense`: sense array for new linear constraints, other than range sense.

`pRhs`: right hand side values for new constraints.

`szPrefix`: name prefix for new constraints.

**Return**

array of new constraint objects.

### Model::AddConstrs()

Add linear constraints to a model.

**Synopsis**

```
ConstrArray AddConstrs(
    int count,
    double *pLower,
    double *pUpper,
    const char *szPrefix)
```

**Arguments**

`count`: number of constraints added to the model.

`pLower`: lower bounds of new constraints.

`pUpper`: upper bounds of new constraints.

`szPrefix`: name prefix for new constraints.

**Return**

array of new constraint objects.

### Model::AddConstrs()

Add linear constraints to a model.

**Synopsis**

```
ConstrArray AddConstrs(
    int count,
    double *pLower,
    double *pUpper,
    const char *szNames,
    size_t len)
```

**Arguments**

`count`: number of constraints added to the model.

`pLower`: lower bounds of new constraints.

`pUpper`: upper bounds of new constraints.

`szNames`: name buffer of new constraints.

`len`: length of the name buffer.

**Return**

array of new constraint objects.

### Model::AddConstrs()

Add linear constraints to a model.

**Synopsis**

```
ConstrArray AddConstrs(const ConstrBuilderArray &builders, const
char *szPrefix)
```

**Arguments**

`builders`: builders for new constraints.

`szPrefix`: name prefix for new constraints.

**Return**

array of new constraint objects.

### Model::AddConstrs()

Add linear constraints to model.

**Synopsis**

```
ConstrArray AddConstrs(
    const ConstrBuilderArray &builders,
    const char *szNames,
    size_t len)
```

**Arguments**

`builders`: builders for new constraints.

`szNames`: name buffer of new constraints.

`len`: length of the name buffer.

**Return**

array of new constraint objects.

### Model::AddDenseMat()

Add a dense symmetric matrix to a model.

**Synopsis**

```
SymMatrix AddDenseMat(
    int dim,
    double *pVals,
    int len)
```

**Arguments**

`dim`: dimension of the dense symmetric matrix.

`pVals`: array of non-zero elements, filled in column-wise up to len or max length of symmetric matrix.

`len`: length of value array.

**Return**

new symmetric matrix object.

### Model::AddDenseMat()

Add a dense symmetric matrix to a model.

**Synopsis**

```
SymMatrix AddDenseMat(int dim, double val)
```

**Arguments**

`dim`: dimension of dense symmetric matrix.

`val`: value to fill dense symmetric matrix.

**Return**

new symmetric matrix object.

### Model::AddDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix AddDiagMat(int dim, double val)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

**Return**

new diagonal matrix object.

### Model::AddDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix AddDiagMat(
    int dim,
    double *pVals,
    int len)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`pVals`: array of values of diagonal elements.

`len`: length of value array.

**Return**

new diagonal matrix object.

### Model::AddDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix AddDiagMat(
    int dim,
    double val,
    int offset)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

`offset`: shift distance against diagonal line.

**Return**

new diagonal matrix object.

### Model::AddDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix AddDiagMat(
    int dim,
    double *pVals,
    int len,
    int offset)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`pVals`: array of values of diagonal elements.

`len`: length of value array.

`offset`: shift distance against diagonal line.

**Return**

new diagonal matrix object.

### Model::AddExpCone()

Add an exponential cone constraint to a model.

**Synopsis**

```
ExpCone AddExpCone(
    int type,
    char *pvtype,
    const char *szPrefix)
```

**Arguments**

`type`: type of the exponential cone constraint.

`pvtype`: types of variables in the exponential cone.

`szPrefix`: name prefix of variables in the exponential cone.

**Return**

object of new exponential cone constraint.

### Model::AddExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone AddExpCone(const ExpConeBuilder &builder)
```

**Arguments**

`builder`: builder for new exponential cone constraint.

**Return**

new exponential cone constraint object.

### Model::AddExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone AddExpCone(const VarArray &vars, int type)
```

**Arguments**

`vars`: variables that participate in the exponential cone constraint.

`type`: type of the exponential cone constraint.

**Return**

object of new exponential cone constraint.

### Model::AddExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone AddExpCone(const MVar<1> &vars, int type)
```

**Arguments**

`vars`: 1-dimensional variables in the exponential cone constraint.

`type`: type of the exponential cone constraint.

**Return**

object of new exponential cone constraint.

### Model::AddExpCones()

Add a list of exponential cone constraints to a model.

**Synopsis**

```
ExpConeArray AddExpCones(const MVar<2> &vars, int type)
```

**Arguments**

`vars`: 2-dimensional array of variables.

`type`: type of an exponential cone.

**Return**

array of new exponential cone constraint objects.

### Model::AddEyeMat()

Add an identity matrix to a model.

**Synopsis**

```
SymMatrix AddEyeMat(int dim)
```

**Arguments**

`dim`: dimension of identity matrix.

**Return**

new identity matrix object.

### Model::AddGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr AddGenConstrIndicator(const GenConstrBuilder &builder,
const char *szName)
```

**Arguments**

`builder`: builder for the general constraint.

`szName`: optional, name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model::AddGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr AddGenConstrIndicator(
    Var binVar,
    int binVal,
    const ConstrBuilder &builder,
    int type,
    const char *szName)
```

**Arguments**

`binVar`: binary indicator variable.

`binVal`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`builder`: builder for linear constraint.

`type`: type of general constraint with default value of COPT_INDICATOR_IF.

`szName`: optional, name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model::AddGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr AddGenConstrIndicator(
    Var binVar,
    int binVal,
    const Expr &expr,
    char sense,
    double rhs,
```

```
    int type,
    const char *szName)
```

**Arguments**

`binVar`: binary indicator variable.

`binVal`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`expr`: expression for new linear constraint.

`sense`: sense for new linear constraint.

`rhs`: right hand side value for new linear constraint.

`type`: type of general constraint with default value of COPT_INDICATOR_IF.

`szName`: optional, name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model::AddGenConstrIndicators()

Add general constraints of type indicator to model.

**Synopsis**

```
GenConstrArray AddGenConstrIndicators(
    Var binVar,
    int binVal,
    const MConstrBuilder<N> &builders,
    int type,
    const char *szPrefix)
```

**Arguments**

`binVar`: binary indicator variable.

`binVal`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`builders`: builder for linear constraints in matrix format.

`type`: type of general constraint with default value of COPT_INDICATOR_IF.

`szPrefix`: optional, name prefix of new general constraints.

**Return**

new general constraint objects of type indicator.

### Model::AddGenConstrIndicators()

Add a list of general constraints to model.

**Synopsis**

```
GenConstrArray AddGenConstrIndicators(const GenConstrBuilderArray
&builders, const char *szPrefix)
```

**Arguments**

`builders`: builders for new general constraints.

`szPrefix`: name prefix for new general constraints.

**Return**

array of new general constraint objects.

### Model::AddGenConstrIndicators()

Add a list of general constraints to model.

**Synopsis**

```
GenConstrArray AddGenConstrIndicators(
    const GenConstrBuilderArray &builders,
    const char *szNames,
    size_t len)
```

**Arguments**

`builders`: builders for new general constraints.

`szNames`: name buffer of new general constraints.

`len`: length of the name buffer.

**Return**

array of new general constraint objects.

### Model::AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(
    const Expr &lhs,
    char sense,
    double rhs,
    const char *szName)
```

**Arguments**

`lhs`: expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

`szName`: optional, name of lazy constraint.

### Model::AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(
    const Expr &lhs,
    char sense,
    const Expr &rhs,
    const char *szName)
```

**Arguments**

`lhs`: left hand side expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy constraint.

`szName`: optional, name of lazy constraint.

### Model::AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(const ConstrBuilder &builder, const char *szName)
```

**Arguments**

`builder`: builder for lazy constraint.

`szName`: optional, name of lazy constraint.

### Model::AddLazyConstrs()

Add lazy constraints to model.

**Synopsis**

```
void AddLazyConstrs(const ConstrBuilderArray &builders, const char
*szPrefix)
```

**Arguments**

`builders`: array of builders for lazy constraints.

`szPrefix`: name prefix of new lazy constraints.

### Model::AddLmiConstr()

Add an LMI constraint to model.

**Synopsis**

```
LmiConstraint AddLmiConstr(const LmiExpr &expr, const char *szName)
```

**Arguments**

`expr`: LMI expression for new LMI constraint.

`szName`: optional, name of new LMI constraint.

**Return**

new LMI constraint object.

### Model::AddMConstr()

Add a MConstr object in N-dimensions to model.

**Synopsis**

```
template <int N> MConstr<N> AddMConstr(const MConstrBuilder<N>
&builder, const char *szPrefix)
```

**Arguments**

`builder`: builder for MConstr object.

`szPrefix`: name prefix for constraints in MConstr object.

**Return**

new MConstr object.

### Model::AddMConstr()

Add a N-dimensional MConstr object to model.

**Synopsis**

```
template <int N> MConstr<N> AddMConstr(
    const MLinExpr<N> &exprs,
    char sense,
    double rhs,
    const char *szPrefix)
```

**Arguments**

`exprs`: N-dimensional MLinExpr object.

`sense`: sense for new linear constraints.

`rhs`: double value at right side of the new linear constraints.

`szPrefix`: name prefix for constraints in MConstr object.

**Return**

new MConstr object.

### Model::AddMPsdConstr()

Add a N-dimensional MPsdConstr object to model.

**Synopsis**

```
template <int N> MPsdConstr<N> AddMPsdConstr(const
MPsdConstrBuilder<N> &builder, const char *szPrefix)
```

**Arguments**

`builder`: builder for MPsdConstr object.

`szPrefix`: name prefix of PSD constraints in MPsdConstr object.

**Return**

new MPsdConstr object.

### Model::AddMPsdConstr()

Add a N-dimensional MPsdConstr object to model.

**Synopsis**

```
template <int N> MPsdConstr<N> AddMPsdConstr(
    const MPsdExpr<N> &exprs,
    char sense,
    double rhs,
    const char *szPrefix)
```

**Arguments**

`exprs`: N-dimensional MPsdExpr object.

`sense`: sense for new PSD constraints.

`rhs`: double value at right side of the new PSD constraints.

`szPrefix`: name prefix of PSD constraints in MPsdConstr object.

**Return**

new MPsdConstr object.

### Model::AddMQConstr()

Add a N-dimensional MQConstr object to model.

**Synopsis**

```
template <int N> MQConstr<N> AddMQConstr(const MQConstrBuilder<N>
&builder, const char *szPrefix)
```

**Arguments**

`builder`: builder for MQConstr object.

`szPrefix`: name prefix of quadratic constraints in MQConstr object.

**Return**

new MQConstr object.

### Model::AddMQConstr()

Add a N-dimensional MQConstr object to model.

**Synopsis**

```
template <int N> MQConstr<N> AddMQConstr(
    const MQuadExpr<N> &exprs,
    char sense,
    double rhs,
    const char *szPrefix)
```

**Arguments**

`exprs`: N-dimensional MQuadExpr object.

`sense`: sense for new quadratic constraints.

`rhs`: double value at right side of the new quadratic constraints.

`szPrefix`: name prefix of quadratic constraints in MQConstr object.

**Return**

new MQConstr object.

### Model::AddMVar()

Add a MVar object in N-dimensions to model.

**Synopsis**

```
template <int N> MVar<N> AddMVar(
    const Shape<N> &shp,
    char vtype,
    const char *szPrefix)
```

**Arguments**

`shp`: shape of MVar object.

`vtype`: type of variables in MVar object.

`szPrefix`: name prefix of variables in MVar object.

**Return**

new MVar object.

### Model::AddMVar()

Add a MVar object in N-dimensions to model.

**Synopsis**

```
MVar<N> AddMVar(
    const Shape<N> &shp,
    double lb,
    double ub,
    double obj,
    char vtype,
    const char *szPrefix)
```

**Arguments**

`shp`: shape of MVar object.

`lb`: lower bound for variables in MVar object.

`ub`: upper bound for variables in MVar object.

`obj`: objective coefficient for variables in MVar object.

`vtype`: type of variables in MVar object.

`szPrefix`: name prefix of variables in MVar object.

**Return**

new MVar object.

### Model::AddMVar()

Add a MVar object in N-dimensions to model.

**Synopsis**

```
MVar<N> AddMVar(
    const Shape<N> &shp,
    double *plb,
    double *pub,
    double *pobj,
    char *pvtype,
    const char *szPrefix)
```

**Arguments**

`shp`: shape of MVar object.

`plb`: lower bounds for variables in MVar object. If NULL, lower bounds are 0.0.

`pub`: upper bounds for variables in Mar object. If NULL, upper bounds are infinity or 1 for binary variables.

`pobj`: objective coefficient for variables in MVar object. If NULL, objective coefficients are 0.0.

`pvtype`: type of variables in MVar object. If NULL, variable types are continuous.

`szPrefix`: name prefix of variables in MVar object.

**Return**

new MVar object.

### Model::AddNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint AddNlConstr(
    const NlExpr &expr,
    char sense,
    double rhs,
    const char *szName)
```

**Arguments**

`expr`: nonlinear expression for the new constraint.

`sense`: sense for new nonlinear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`szName`: optional, name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model::AddNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint AddNlConstr(
    const NlExpr &lhs,
    char sense,
    const NlExpr &rhs,
    const char *szName)
```

**Arguments**

`lhs`: left hand side nonlinear expression for the new constraint.

`sense`: sense for new nonlinear constraint, other than range sense.

`rhs`: right hand side nonlinear expression for the new constraint.

`szName`: optional, name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model::AddNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint AddNlConstr(
    const NlExpr &expr,
    double lb,
    double ub,
    const char *szName)
```

**Arguments**

`expr`: nonlinear expression for the new constraint.

`lb`: lower bound for the new nonlinear constraint.

`ub`: upper bound for the new nonlinear constraint

`szName`: optional, name of new constraint.

**Return**

new nonlinear constraint object.

### Model::AddNlConstr()

Add a nonlinear constraint to a model.

**Synopsis**

```
NlConstraint AddNlConstr(const NlConstrBuilder &builder, const char
*szName)
```

**Arguments**

`builder`: builder for the new nonlinear constraint.

`szName`: optional, name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model::AddNlConstrs()

Add nonlinear constraints to a model.

**Synopsis**

```
NlConstrArray AddNlConstrs(const NlConstrBuilderArray &builders,
const char *szPrefix)
```

**Arguments**

`builders`: builders for new nonlinear constraints.

`szPrefix`: name prefix for new constraints.

**Return**

array of new nonlinear constraint objects.

### Model::AddNlConstrs()

Add nonlinear constraints to model.

**Synopsis**

```
NlConstrArray AddNlConstrs(
    const NlConstrBuilderArray &builders,
    const char *szNames,
    size_t len)
```

**Arguments**

`builders`: builders for new nonlinear constraints.

`szNames`: name buffer of new constraints.

`len`: length of the name buffer.

**Return**

array of new nonlinear constraint objects.

### Model::AddOnesMat()

Add a dense symmetric matrix of value one to a model.

**Synopsis**

```
SymMatrix AddOnesMat(int dim)
```

**Arguments**

`dim`: dimension of dense symmetric matrix.

**Return**

new symmetric matrix object.

### Model::AddPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint AddPsdConstr(
    const PsdExpr &expr,
    char sense,
    double rhs,
    const char *szName)
```

**Arguments**

`expr`: PSD expression for new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: double value at right side of the new PSD constraint.

`szName`: optional, name of new PSD constraint.

**Return**

new PSD constraint object.

### Model::AddPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint AddPsdConstr(
    const PsdExpr &expr,
    double lb,
    double ub,
    const char *szName)
```

**Arguments**

`expr`: expression for new PSD constraint.

`lb`: lower bound for ew PSD constraint.

`ub`: upper bound for new PSD constraint

`szName`: optional, name of new PSD constraint.

**Return**

new PSD constraint object.

### Model::AddPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint AddPsdConstr(
    const PsdExpr &lhs,
    char sense,
    const PsdExpr &rhs,
    const char *szName)
```

**Arguments**

`lhs`: PSD expression at left side of new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: PSD expression at right side of new PSD constraint.

`szName`: optional, name of new PSD constraint.

**Return**

new PSD constraint object.

### Model::AddPsdConstr()

Add a PSD constraint to a model.

**Synopsis**

```
PsdConstraint AddPsdConstr(const PsdConstrBuilder &builder, const
char *szName)
```

**Arguments**

`builder`: builder for new PSD constraint.

`szName`: optional, name of new PSD constraint.

**Return**

new PSD constraint object.

### Model::AddPsdVar()

Add a new PSD variable to model.

**Synopsis**

```
PsdVar AddPsdVar(int dim, const char *szName)
```

**Arguments**

`dim`: dimension of new PSD variable.

`szName`: name of new PSD variable.

**Return**

PSD variable object.

### Model::AddPsdVars()

Add new PSD variables to model.

**Synopsis**

```
PsdVarArray AddPsdVars(
    int count,
    int *pDim,
    const char *szPrefix)
```

**Arguments**

`count`: number of new PSD variables.

`pDim`: array of dimensions of new PSD variables.

`szPrefix`: name prefix of new PSD variables.

**Return**

array of PSD variable objects.

### Model::AddPsdVars()

Add new PSD variables to model.

**Synopsis**

```
PsdVarArray AddPsdVars(
    int count,
    int *pDim,
    const char *szNames,
    size_t len)
```

**Arguments**

`count`: number of new PSD variables.

`pDim`: array of dimensions of new PSD variables.

`szNames`: name buffer of new PSD variables.

`len`: length of the name buffer.

**Return**

array of PSD variable objects.

### Model::AddQConstr()

Add a quadratic constraint to model.

**Synopsis**

```
QConstraint AddQConstr(
    const QuadExpr &expr,
    char sense,
    double rhs,
    const char *szName)
```

**Arguments**

`expr`: quadratic expression for the new constraint.

`sense`: sense for new quadratic constraint.

`rhs`: double value at right side of the new quadratic constraint.

`szName`: optional, name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model::AddQConstr()

Add a quadratic constraint to model.

**Synopsis**

```
QConstraint AddQConstr(
    const QuadExpr &lhs,
    char sense,
    const QuadExpr &rhs,
    const char *szName)
```

**Arguments**

`lhs`: quadratic expression at left side of the new quadratic constraint.

`sense`: sense for new quadratic constraint.

`rhs`: quadratic expression at right side of the new quadratic constraint.

`szName`: optional, name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model::AddQConstr()

Add a quadratic constraint to a model.

**Synopsis**

```
QConstraint AddQConstr(const QConstrBuilder &builder, const char
*szName)
```

**Arguments**

`builder`: builder for the new quadratic constraint.

`szName`: optional, name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model::AddRow()

Add linear constraint to model via advanced interface.

**Synopsis**

```
AddRow(
    int rowCnt,
    const int *rowIdx,
    const double *rowElem,
    char sense,
    double rhs,
    const char *szName)
```

**Arguments**

`rowCnt`: number of terms in the linear constraint.

`rowIdx`: array of variable indexes in the linear constraint.

`rowElem`: array of coefficients in the linear constraint.

`sense`: constraint sense, excluding range bound.

`rhs`: right hand side value of constraint.

`szName`: name of new constraint, with default value of empty.

### Model::AddRow()

Add linear constraint to model via advanced interface.

**Synopsis**

```
AddRow(
    int rowCnt,
    const int *rowIdx,
    const double *rowElem,
    double lb,
    double ub,
    const char *szName)
```

**Arguments**

`rowCnt`: number of terms in the linear constraint.

`rowIdx`: array of variable indexes in the linear constraint.

`rowElem`: array of coefficients in the linear constraint.

`lb`: lower bound of new constraint.

`ub`: upper bound of new constraint.

`szName`: name of new constraint, with default value of empty.

### Model::AddRows()

Add linear constraints to model via advanced interface.

**Synopsis**

```
ConstrArray AddRows(
    int count,
    char *pSense,
    double *pRhs,
    char const* const *names)
```

**Arguments**

`count`: number of constraints added to model.

`pSense`: senses for new constraints, excluding range bound.

`pRhs`: right hand side values for new constraints.

`names`: array of names of new constraints, with default value of empty.

**Return**

new constraint array objects.

### Model::AddRows()

Add linear constraints to model via advanced interface.

**Synopsis**

```
ConstrArray AddRows(
    int count,
    double *plb,
    double *pub,
    char const* const *names)
```

**Arguments**

`count`: number of constraints added to model.

`plb`: lower bounds for new constraints.

`pub`: upper bounds for new constraints.

`names`: array of names of new constraints, with default value of empty.

**Return**

new constraint array objects.

### Model::AddRows()

With coefficient matrix in CSR format, add linear constraints to model via advanced interface.

**Synopsis**

```
ConstrArray AddRows(
    int count,
    int *rowBeg,
    int *rowCnt,
    int *rowIdx,
    double *rowElem,
    char *rowSense,
    double *rowBound,
    double *rowUpper,
    char const* const *names)
```

**Arguments**

`count`: number of constraints added to model.

`rowBeg`: indexes of begin elements in CSR format. If empty, constraints are added without coefficients.

`rowCnt`: count of nonzero elements in each row. If empty, use rowBeg to calculate count.

`rowIdx`: variable indexes of each row in CSR format.

`rowElem`: corresponding coefficients of each row in CSR format.

`rowSense`: senses for new constraints, including range bound. If empty, rowBound and rowUpper are used as lower and upper bounds respectively.

`rowBound`: bounds for new constraints, used for sense of equal, less_than and greater_than.

`rowUpper`: bounds for new constraints, used as range bound or upper bound.

`names`: array of names of new constraints, with default value of empty.

**Return**

new constraint array objects.

### Model::AddSos()

Add a SOS constraint to model.

**Synopsis**

```
Sos AddSos(const SosBuilder &builder)
```

**Arguments**

`builder`: builder for new SOS constraint.

**Return**

new SOS constraint object.

### Model::AddSos()

Add a SOS constraint to model.

**Synopsis**

```
Sos AddSos(
    const VarArray &vars,
    const double *pWeights,
    int type)
```

**Arguments**

`vars`: variables that participate in the SOS constraint.

`pWeights`: optional, weights for variables in the SOS constraint.

`type`: type of SOS constraint.

**Return**

new SOS constraint object.

### Model::AddSparseMat()

Add a sparse symmetric matrix to a model.

**Synopsis**

```
SymMatrix AddSparseMat(
    int dim,
    int nElems,
    int *pRows,
    int *pCols,
    double *pVals)
```

**Arguments**

`dim`: dimension of the sparse symmetric matrix.

`nElems`: number of non-zero elements in the sparse symmetric matrix.

`pRows`: array of row indexes of non-zero elements.

`pCols`: array of col indexes of non-zero elements.

`pVals`: array of values of non-zero elements.

**Return**

new symmetric matrix object.

### Model::AddSymMat()

Given a symmetric matrix expression, add results matrix to model.

**Synopsis**

```
SymMatrix AddSymMat(const SymMatExpr &expr)
```

**Arguments**

`expr`: symmetric matrix expression object.

**Return**

results symmetric matrix object.

### Model::AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(
    const Expr &lhs,
    char sense,
    double rhs,
    const char *szName)
```

**Arguments**

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

`szName`: optional, name of user cut.

### Model::AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(
    const Expr &lhs,
    char sense,
    const Expr &rhs,
    const char *szName)
```

**Arguments**

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

`szName`: optional, name of user cut.

### Model::AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(const ConstrBuilder &builder, const char *szName)
```

**Arguments**

`builder`: builder for user cut.

`szName`: optional, name of user cut.

### Model::AddUserCuts()

Add user cuts to model.

**Synopsis**

```
void AddUserCuts(const ConstrBuilderArray &builders, const char
*szPrefix)
```

**Arguments**

`builders`: array of builders for user cuts.

`szPrefix`: name prefix of new user cuts.

### Model::AddVar()

Add a variable to model.

**Synopsis**

```
Var AddVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    const char *szName)
```

**Arguments**

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`szName`: optional, name for new variable.

**Return**

new variable object.

### Model::AddVar()

Add a variable and the associated non-zero coefficients as column.

**Synopsis**

```
Var AddVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    const Column &col,
    const char *szName)
```

**Arguments**

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`col`: column object for specifying a set of constraints to which the variable belongs.

`szName`: optional, name for new variable.

**Return**

new variable object.

### Model::AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    char vtype,
    const char *szPrefix)
```

**Arguments**

`count`: the number of variables to add.

`vtype`: variable types for new variables.

`szPrefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model::AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    char vtype,
    const char *szNames,
    size_t len)
```

**Arguments**

`count`: the number of variables to add.

`vtype`: variable types for new variables.

`szNames`: name buffer for new variables.

`len`: length of name buffer.

**Return**

array of new variable objects.

### Model::AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    double lb,
    double ub,
    double obj,
    char vtype,
    const char *szPrefix)
```

**Arguments**

`count`: the number of variables to add.

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable types for new variables.

`szPrefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model::AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    double lb,
    double ub,
    double obj,
    char vtype,
    const char *szNames,
    size_t len)
```

**Arguments**

`count`: the number of variables to add.

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable types for new variables.

`szNames`: name buffer for new variables.

`len`: length of name buffer.

**Return**

array of new variable objects.

### Model::AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    double *plb,
    double *pub,
    double *pobj,
    char *pvtype,
    const char *szPrefix)
```

**Arguments**

`count`: the number of variables to add.

`plb`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`pub`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`pobj`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`pvtype`: variable types for new variables. if NULL, variable types are continuous.

`szPrefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model::AddVars()

Add new decision variables to a model.

**Synopsis**

```
VarArray AddVars(
    int count,
    double *plb,
    double *pub,
    double *pobj,
    char *pvtype,
    const char *szNames,
    size_t len)
```

**Arguments**

`count`: the number of variables to add.

`plb`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`pub`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`pobj`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`pvtype`: variable types for new variables. if NULL, variable types are continuous.

`szNames`: name buffer for new variables.

`len`: length of name buffer.

**Return**

array of new variable objects.

### Model::AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    double *plb,
    double *pub,
    double *pobj,
    char *pvtype,
    const ColumnArray &cols,
```

```
    const char *szPrefix)
```

**Arguments**

`count`: the number of variables to add.

`plb`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`pub`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`pobj`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`pvtype`: variable types for new variables. if NULL, variable types are continuous.

`cols`: column objects for specifying a set of constraints to which each new variable belongs.

`szPrefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model::BSolve()

Given a sparse vector, perform a backward solve. The result is stored in place in the same vector.

**Synopsis**

```
void BSolve(
    double *pVals,
    int *pInd,
    int *pLen)
```

**Arguments**

`pVals`: input values of the sparse vector; output values of the result, up to row size.

`pInd`: input indices of the sparse vector; output indices of the result, up to row size.

`pLen`: input length of the sparse vector; output length of the result.

### Model::BSolve()

Given a sparse vector, perform a backward solve. The result is stored in place in the same vector.

**Synopsis**

```
int BSolve(
    NdArray<double, 1> vals,
    NdArray<int, 1> idxes,
    int len)
```

**Arguments**

`vals`: input values of the sparse vector; output values of the result, up to row size.

`idxes`: input indices of the sparse vector; output indices of the result, up to row size.

`len`: length of the sparse input vector.

**Return**

length of the result sparse vector.

### Model::Clear()

Clear all settings including problem itself.

**Synopsis**

```
void Clear()
```

### Model::Clone()

Deep copy COPT model.

**Synopsis**

```
Model Clone()
```

**Return**

cloned model object.

### Model::ComputeIIS()

Compute IIS for infeasible model.

**Synopsis**

```
void ComputeIIS()
```

### Model::ComputeShadowPriceRange()

Compute the shadow price range for a given constraint.

**Synopsis**

```
void ComputeShadowPriceRange(
    int iRow,
    int ifRhs,
    double *pLower,
    double *pUpper)
```

**Arguments**

`iRow`: constraint (row) index.

`ifRhs`: for range constraint, whether to use RHS for computing shadow price.

`pLower`: output lower bound of the shadow price.

`pUpper`: output upper bound of the shadow price.

### Model::DelNlObj()

Delete nonlinear part of objective in model.

**Synopsis**

```
void DelNlObj()
```

### Model::DelObjN()

Delete linear part of specific multi-objective function in model.

**Synopsis**

```
void DelObjN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

### Model::DelPsdObj()

Delete PSD part of objective in model.

**Synopsis**

```
void DelPsdObj()
```

### Model::DelQuadObj()

Delete quadratic part of objective in model.

**Synopsis**

```
void DelQuadObj()
```

### Model::FeasRelax()

Compute feasibility relaxation for infeasible model.

**Synopsis**

```
void FeasRelax(
    const VarArray &vars,
    double *pColLowPen,
    double *pColUppPen,
    const ConstrArray &cons,
    double *pRowBndPen,
    double *pRowUppPen)
```

**Arguments**

`vars`: an array of variables.

`pColLowPen`: penalties for lower bounds of variables.

`pColUppPen`: penalties for upper bounds of variables.

`cons`: an array of constraints.

`pRowBndPen`: penalties for right hand sides of constraints.

`pRowUppPen`: penalties for upper bounds of range constraints.

### Model::FeasRelax()

Compute feasibility relaxation for infeasible model.

**Synopsis**

```
void FeasRelax(
    const MVar<1> &vars,
    double *pColLowPen,
    double *pColUppPen,
    const MConstr<1> &cons,
    double *pRowBndPen,
    double *pRowUppPen)
```

**Arguments**

`vars`: one-dimensional variables.

`pColLowPen`: penalties for lower bounds of variables.

`pColUppPen`: penalties for upper bounds of variables.

`cons`: one-dimensional constraints.

`pRowBndPen`: penalties for right hand sides of constraints.

`pRowUppPen`: penalties for upper bounds of range constraints.

### Model::FeasRelax()

Compute feasibility relaxation for infeasible model.

**Synopsis**

```
void FeasRelax(
    const MVar<N> &vars,
    const NdArray<double, N> &colLowPen,
    const NdArray<double, N> &colUppPen,
    const MConstr<M> &cons,
    const NdArray<double, M> &rowBndPen,
    const NdArray<double, M> &rowUppPen)
```

**Arguments**

`vars`: N-dimensional variables.

`colLowPen`: penalties for lower bounds of variables.

`colUppPen`: penalties for upper bounds of variables.

`cons`: M-dimensional constraints.

`rowBndPen`: penalties for right hand sides of constraints.

`rowUppPen`: penalties for upper bounds of range constraints.

### Model::FeasRelax()

Compute feasibility relaxation for infeasible model.

**Synopsis**

```
void FeasRelax(int ifRelaxVars, int ifRelaxCons)
```

**Arguments**

`ifRelaxVars`: whether to relax variables.

`ifRelaxCons`: whether to relax constraints.

### Model::FSolve()

Given a sparse vector, perform a forward solve. The result is stored in place in the same vector.

**Synopsis**

```
void FSolve(
    double *pVals,
    int *pInd,
    int *pLen)
```

**Arguments**

`pVals`: input values of the sparse vector; output values of the result, up to row size.

`pInd`: input indices of the sparse vector; output indices of the result, up to row size.

`pLen`: input length of the sparse vector; output length of the result.

### Model::FSolve()

Given a sparse vector, perform a forward solve. The result is stored in place in the same vector.

**Synopsis**

```
int FSolve(
    NdArray<double, 1> vals,
    NdArray<int, 1> idxes,
    int len)
```

**Arguments**

`vals`: input values of the sparse vector; output values of the result, up to row size.

`idxes`: input indices of the sparse vector; output indices of the result, up to row size.

`len`: length of the sparse input vector.

**Return**

length of the result sparse vector.

### Model::Get()

Query values of information associated with variables.

**Synopsis**

```
int Get(
    const char *szName,
    const VarArray &vars,
    double *pOut)
```

**Arguments**

`szName`: name of information.

`vars`: a list of desired variables.

`pOut`: output array of information values.

**Return**

the number of valid variables. If failed, return -1.

### Model::Get()

Query values of information associated with constraints.

**Synopsis**

```
int Get(
    const char *szName,
    const ConstrArray &constrs,
    double *pOut)
```

**Arguments**

`szName`: name of information.

`constrs`: a list of desired constraints.

`pOut`: output array of information values.

**Return**

the number of valid constraints. If failed, return -1.

### Model::Get()

Query values of information associated with nonlinear constraints.

**Synopsis**

```
int Get(
    const char *szName,
    const NlConstrArray &constrs,
    double *pOut)
```

**Arguments**

`szName`: name of information.

`constrs`: a list of desired nonlinear constraints.

`pOut`: output array of information values.

**Return**

the number of valid nonlinear constraints. If failed, return -1.

### Model::Get()

Query values of information associated with quadratic constraints.

**Synopsis**

```
int Get(
    const char *szName,
    const QConstrArray &constrs,
    double *pOut)
```

**Arguments**

`szName`: name of information.

`constrs`: a list of desired quadratic constraints.

`pOut`: output array of information values.

**Return**

the number of valid quadratic constraints. If failed, return -1.

### Model::Get()

Query values of information associated with PSD constraints.

**Synopsis**

```
int Get(
    const char *szName,
    const PsdConstrArray &constrs,
    double *pOut)
```

**Arguments**

`szName`: name of information.

`constrs`: a list of desired PSD constraints.

`pOut`: output array of information values.

**Return**

the number of valid PSD constraints. If failed, return -1.

### Model::GetAffineCone()

Get an affine cone constraint of given index in model.

**Synopsis**

```
AffineCone GetAffineCone(int idx)
```

**Arguments**

`idx`: index of the desired affine cone constraint.

**Return**

the desired affine cone constraint object.

### Model::GetAffineConeBuilder()

Get builder of given affine cone constraint in model.

**Synopsis**

```
AffineConeBuilder GetAffineConeBuilder(const AffineCone &cone)
```

**Arguments**

`cone`: an affine cone constraint.

**Return**

desired affine cone constraint builder.

### Model::GetAffineConeBuilders()

Get builders of desired affine cone constraints in model.

**Synopsis**

```
AffineConeBuilderArray GetAffineConeBuilders(const AffineConeArray
&cones)
```

**Arguments**

`cones`: array of affine cone constraints.

**Return**

array object of desired affine cone constraint builders.

### Model::GetAffineConeBuilders()

Get builders of all affine cone constraints in model.

**Synopsis**

```
AffineConeBuilderArray GetAffineConeBuilders()
```

**Return**

array object of all affine cone constraint builders.

### Model::GetAffineConeByName()

Get an affine cone constraint of given name in model.

**Synopsis**

```
AffineCone GetAffineConeByName(const char *szName)
```

**Arguments**

`szName`: name of the affine cone constraint.

**Return**

the desired affine cone constraint object.

### Model::GetAffineCones()

Get all affine cone constraints in model.

**Synopsis**

```
AffineConeArray GetAffineCones()
```

**Return**

array object of affine cone constraints.

### Model::GetBasisHead()

Get indices of the variables that make up the current basis.

**Synopsis**

```
NdArray<int, 1> GetBasisHead()
```

**Return**

array of basis head indices.

### Model::GetBasisHead()

Get indices of the variables that make up the current basis.

**Synopsis**

```
void GetBasisHead(int *bHead, int len)
```

**Arguments**

`bHead`: output array storing basis head indices.

`len`: length of output array, which is at least the number of rows in model.

### Model::GetCoeff()

Get the coefficient of variable in a linear constraint.

**Synopsis**

```
double GetCoeff(const Constraint &constr, const Var &var)
```

**Arguments**

`constr`: The requested constraint.

`var`: The requested variable.

**Return**

The requested coefficient.

### Model::GetCol()

Get a column object that have a list of constraints in which the variable participates.

**Synopsis**

```
Column GetCol(const Var &var)
```

**Arguments**

`var`: a variable object.

**Return**

a column object associated with a variable.

### Model::GetColBasis()

Get status of column basis.

**Synopsis**

```
int GetColBasis(int *pBasis)
```

**Arguments**

`pBasis`: integer pointer to basis status.

**Return**

number of columns.

### Model::GetCone()

Get a cone constraint of given index in model.

**Synopsis**

```
Cone GetCone(int idx)
```

**Arguments**

`idx`: index of the desired cone constraint.

**Return**

the desired cone constraint object.

### Model::GetConeBuilders()

Get builders of all cone constraints in model.

**Synopsis**

```
ConeBuilderArray GetConeBuilders()
```

**Return**

array object of all cone constraint builders.

### Model::GetConeBuilders()

Get builders of given cone constraints in model.

**Synopsis**

```
ConeBuilderArray GetConeBuilders(const ConeArray &cones)
```

**Arguments**

`cones`: array of cone constraints.

**Return**

array object of desired cone constraint builders.

### Model::GetCones()

Get all cone constraints in model.

**Synopsis**

```
ConeArray GetCones()
```

**Return**

array object of cone constraints.

### Model::GetConstr()

Get a constraint of given index in model.

**Synopsis**

```
Constraint GetConstr(int idx)
```

**Arguments**

`idx`: index of the desired constraint.

**Return**

the desired constraint object.

### Model::GetConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

**Synopsis**

```
ConstrBuilder GetConstrBuilder(Constraint constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

constraint builder object.

### Model::GetConstrBuilders()

Get builders of all constraints in model.

**Synopsis**

```
ConstrBuilderArray GetConstrBuilders()
```

**Return**

array object of constraint builders.

### Model::GetConstrByName()

Get a constraint of given name in model.

**Synopsis**

```
Constraint GetConstrByName(const char *szName)
```

**Arguments**

`szName`: name of the desired constraint.

**Return**

the desired constraint object.

### Model::GetConstrLowerIIS()

Get IIS status of lower bounds of constraints.

**Synopsis**

```
int GetConstrLowerIIS(const ConstrArray &constrs, int *pLowerIIS)
```

**Arguments**

`constrs`: Array of constraints.

`pLowerIIS`: IIS status of lower bounds of constraints.

**Return**

Number of constraints.

### Model::GetConstrs()

Get all constraints in model.

**Synopsis**

```
ConstrArray GetConstrs()
```

**Return**

array object of constraints.

### Model::GetConstrUpperIIS()

Get IIS status of upper bounds of constraints.

**Synopsis**

```
int GetConstrUpperIIS(const ConstrArray &constrs, int *pUpperIIS)
```

**Arguments**

`constrs`: Array of constraints.

`pUpperIIS`: IIS status of upper bounds of constraints.

**Return**

Number of constraints.

### Model::GetDblAttr()

Get value of a COPT double attribute.

**Synopsis**

```
double GetDblAttr(const char *szAttr)
```

**Arguments**

`szAttr`: name of double attribute.

**Return**

value of double attribute.

### Model::GetDblAttrN()

Get value of a double attribute of a multi-objective function.

**Synopsis**

```
double GetDblAttrN(int idx, const char *szAttr)
```

**Arguments**

`idx`: index of a multi-objective function.

`szAttr`: name of double attribute.

**Return**

value of double attribute.

### Model::GetDblParam()

Get value of a COPT double parameter.

**Synopsis**

```
double GetDblParam(const char *szParam)
```

**Arguments**

`szParam`: name of double parameter.

**Return**

value of double parameter.

### Model::GetDblParamN()

Get value of a double parameter of a multi-objective function.

**Synopsis**

```
double GetDblParamN(int idx, const char *szParam)
```

**Arguments**

`idx`: index of a multi-objective function.

`szParam`: name of double parameter.

**Return**

value of double parameter.

### Model::GetExpCone()

Get an exponential cone constraint of given index in model.

**Synopsis**

```
ExpCone GetExpCone(int idx)
```

**Arguments**

`idx`: index of the desired exponential cone constraint.

**Return**

the desired exponential cone constraint object.

### Model::GetExpConeBuilders()

Get builders of all exponential cone constraints in model.

**Synopsis**

```
ExpConeBuilderArray GetExpConeBuilders()
```

**Return**

array object of all exponential cone constraint builders.

### Model::GetExpConeBuilders()

Get builders of given exponential cone constraints in model.

**Synopsis**

```
ExpConeBuilderArray GetExpConeBuilders(const ExpConeArray &cones)
```

**Arguments**

`cones`: array of exponential cone constraints.

**Return**

array object of desired exponential cone constraint builders.

### Model::GetExpCones()

Get all exponential cone constraints in model.

**Synopsis**

```
ExpConeArray GetExpCones()
```

**Return**

array object of exponential cone constraints.

### Model::GetGenConstr()

Get a general constraint of given index in model.

**Synopsis**

```
GenConstr GetGenConstr(int idx)
```

**Arguments**

`idx`: index of the desired general constraint.

**Return**

the desired general constraint object.

### Model::GetGenConstrByName()

Get a general constraint of given name in model.

**Synopsis**

```
GenConstr GetGenConstrByName(const char *szName)
```

**Arguments**

`szName`: name of the desired general constraint.

**Return**

the desired general constraint object.

### Model::GetGenConstrIndicator()

Get builder of given general constraint of type indicator.

**Synopsis**

```
GenConstrBuilder GetGenConstrIndicator(const GenConstr &indicator)
```

**Arguments**

`indicator`: a general constraint of type indicator.

**Return**

builder object of general constraint of type indicator.

### Model::GetGenConstrIndicators()

Get builders of all general constraints in model.

**Synopsis**

```
GenConstrBuilderArray GetGenConstrIndicators()
```

**Return**

array object of general constraint builders.

### Model::GetGenConstrs()

Get all general constraints in model.

**Synopsis**

```
GenConstrArray GetGenConstrs()
```

**Return**

array object of general constraints.

### Model::GetIndicatorIIS()

Get IIS status of indicator constraints.

**Synopsis**

```
int GetIndicatorIIS(const GenConstrArray &genconstrs, int *pIIS)
```

**Arguments**

`genconstrs`: Array of indicator constraints.

`pIIS`: IIS status of indicator constraints.

**Return**

Number of indicator constraints.

### Model::GetIntAttr()

Get value of a COPT integer attribute.

**Synopsis**

```
int GetIntAttr(const char *szAttr)
```

**Arguments**

`szAttr`: name of integer attribute.

**Return**

value of integer attribute.

### Model::GetIntAttrN()

Get value of an integer attribute of a multi-objective function.

**Synopsis**

```
int GetIntAttrN(int idx, const char *szAttr)
```

**Arguments**

`idx`: index of a multi-objective function.

`szAttr`: name of integer attribute.

**Return**

value of integer attribute.

### Model::GetIntParam()

Get value of a COPT integer parameter.

**Synopsis**

```
int GetIntParam(const char *szParam)
```

**Arguments**

`szParam`: name of integer parameter.

**Return**

value of integer parameter.

### Model::GetIntParamN()

Get value of an integer parameter of a multi-objective function.

**Synopsis**

```
int GetIntParamN(int idx, const char *szParam)
```

**Arguments**

`idx`: index of a multi-objective function.

`szParam`: name of integer parameter.

**Return**

value of integer parameter.

### Model::GetLmiCoeff()

Get the symmetric matrix of variable in LMI constraint.

**Synopsis**

```
SymMatrix GetLmiCoeff(const LmiConstraint &constr, const Var &var)
```

**Arguments**

`constr`: The desired LMI constraint.

`var`: The desired variable.

**Return**

The associated coefficient matrix.

#### Model::GetLmiConstr()

Get LMI constraint of given index in model.

**Synopsis**

```
LmiConstraint GetLmiConstr(int idx)
```

**Arguments**

`idx`: index of desired LMI constraint.

**Return**

LMI constraint object.

#### Model::GetLmiConstrByName()

Get LMI constraint of given name in model.

**Synopsis**

```
LmiConstraint GetLmiConstrByName(const char *szName)
```

**Arguments**

`szName`: name of desired LMI constraint.

**Return**

LMI constraint object.

#### Model::GetLmiConstrs()

Get all LMI constraints in model.

**Synopsis**

```
LmiConstrArray GetLmiConstrs()
```

**Return**

array object of LMI constraints.

#### Model::GetLmiRhs()

Get the symmetric matrix of constant of LMI constraint.

**Synopsis**

```
SymMatrix GetLmiRhs(const LmiConstraint &constr)
```

**Arguments**

`constr`: The desired LMI constraint.

**Return**

matrix of constant term.

### Model::GetLmiRow()

Get variables and associated symmetric matrices that participate in a LMI constraint.

**Synopsis**

```
LmiExpr GetLmiRow(const LmiConstraint &constr)
```

**Arguments**

`constr`: given LMI constraint object.

**Return**

pointer to LMI expression object of LMI constraint.

### Model::GetLpSolution()

Get LP solution.

**Synopsis**

```
void GetLpSolution(
    double *pValue,
    double *pSlack,
    double *pRowDual,
    double *pRedCost)
```

**Arguments**

`pValue`: optional, double pointer to solution values.

`pSlack`: optional, double pointer to slack values.

`pRowDual`: optional, double pointer to dual values.

`pRedCost`: optional, double pointer to reduced costs.

### Model::GetName()

Get name of the model.

**Synopsis**

```
const char *GetName()
```

**Return**

model name.

### Model::GetNlConstr()

Get a nonlinear constraint of given index in model.

**Synopsis**

```
NlConstraint GetNlConstr(int idx)
```

**Arguments**

`idx`: index of the desired nonlinear constraint.

**Return**

the desired nonlinear constraint object.

### Model::GetNlConstrBuilder()

Get builder of a nonlinear constraint in model, including nonlinear expression, sense and RHS.

**Synopsis**

```
NlConstrBuilder GetNlConstrBuilder(const NlConstraint &constr)
```

**Arguments**

`constr`: a nonlinear constraint object.

**Return**

nonlinear constraint builder object.

### Model::GetNlConstrBuilders()

Get builders of all nonlinear constraints in model.

**Synopsis**

```
NlConstrBuilderArray GetNlConstrBuilders()
```

**Return**

array object of nonlinear constraint builders.

### Model::GetNlConstrByName()

Get a nonlinear constraint of given name in model.

**Synopsis**

```
NlConstraint GetNlConstrByName(const char *szName)
```

**Arguments**

`szName`: name of the desired constraint.

**Return**

the desired nonlinear constraint object.

### Model::GetNlConstrs()

Get all nonlinear constraints in model.

**Synopsis**

```
NlConstrArray GetNlConstrs()
```

**Return**

array object of nonlinear constraints.

### Model::GetNlObjective()

Get nonlinear objective of model.

**Synopsis**

```
NlExpr GetNlObjective()
```

**Return**

a nonlinear expression object.

### Model::GetNlRow()

Get nonlinear expression of a nonlinear constraint.

**Synopsis**

```
NlExpr GetNlRow(const NlConstraint &constr)
```

**Arguments**

`constr`: a nonlinear constraint object.

**Return**

output object of nonlinear expression.

### Model::GetObjective()

Get linear expression of objective for model.

**Synopsis**

```
Expr GetObjective()
```

**Return**

a linear expression object.

### Model::GetObjectiveN()

Get linear expression of a multi-objective function in model.

**Synopsis**

```
Expr GetObjectiveN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

**Return**

a linear expression object.

### Model::GetObjParamN()

Get value of objective parameter of a multi-objective function.

**Synopsis**

```
double GetObjParamN(int idx, const char *szParam)
```

**Arguments**

`idx`: index of a multi-objective function.

`szParam`: name of objective parameter, including priority, weight, abstol and reltol.

**Return**

value of objective parameter.

### Model::GetParamAttrType()

Get type of a COPT parameter or attribute.

**Synopsis**

```
int GetParamAttrType(const char *szName)
```

**Arguments**

`szName`: name of parameter or attribute.

**Return**

type of parameter or attribute.

### Model::GetParamInfo()

Get current, default, minimum, maximum of COPT integer parameter.

**Synopsis**

```
void GetParamInfo(
    const char *szParam,
    int *pnCur,
    int *pnDef,
    int *pnMin,
    int *pnMax)
```

**Arguments**

`szParam`: name of integer parameter.

`pnCur`: out, current value of integer parameter.

`pnDef`: out, default value of integer parameter.

`pnMin`: out, minimum value of integer parameter.

`pnMax`: out, maximum value of integer parameter.

### Model::GetParamInfo()

Get current, default, minimum, maximum of COPT double parameter.

**Synopsis**

```
void GetParamInfo(
    const char *szParam,
    double *pdCur,
    double *pdDef,
    double *pdMin,
    double *pdMax)
```

**Arguments**

`szParam`: name of double parameter.

`pdCur`: out, current value of double parameter.

`pdDef`: out, default value of double parameter.

`pdMin`: out, minimum value of double parameter.

`pdMax`: out, maximum value of double parameter.

### Model::GetPoolObjVal()

Get the iSol-th objective value in solution pool.

**Synopsis**

```
double GetPoolObjVal(int iSol)
```

**Arguments**

`iSol`: Index of solution.

**Return**

The requested objective value.

### Model::GetPoolObjValN()

Get the objective value of required multi-objective function in solution pool.

**Synopsis**

```
double GetPoolObjValN(int idx, int iSol)
```

**Arguments**

`idx`: index of a multi-objective function.

`iSol`: index of solution.

**Return**

value of required multi-objective function.

### Model::GetPoolSolution()

Get the iSol-th solution in solution pool.

**Synopsis**

```
int GetPoolSolution(
    int iSol,
    const VarArray &vars,
    double *pColVals)
```

**Arguments**

`iSol`: Index of solution.

`vars`: The requested variables.

`pColVals`: Pointer to the requested solutions.

**Return**

The length of requested solution array.

### Model::GetPsdCoeff()

Get the symmetric matrix of PSD variable in a PSD constraint.

**Synopsis**

```
SymMatrix GetPsdCoeff(const PsdConstraint &constr, const PsdVar
&var)
```

**Arguments**

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

**Return**

The associated coefficient matrix.

### Model::GetPsdConstr()

Get PSD constraint of given index in model.

**Synopsis**

```
PsdConstraint GetPsdConstr(int idx)
```

**Arguments**

`idx`: index of desired PSD constraint.

**Return**

PSD constraint object.

### Model::GetPsdConstrBuilder()

Get builder of a PSD constraint in model, including PSD variables, sense and associated symmetric matrix.

**Synopsis**

```
PsdConstrBuilder GetPsdConstrBuilder(const PsdConstraint &constr)
```

**Arguments**

`constr`: PSD constraint object.

**Return**

pointer to PSD constraint builder object.

### Model::GetPsdConstrBuilders()

Get builders of all PSD constraints in model.

**Synopsis**

```
PsdConstrBuilderArray GetPsdConstrBuilders()
```

**Return**

pointer to array object of PSD constraint builders.

### Model::GetPsdConstrByName()

Get PSD constraint of given name in model.

**Synopsis**

```
PsdConstraint GetPsdConstrByName(const char *szName)
```

**Arguments**

`szName`: name of desired PSD constraint.

**Return**

PSD constraint object.

### Model::GetPsdConstrs()

Get all PSD constraints in model.

**Synopsis**

```
PsdConstrArray GetPsdConstrs()
```

**Return**

pointer to array object of PSD constraints.

### Model::GetPsdObjective()

Get PSD objective of model.

**Synopsis**

```
PsdExpr GetPsdObjective()
```

**Return**

a PSD expression object.

### Model::GetPsdRow()

Get PSD variables and associated symmetric matrices that participate in a PSD constraint.

**Synopsis**

```
PsdExpr GetPsdRow(const PsdConstraint &constr)
```

**Arguments**

`constr`: PSD constraint object.

**Return**

pointer to PSD expression object of the PSD constraint.

### Model::GetPsdRow()

Get PSD variables, associated symmetric matrix, LB/UB that participate in a PSD constraint.

**Synopsis**

```
PsdExpr GetPsdRow(
    const PsdConstraint &constr,
    double *pLower,
    double *pUpper)
```

**Arguments**

`constr`: a PSD constraint object.

`pLower`: pointer to double value of lower bound.

`pUpper`: pointer to double value of upper bound.

**Return**

pointer to PSD expression object of the PSD constraint.

### Model::GetPsdVar()

Get a PSD variable of given index in model.

**Synopsis**

```
PsdVar GetPsdVar(int idx)
```

**Arguments**

`idx`: index of the desired PSD variable.

**Return**

the desired PSD variable object.

### Model::GetPsdVarByName()

Get a PSD variable of given name in model.

**Synopsis**

```
PsdVar GetPsdVarByName(const char *szName)
```

**Arguments**

`szName`: name of the desired PSD variable.

**Return**

the desired PSD variable object.

### Model::GetPsdVars()

Get all PSD variables in model.

**Synopsis**

```
PsdVarArray GetPsdVars()
```

**Return**

array object of PSD variables.

### Model::GetQConstr()

Get a quadratic constraint of given index in model.

**Synopsis**

```
QConstraint GetQConstr(int idx)
```

**Arguments**

`idx`: index of the desired quadratic constraint.

**Return**

the desired quadratic constraint object.

### Model::GetQConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

**Synopsis**

```
QConstrBuilder GetQConstrBuilder(const QConstraint &constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

constraint builder object.

### Model::GetQConstrBuilders()

Get builders of all constraints in model.

**Synopsis**

```
QConstrBuilderArray GetQConstrBuilders()
```

**Return**

array object of constraint builders.

### Model::GetQConstrByName()

Get a quadratic constraint of given name in model.

**Synopsis**

```
QConstraint GetQConstrByName(const char *szName)
```

**Arguments**

`szName`: name of the desired constraint.

**Return**

the desired quadratic constraint object.

### Model::GetQConstrs()

Get all quadratic constraints in model.

**Synopsis**

```
QConstrArray GetQConstrs()
```

**Return**

array object of quadratic constraints.

### Model::GetQuadObjective()

Get quadratic objective of model.

**Synopsis**

```
QuadExpr GetQuadObjective()
```

**Return**

a quadratic expression object.

### Model::GetQuadRow()

Get two variables and associated coefficients that participate in a quadratic constraint.

**Synopsis**

```
QuadExpr GetQuadRow(const QConstraint &constr)
```

**Arguments**

`constr`: a quadratic constraint object.

**Return**

quadratic expression object of the constraint.

### Model::GetQuadRow()

Get two variables and associated coefficients that participate in a quadratic constraint.

**Synopsis**

```
QuadExpr GetQuadRow(
    const QConstraint &constr,
    char *pSense,
    double *pBound)
```

**Arguments**

`constr`: a quadratic constraint object.

`pSense`: sense of quadratic constraint.

`pBound`: right hand side of quadratic constraint.

**Return**

quadratic expression object of the constraint.

### Model::GetRow()

Get variables that participate in a constraint, and the associated coefficients.

**Synopsis**

```
Expr GetRow(const Constraint &constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

expression object of the constraint.

### Model::GetRowBasis()

Get status of row basis.

**Synopsis**

```
int GetRowBasis(int *pBasis)
```

**Arguments**

`pBasis`: integer pointer to basis status.

**Return**

number of rows.

#### Model::GetSolution()

Get MIP solution.

**Synopsis**

```
void GetSolution(double *pValue)
```

**Arguments**

`pValue`: double pointer to solution values.

#### Model::GetSos()

Get a SOS constraint of given index in model.

**Synopsis**

```
Sos GetSos(int idx)
```

**Arguments**

`idx`: index of the desired SOS constraint.

**Return**

the desired SOS constraint object.

#### Model::GetSosBuilders()

Get builders of all SOS constraints in model.

**Synopsis**

```
SosBuilderArray GetSosBuilders()
```

**Return**

array object of SOS constraint builders.

#### Model::GetSosBuilders()

Get builders of given SOS constraints in model.

**Synopsis**

```
SosBuilderArray GetSosBuilders(const SosArray &soss)
```

**Arguments**

`soss`: array of SOS constraints.

**Return**

array object of desired SOS constraint builders.

### Model::GetSOSIIS()

Get IIS status of SOS constraints.

**Synopsis**

```
int GetSOSIIS(const SosArray &soss, int *pIIS)
```

**Arguments**

`soss`: Array of SOS constraints.

`pIIS`: IIS status of SOS constraints.

**Return**

Number of SOS constraints.

### Model::GetSoss()

Get all SOS constraints in model.

**Synopsis**

```
SosArray GetSoss()
```

**Return**

array object of SOS constraints.

### Model::GetSymMat()

Get a symmetric matrix of given index in model.

**Synopsis**

```
SymMatrix GetSymMat(int idx)
```

**Arguments**

`idx`: index of the desired symmetric matrix.

**Return**

the desired symmetric matrix object.

### Model::GetVar()

Get a variable of given index in model.

**Synopsis**

```
Var GetVar(int idx)
```

**Arguments**

`idx`: index of the desired variable.

**Return**

the desired variable object.

### Model::GetVarByName()

Get a variable of given name in model.

**Synopsis**

```
Var GetVarByName(const char *szName)
```

**Arguments**

`szName`: name of the desired variable.

**Return**

the desired variable object.

### Model::GetVarLowerIIS()

Get IIS status of lower bounds of variables.

**Synopsis**

```
int GetVarLowerIIS(const VarArray &vars, int *pLowerIIS)
```

**Arguments**

`vars`: Array of variables

`pLowerIIS`: IIS status of lower bounds of variables.

**Return**

Number of variables.

### Model::GetVars()

Get all variables in model.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variable array object.

### Model::GetVarUpperIIS()

Get IIS status of upper bounds of variables.

**Synopsis**

```
int GetVarUpperIIS(const VarArray &vars, int *pUpperIIS)
```

**Arguments**

`vars`: Array of variables

`pUpperIIS`: IIS status of upper bounds of variables.

**Return**

Number of variables.

### Model::Interrupt()

Interrupt optimization of current problem.

**Synopsis**

```
void Interrupt()
```

### Model::LoadMatrix()

Load matrix data to formulate problem.

**Synopsis**

```
void LoadMatrix(
    int nCol,
    int nRow,
    double *pCost,
    int *pMatBeg,
    int *pMatCnt,
    int *pMatIdx,
    double *pMatElem,
    double *pLHS,
    double *pRHS,
    double *pLB,
    double *pUB,
    char *pType)
```

**Arguments**

`nCol`: Number of columns.

`nRow`: Number of rows.

`pCost`: Objective cost.

`pMatBeg`: Begin pointer.

`pMatCnt`: Counter pointer.

`pMatIdx`: Index pointer.

`pMatElem`: Elements pointer.

`pLHS`: Lower bounds of rows.

`pRHS`: Upper bounds of rows.

`pLB`: Lower bounds of columns.

`pUB`: Upper bounds of columns.

`pType`: Column types.

### Model::LoadMipStart()

Load final initial values of variables to the problem.

**Synopsis**

```
void LoadMipStart()
```

### Model::LoadNlData()

Load nonlinear data into problem with customized callback.

**Synopsis**

```
void LoadNlData(
    int nCol,
    int nRow,
    int iObjSense,
    int nObjGrad,
    const int *objGradIndex,
    int nnzJac,
    const int *jacRowIndex,
    const int *jacColIndex,
    int nnzHess,
    const int *hessRowIndex,
    const int *hessColIndex,
    const double *colLower,
    const double *colUpper,
    const double *rowLower,
    const double *rowUpper,
    const double *initColVal,
    int evalType,
    INlpCallback *pcb)
```

**Arguments**

`nCol`: number of columns.

`nRow`: number of rows.

`iObjSense`: sense of objective.

`nObjGrad`: number of objective gradients.

`objGradIndex`: a list of index for objective gradients.

`nnzJac`: number of non-zeros in Jacobian matrix.

`jacRowIndex`: a list of row index in Jacobian matrix.

`jacColIndex`: a list of col index in Jacobian matrix.

`nnzHess`: number of non-zeros in Hessian matrix.

`hessRowIndex`: a list of row index in Hessian matrix.

`hessColIndex`: a list of col index in Hessian matrix.

`colLower`: lower bounds for columns.

`colUpper`: upper bounds for columns.

`rowLower`: lower bounds for rows.

`rowUpper`: upper bounds for rows.

`initColVal`: a list of initial column values.

`evalType`: evaluation type of the nonlinear model.

`pcb`: pointer to callback object for nonlinear model.

### Model::LoadTuneParam()

Load specified tuned parameters into model.

**Synopsis**

```
void LoadTuneParam(int idx)
```

**Arguments**

`idx`: Index of tuned parameters.

### Model::Read()

Read problem, solution, basis, MIP start or COPT parameters from file.

**Synopsis**

```
void Read(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadBasis()

Read basis from file.

**Synopsis**

```
void ReadBasis(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadBin()

Read problem in COPT binary format from file.

**Synopsis**

```
void ReadBin(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadCbf()

Read problem in CBF format from file.

**Synopsis**

```
void ReadCbf(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadJsonSol()

Read solution in format of JSON from file.

**Synopsis**

```
void ReadJsonSol(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadLp()

Read problem in LP format from file.

**Synopsis**

```
void ReadLp(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadMps()

Read problem in MPS format from file.

**Synopsis**

```
void ReadMps(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadMst()

Read MIP start information from file.

**Synopsis**

```
void ReadMst(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadOrd()

Read branching order from file.

**Synopsis**

```
void ReadOrd(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadParam()

Read COPT parameters from file.

**Synopsis**

```
void ReadParam(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadSdpa()

Read problem in SDPA format from file.

**Synopsis**

```
void ReadSdpa(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadSol()

Read solution from file.

**Synopsis**

```
void ReadSol(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::ReadTune()

Read tuning parameters from file.

**Synopsis**

```
void ReadTune(const char *szFileName)
```

**Arguments**

`szFileName`: an input file name.

### Model::Remove()

Remove a list of variables from model.

**Synopsis**

```
void Remove(VarArray &vars)
```

**Arguments**

`vars`: an array of variables.

### Model::Remove()

Remove a list of constraints from model.

**Synopsis**

```
void Remove(ConstrArray &constrs)
```

**Arguments**

`constrs`: an array of constraints.

### Model::Remove()

Remove a list of nonlinear constraints from model.

**Synopsis**

```
void Remove(NlConstrArray &constrs)
```

**Arguments**

`constrs`: an array of nonlinear constraints.

### Model::Remove()

Remove a list of SOS constraints from model.

**Synopsis**

```
void Remove(SosArray &soss)
```

**Arguments**

`soss`: an array of SOS constraints.

### Model::Remove()

Remove a list of general constraints from model.

**Synopsis**

```
void Remove(GenConstrArray &genConstrs)
```

**Arguments**

`genConstrs`: an array of general constraints.

### Model::Remove()

Remove a list of cone constraints from model.

**Synopsis**

```
void Remove(ConeArray &cones)
```

**Arguments**

`cones`: an array of cone constraints.

### Model::Remove()

Remove a list of exponential cone constraints from model.

**Synopsis**

```
void Remove(ExpConeArray &cones)
```

**Arguments**

`cones`: an array of exponential cone constraints.

### Model::Remove()

Remove a list of affine cone constraints from model.

**Synopsis**

```
void Remove(AffineConeArray &cones)
```

**Arguments**

`cones`: an array of affine cone constraints.

### Model::Remove()

Remove a list of quadratic constraints from model.

**Synopsis**

```
void Remove(QConstrArray &qconstrs)
```

**Arguments**

`qconstrs`: an array of quadratic constraints.

### Model::Remove()

Remove a list of PSD variables from model.

**Synopsis**

```
void Remove(PsdVarArray &vars)
```

**Arguments**

`vars`: an array of PSD variables.

#### Model::Remove()

Remove a list of PSD constraints from model.

**Synopsis**

```
void Remove(PsdConstrArray &constrs)
```

**Arguments**

`constrs`: an array of PSD constraints.

#### Model::Remove()

Remove a list of LMI constraints from model.

**Synopsis**

```
void Remove(LmiConstrArray &constrs)
```

**Arguments**

`constrs`: an array of LMI constraints.

#### Model::Reset()

Reset solution of problem only.

**Synopsis**

```
void Reset()
```

#### Model::ResetAll()

Reset solution of problem, and additional information such as MIP start, etc.

**Synopsis**

```
void ResetAll()
```

#### Model::ResetObjParamN()

Reset objective parameters of a multi-objective function.

**Synopsis**

```
void ResetObjParamN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

#### Model::ResetParam()

Reset parameters to default settings.

**Synopsis**

```
void ResetParam()
```

### Model::ResetParamN()

Reset double and integer parameters of a multi-objective function.

**Synopsis**

```
void ResetParamN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

### Model::Set()

Set values of information associated with variables.

**Synopsis**

```
void Set(
    const char *szName,
    const VarArray &vars,
    double *pVals,
    int len)
```

**Arguments**

`szName`: name of double information.

`vars`: a list of desired variables.

`pVals`: array of information values.

`len`: length of value array.

### Model::Set()

Set values of information associated with constraints.

**Synopsis**

```
void Set(
    const char *szName,
    const ConstrArray &constrs,
    double *pVals,
    int len)
```

**Arguments**

`szName`: name of double information.

`constrs`: a list of desired constraints.

`pVals`: array of information values.

`len`: length of value array.

### Model::Set()

Set values of information associated with nonlinear constraints.

**Synopsis**

```
void Set(
    const char *szName,
    const NlConstrArray &constrs,
    double *pVals,
    int len)
```

**Arguments**

`szName`: name of double information.

`constrs`: a list of desired nonlinear constraints.

`pVals`: array of values of information.

`len`: length of value array.

### Model::Set()

Set values of information associated with PSD constraints.

**Synopsis**

```
void Set(
    const char *szName,
    const PsdConstrArray &constrs,
    double *pVals,
    int len)
```

**Arguments**

`szName`: name of double information.

`constrs`: a list of desired PSD constraints.

`pVals`: array of values of information.

`len`: length of value array.

### Model::SetBasis()

Set column and row basis status to model.

**Synopsis**

```
void SetBasis(int *pColBasis, int *pRowBasis)
```

**Arguments**

`pColBasis`: pointer to status of column basis.

`pRowBasis`: pointer to status of row basis.

### Model::SetCallback()

Set user callback to COPT model.

**Synopsis**

```
void SetCallback(ICallback *pcb, int cbctx)
```

**Arguments**

`pcb`: pointer to user callback object.

`cbctx`: COPT callback context, which is declared in public C header.

### Model::SetCoeff()

Set the coefficient of a variable in a linear constraint.

**Synopsis**

```
void SetCoeff(
    const Constraint &constr,
    const Var &var,
    double newVal)
```

**Arguments**

`constr`: The requested constraint.

`var`: The requested variable.

`newVal`: New coefficient.

### Model::SetCoeffs()

Set a list of coefficients in the model.

**Synopsis**

```
void SetCoeffs(
    const ConstrArray &constrs,
    const VarArray &vars,
    double *vals,
    int len)
```

**Arguments**

`constrs`: A list of constraints for coefficients to be set.

`vars`: A list of vars for coefficients to be set.

`vals`: New values for coefficients.

`len`: Length of values.

### Model::SetCost()

Set objective for model

**Synopsis**

```
void SetCost(
    int num,
    const int *list,
    const double *obj)
```

**Arguments**

`num`: number of terms in objective.

`list`: variable indexes in objective.

`obj`: corresponding coefficients in objective.

### Model::SetDblParam()

Set value of a COPT double parameter.

**Synopsis**

```
void SetDblParam(const char *szParam, double dVal)
```

**Arguments**

`szParam`: name of integer parameter.

`dVal`: double value.

### Model::SetDblParamN()

Set value of a double parameter of a multi-objective function.

**Synopsis**

```
void SetDblParamN(
    int idx,
    const char *szParam,
    double val)
```

**Arguments**

`idx`: index of a multi-objective function.

`szParam`: name of double parameter.

`val`: new value of double parameter.

### Model::SetIntParam()

Set value of a COPT integer parameter.

**Synopsis**

```
void SetIntParam(const char *szParam, int nVal)
```

**Arguments**

`szParam`: name of integer parameter.

`nVal`: integer value.

### Model::SetIntParamN()

Set value of an integer parameter of a multi-objective function.

**Synopsis**

```
void SetIntParamN(
    int idx,
    const char *szParam,
    int val)
```

**Arguments**

`idx`: index of a multi-objective function.

`szParam`: name of integer parameter.

`val`: new value of integer parameter.

### Model::SetLmiCoeff()

Set the coefficient matrix of a variable in LMI constraint.

**Synopsis**

```
void SetLmiCoeff(
    const LmiConstraint &constr,
    const Var &var,
    const SymMatrix &mat)
```

**Arguments**

`constr`: The desired LMI constraint.

`var`: The desired variable.

`mat`: new coefficient matrix.

### Model::SetLmiRhs()

Set constant matrix of LMI constraint.

**Synopsis**

```
void SetLmiRhs(const LmiConstraint &constr, const SymMatrix &mat)
```

**Arguments**

`constr`: The desired LMI constraint.

`mat`: new constant matrix.

### Model::SetLpSolution()

Set LP solution.

**Synopsis**

```
void SetLpSolution(
    double *pValue,
    double *pSlack,
    double *pRowDual,
    double *pRedCost)
```

**Arguments**

`pValue`: double pointer to solution values.

`pSlack`: double pointer to slack values.

`pRowDual`: double pointer to dual values.

`pRedCost`: double pointer to reduced costs.

### Model::SetMipStart()

Set initial values for variables of given number, starting from the first one.

**Synopsis**

```
void SetMipStart(int count, double *pVals)
```

**Arguments**

`count`: the number of variables to set.

`pVals`: pointer to initial values of variables.

### Model::SetMipStart()

Set initial value for the specified variable.

**Synopsis**

```
void SetMipStart(const Var &var, double val)
```

**Arguments**

`var`: an interested variable.

`val`: initial value of the variable.

### Model::SetMipStart()

Set initial values for an array of variables.

**Synopsis**

```
void SetMipStart(const VarArray &vars, double *pVals)
```

**Arguments**

`vars`: a list of interested variables.

`pVals`: pointer to initial values of variables.

### Model::SetNames()

Set names for given variables in model.

**Synopsis**

```
void SetNames(
    const VarArray &vars,
    const char *szNames,
    size_t len)
```

**Arguments**

`vars`: array object of variables.

`szNames`: name buffer for variables.

`len`: length of name buffer.

### Model::SetNames()

Set names for given constraints in model.

**Synopsis**

```
void SetNames(
    const ConstrArray &cons,
    const char *szNames,
    size_t len)
```

**Arguments**

`cons`: array object of constraints.

`szNames`: name buffer for constraints.

`len`: length of name buffer.

### Model::SetNames()

Set names for given general constraints in model.

**Synopsis**

```
void SetNames(
    const GenConstrArray &genConstrs,
    const char *szNames,
    size_t len)
```

**Arguments**

`genConstrs`: array object of general constraints.

`szNames`: name buffer for general constraints.

`len`: length of name buffer.

### Model::SetNames()

Set names for given affine cone constraints in model.

**Synopsis**

```
void SetNames(
    const AffineConeArray &cons,
    const char *szNames,
    size_t len)
```

**Arguments**

`cons`: array object of affine cone constraints.

`szNames`: name buffer for affine cone constraints.

`len`: length of name buffer.

### Model::SetNames()

Set names for given nonlinear constraints in model.

**Synopsis**

```
void SetNames(
    const NlConstrArray &cons,
    const char *szNames,
    size_t len)
```

**Arguments**

`cons`: array object of nonlinear constraints.

`szNames`: name buffer for nonlinear constraints.

`len`: length of name buffer.

### Model::SetNames()

Set names for given quadratic constraints in model.

**Synopsis**

```
void SetNames(
    const QConstrArray &cons,
    const char *szNames,
    size_t len)
```

**Arguments**

`cons`: array object of quadratic constraints.

`szNames`: name buffer for quadratic constraints.

`len`: length of name buffer.

### Model::SetNames()

Set names for given PSD variables in model.

**Synopsis**

```
void SetNames(
    const PsdVarArray &vars,
    const char *szNames,
    size_t len)
```

**Arguments**

`vars`: array object of PSD variables.

`szNames`: name buffer for PSD variables.

`len`: length of name buffer.

### Model::SetNames()

Set names for given PSD constraints in model.

**Synopsis**

```
void SetNames(
    const PsdConstrArray &cons,
    const char *szNames,
    size_t len)
```

**Arguments**

`cons`: array object of PSD constraints.

`szNames`: name buffer for PSD constraints.

`len`: length of name buffer.

### Model::SetNames()

Set names for given LMI constraints in model.

**Synopsis**

```
void SetNames(
    const LmiConstrArray &cons,
    const char *szNames,
    size_t len)
```

**Arguments**

`cons`: array object of LMI constraints.

`szNames`: name buffer for LMI constraints.

`len`: length of name buffer.

### Model::SetNlObjective()

Set nonlinear objective for model.

**Synopsis**

```
void SetNlObjective(const NlExpr &expr, int sense)
```

**Arguments**

`expr`: nonlinear expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

### Model::SetNlPrimalStart()

Given count, set initial values for variables of NLP from beginning.

**Synopsis**

```
void SetNlPrimalStart(int count, double *pVals)
```

**Arguments**

`count`: the number of variables to set.

`pVals`: pointer to initial values of variables.

### Model::SetNlPrimalStart()

Set initial value for the specified variable of NLP.

**Synopsis**

```
void SetNlPrimalStart(const Var &var, double val)
```

**Arguments**

`var`: an interested variable.

`val`: initial value of the variable.

### Model::SetNlPrimalStart()

Set initial values for an array of variables of NLP.

**Synopsis**

```
void SetNlPrimalStart(const VarArray &vars, double *pVals)
```

**Arguments**

`vars`: a list of interested variables.

`pVals`: pointer to initial values of variables.

### Model::SetObjConst()

Set objective constant.

**Synopsis**

```
void SetObjConst(double constant)
```

**Arguments**

`constant`: constant value to set.

### Model::SetObjective()

Set objective for model.

**Synopsis**

```
void SetObjective(const Expr &expr, int sense)
```

**Arguments**

`expr`: expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

### Model::SetObjectiveN()

Set a multi-objective function in model.

**Synopsis**

```
void SetObjectiveN(
    int idx,
    const Expr &expr,
    int sense,
    double priority,
    double weight,
    double abstol,
    double reltol)
```

**Arguments**

`idx`: index of a multi-objective function.

`expr`: linear expression of the multi-objective function.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

`priority`: an optional objective parameter. Default value is 0.0.

`weight`: an optional objective parameter. Default value is 1.0.

`abstol`: absolute tolerance is an optional objective parameter. Default value is 1e-6.

`reltol`: relative tolerance is an optional objective parameter. Default value is 0.0.

### Model::SetObjParamN()

Set value of objective parameter of a multi-objective function.

**Synopsis**

```
void SetObjParamN(
    int idx,
    const char *szParam,
    double val)
```

**Arguments**

`idx`: index of a multi-objective function.

`szParam`: name of objective parameter, including priority, weight, abstol and reltol.

`val`: new value of objective parameter.

### Model::SetObjSense()

Set objective sense for model.

**Synopsis**

```
void SetObjSense(int sense)
```

**Arguments**

`sense`: the objective sense.

### Model::SetPsdCoeff()

Set the coefficient matrix of a PSD variable in a PSD constraint.

**Synopsis**

```
void SetPsdCoeff(
    const PsdConstraint &constr,
    const PsdVar &var,
    const SymMatrix &mat)
```

**Arguments**

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

`mat`: new coefficient matrix.

### Model::SetPsdObjective()

Set PSD objective for model.

**Synopsis**

```
void SetPsdObjective(const PsdExpr &expr, int sense)
```

**Arguments**

`expr`: PSD expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

### Model::SetQuadCost()

Set quadratic objective of the model.

**Synopsis**

```
void SetQuadCost(
    int nQElem,
    const int *pMatRow,
    const int *pMatCol,
    const double *pMatElem)
```

**Arguments**

`nQElem`: number of Q objective elements.

`pMatRow`: row indices of Q objective elements.

`pMatCol`: column indices of Q objective elements.

`pMatElem`: nonzero elements of Q objective.

### Model::SetQuadObjective()

Set quadratic objective for model.

**Synopsis**

```
void SetQuadObjective(const QuadExpr &expr, int sense)
```

**Arguments**

`expr`: quadratic expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

### Model::SetSlackBasis()

Set slack basis to model.

**Synopsis**

```
void SetSlackBasis()
```

### Model::SetSolverLogCallback()

Set log callback for COPT.

**Synopsis**

```
void SetSolverLogCallback(ILogCallback *pcb)
```

**Arguments**

`pcb`: pointer to ILogCallback object.

### Model::SetSolverLogFile()

Set log file for COPT.

**Synopsis**

```
void SetSolverLogFile(const char *szLogFile)
```

**Arguments**

`szLogFile`: log file name.

### Model::Solve()

Solve the model as MIP.

**Synopsis**

```
void Solve()
```

### Model::SolveLp()

Solve the model as LP.

**Synopsis**

```
void SolveLp()
```

### Model::Tune()

Tune model.

**Synopsis**

```
void Tune()
```

### Model::Update()

Update current model, including numerical range and removed items.

**Synopsis**

```
void Update()
```

### Model::Write()

Output problem, solution, basis, MIP start or modified COPT parameters to file.

**Synopsis**

```
void Write(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WriteBasis()

Output optimal basis to a file of type '.bas'.

**Synopsis**

```
void WriteBasis(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WriteBin()

Output problem to a file as COPT binary format.

**Synopsis**

```
void WriteBin(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WriteIIS()

Output IIS to file.

**Synopsis**

```
void WriteIIS(const char *szFileName)
```

**Arguments**

`szFileName`: Output file name.

### Model::WriteJsonSol()

Output solution to a file of type '.json'.

**Synopsis**

```
void WriteJsonSol(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WriteLp()

Output problem to a file as LP format.

**Synopsis**

```
void WriteLp(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WriteMps()

Output problem to a file as MPS format.

**Synopsis**

```
void WriteMps(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WriteMpsStr()

Output problem to a buffer as MPS format.

**Synopsis**

```
ProbBuffer WriteMpsStr()
```

**Return**

output problem buffer.

### Model::WriteMst()

Output MIP start information to a file of type '.mst'.

**Synopsis**

```
void WriteMst(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WriteNL()

Output problem to a file as NL format.

**Synopsis**

```
void WriteNL(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WriteOrd()

Output branching order to file.

**Synopsis**

```
void WriteOrd(const char *szFileName)
```

**Arguments**

`szFileName`: Output file name.

### Model::WriteParam()

Output modified COPT parameters to a file of type '.par'.

**Synopsis**

```
void WriteParam(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

### Model::WritePoolSol()

Output selected pool solution to a file of type '.sol'.

**Synopsis**

```
void WritePoolSol(int iSol, const char *szFileName)
```

**Arguments**

`iSol`: index of pool solution.

`szFileName`: an output file name.

### Model::WriteRelax()

Output feasibility relaxation problem to file.

**Synopsis**

```
void WriteRelax(const char *szFileName)
```

**Arguments**

`szFileName`: Output file name.

### Model::WriteSol()

Output solution to a file of type '.sol'.

**Synopsis**

```
void WriteSol(const char *szFileName)
```

**Arguments**

`szFileName`: an output file name.

#### Model::WriteTuneParam()

Output specified tuned parameters to a file of type '.par'.

**Synopsis**

```
void WriteTuneParam(int idx, const char *szFileName)
```

**Arguments**

`idx`: Index of tuned parameters.

`szFileName`: Output file name.

### 25.5.4 Var

COPT variable object. Variables are always associated with a particular model. User creates a variable object by adding a variable to a model, rather than by using constructor of Var class.

#### Var::Get()

Get information of the variable. Support information of "Value", "RedCost", "PrimalRay", "LB", "UB" and "Obj".

**Synopsis**

```
double Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of information.

**Return**

value of information.

#### Var::GetBasis()

Get basis status of this variable.

**Synopsis**

```
int GetBasis()
```

**Return**

basis status.

#### Var::GetIdx()

Get index of the variable.

**Synopsis**

```
inline int GetIdx()
```

**Return**

variable index.

### Var::GetLowerIIS()

Get IIS status for lower bound of the variable.

**Synopsis**

```
int GetLowerIIS()
```

**Return**

IIS status.

### Var::GetName()

Get name of the variable.

**Synopsis**

```
const char *GetName()
```

**Return**

variable name.

### Var::GetType()

Get type of the variable.

**Synopsis**

```
char GetType()
```

**Return**

variable type.

### Var::GetUpperIIS()

Get IIS status for upper bound of the variable.

**Synopsis**

```
int GetUpperIIS()
```

**Return**

IIS status.

### Var::Remove()

Remove variable from model.

**Synopsis**

```
void Remove()
```

#### Var::Set()

Set information value of the variable. Support information of "LB", "UB" and "Obj".

**Synopsis**

```
void Set(const char *szInfo, double value)
```

**Arguments**

`szInfo`: name of information.

`value`: new information value.

#### Var::SetName()

Set name of the variable.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: variable name.

#### Var::SetType()

Set type of the variable.

**Synopsis**

```
void SetType(char type)
```

**Arguments**

`type`: variable type.

### 25.5.5 VarArray

COPT variable array object. To store and access a set of C++ *Var* objects, Cardinal Optimizer provides C++ VarArray class, which defines the following methods.

#### VarArray::GetVar()

Get i-th variable object.

**Synopsis**

```
Var &GetVar(int i)
```

**Arguments**

`i`: index of the variable.

**Return**

variable object with index i.

#### VarArray::PushBack()

Add a variable object to variable array.

**Synopsis**

```
void PushBack(const Var &var)
```

**Arguments**

`var`: a variable object.

#### VarArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for variable object.

#### VarArray::Size()

Get the number of variable objects.

**Synopsis**

```
int Size()
```

**Return**

number of variable objects.

### 25.5.6 Expr

COPT linear expression object. A linear expression consists of a constant term, a list of terms of variables and associated coefficients. Linear expressions are used to build constraints.

#### Expr::Expr()

Constructor of a constant linear expression.

**Synopsis**

```
Expr(double constant)
```

**Arguments**

`constant`: constant value in expression object.

#### Expr::Expr()

Constructor of a linear expression with one term.

**Synopsis**

```
Expr(const Var &var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

#### Expr::AddConstant()

Add constant for the expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### Expr::AddExpr()

Add an expression to self.

**Synopsis**

```
void AddExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

#### Expr::AddTerm()

Add a term to expression object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: a variable for new term.

`coeff`: coefficient for new term.

#### Expr::AddTerms()

Add terms to expression object.

**Synopsis**

```
int AddTerms(
    const VarArray &vars,
    double *pCoeff,
    int len)
```

**Arguments**

`vars`: variables for added terms.

`pCoeff`: coefficient array for added terms.

`len`: length of coefficient array.

**Return**

number of added terms.

### Expr::Clear()

Clear linear expression object.

**Synopsis**

```
void Clear()
```

### Expr::Clone()

Deep copy linear expression object.

**Synopsis**

```
Expr Clone()
```

**Return**

cloned expression object.

### Expr::Evaluate()

Evaluate linear expression after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of linear expression.

### Expr::GetCoeff()

Get coefficient from the i-th term in expression.

**Synopsis**

```
double GetCoeff(int i)
```

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term in expression object.

### Expr::GetConstant()

Get constant in expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in expression.

### Expr::GetVar()

Get variable from the i-th term in expression.

**Synopsis**

```
Var &GetVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

variable of the i-th term in expression object.

### Expr::operator*=()

Multiply a constant to self.

**Synopsis**

```
void operator*=(double c)
```

**Arguments**

`c`: constant multiplier.

### Expr::operator*()

Multiply constant and return new expression.

**Synopsis**

```
Expr operator*(double c)
```

**Arguments**

`c`: constant multiplier.

**Return**

result expression.

### Expr::operator*()

Multiply a variable and return new quadratic expression object.

**Synopsis**

```
QuadExpr operator*(const Var &var)
```

**Arguments**

`var`: variable object.

**Return**

result quadratic expression.

### Expr::operator*()

Multiply a linear expression and return new quadratic expression object.

**Synopsis**

```
QuadExpr operator*(const Expr &other)
```

**Arguments**

`other`: linear expression object.

**Return**

result quadratic expression.

### Expr::operator/()

Divided by a constant and return new expression.

**Synopsis**

```
Expr operator/(double c)
```

**Arguments**

`c`: constant divisor.

**Return**

result expression.

### Expr::operator/()

Divided by a variable and return new nonlinear expression.

**Synopsis**

```
NlExpr operator/(const Var &var)
```

**Arguments**

`var`: a variable as divisor.

**Return**

result nonlinear expression.

### Expr::operator/()

Divided by a linear expression and return new nonlinear expression.

**Synopsis**

```
NlExpr operator/(const Expr &other)
```

**Arguments**

`other`: a linear expression as divisor.

**Return**

result nonlinear expression.

### Expr::operator+=()

Add an expression to self.

**Synopsis**

```
void operator+=(const Expr &expr)
```

**Arguments**

`expr`: expression to be added.

### Expr::operator+()

Add expression and return new expression.

**Synopsis**

```
Expr operator+(const Expr &other)
```

**Arguments**

`other`: other expression to add.

**Return**

result expression.

### Expr::operator-=()

Subtract an expression from self.

**Synopsis**

```
void operator-=(const Expr &expr)
```

**Arguments**

`expr`: expression to be subtracted.

### Expr::operator-()

Subtract expression and return new expression.

**Synopsis**

```
Expr operator-(const Expr &other)
```

**Arguments**

`other`: other expression to subtract.

**Return**

result expression.

### Expr::Remove()

Remove i-th term from expression object.

**Synopsis**

```
void Remove(int i)
```

**Arguments**

`i`: index of the term to be removed.

### Expr::Remove()

Remove the term associated with variable from expression.

**Synopsis**

```
void Remove(const Var &var)
```

**Arguments**

`var`: a variable whose term should be removed.

### Expr::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(size_t n)
```

**Arguments**

`n`: minimum capacity for linear expression object.

### Expr::SetCoeff()

Set coefficient for the i-th term in expression.

**Synopsis**

```
void SetCoeff(int i, double val)
```

**Arguments**

`i`: index of the term.

`val`: coefficient of the term.

#### Expr::SetConstant()

Set constant for the expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### Expr::Size()

Get number of terms in expression.

**Synopsis**

```
size_t Size()
```

**Return**

number of terms.

### 25.5.7 Constraint

COPT constraint object. Constraints are always associated with a particular model. User creates a constraint object by adding a constraint to a model, rather than by using constructor of Constraint class.

#### Constraint::ComputeShadowPriceRange()

Compute the shadow price range for a given constraint.

**Synopsis**

```
void ComputeShadowPriceRange(
    int ifRhs,
    double *pLower,
    double *pUpper)
```

**Arguments**

`ifRhs`: for range constraint, whether to use RHS for computing shadow price.

`pLower`: output lower bound of the shadow price.

`pUpper`: output upper bound of the shadow price.

#### Constraint::Get()

Get information value of the constraint. Support information of "Dual", "Slack", "DualFarkas", "LB" and "UB".

**Synopsis**

```
double Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of the information being queried.

**Return**

value of information.

### Constraint::GetBasis()

Get basis status of this constraint.

**Synopsis**

```
int GetBasis()
```

**Return**

basis status.

### Constraint::GetIdx()

Get index of the constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

the index of the constraint.

### Constraint::GetLowerIIS()

Get IIS status for lower bound of the constraint.

**Synopsis**

```
int GetLowerIIS()
```

**Return**

IIS status.

### Constraint::GetName()

Get name of the constraint.

**Synopsis**

```
const char *GetName()
```

**Return**

the name of the constraint.

### Constraint::GetUpperIIS()

Get IIS status for upper bound of the constraint.

**Synopsis**

```
int GetUpperIIS()
```

**Return**

IIS status.

#### Constraint::Remove()

Remove this constraint from model.

**Synopsis**

```
void Remove()
```

#### Constraint::Set()

Set information value of the constraint. Support information of "LB" and "UB".

**Synopsis**

```
void Set(const char *szInfo, double value)
```

**Arguments**

`szInfo`: name of the information.

`value`: new information value.

#### Constraint::SetName()

Set name for the constraint.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: the name to set.

### 25.5.8 ConstrArray

COPT constraint array object. To store and access a set of C++ *Constraint* objects, Cardinal Optimizer provides C++ ConstrArray class, which defines the following methods.

#### ConstrArray::GetConstr()

Get i-th constraint object.

**Synopsis**

```
Constraint &GetConstr(int i)
```

**Arguments**

`i`: index of the constraint.

**Return**

constraint object with index i.

#### ConstrArray::PushBack()

Add a constraint object to constraint array.

**Synopsis**

```
void PushBack(const Constraint &constr)
```

**Arguments**

`constr`: a constraint object.

#### ConstrArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for Constraint object.

#### ConstrArray::Size()

Get the number of constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of constraint objects.

### 25.5.9 ConstrBuilder

COPT constraint builder object. To help building a constraint, given a linear expression, constraint sense and right-hand side value, Cardinal Optimizer provides C++ ConstrBuilder class, which defines the following methods.

#### ConstrBuilder::GetExpr()

Get expression associated with constraint.

**Synopsis**

```
const Expr &GetExpr()
```

**Return**

expression object.

### ConstrBuilder::GetRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of the constraint.

### ConstrBuilder::GetSense()

Get sense associated with constraint.

**Synopsis**

```
char GetSense()
```

**Return**

constraint sense.

### ConstrBuilder::Set()

Set detail of a constraint to its builder object.

**Synopsis**

```
void Set(
    const Expr &expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of the constraint.

### ConstrBuilder::SetRange()

Set a range constraint to its builder.

**Synopsis**

```
void SetRange(const Expr &expr, double range)
```

**Arguments**

`expr`: expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must be greater than 0.

### 25.5.10 ConstrBuilderArray

COPT constraint builder array object. To store and access a set of C++ *ConstrBuilder* objects, Cardinal Optimizer provides C++ ConstrBuilderArray class, which defines the following methods.

#### ConstrBuilderArray::GetBuilder()

Get i-th constraint builder object.

**Synopsis**

```
ConstrBuilder &GetBuilder(int i)
```

**Arguments**

`i`: index of the constraint builder.

**Return**

constraint builder object with index i.

#### ConstrBuilderArray::PushBack()

Add a constraint builder object to constraint builder array.

**Synopsis**

```
void PushBack(const ConstrBuilder &builder)
```

**Arguments**

`builder`: a constraint builder object.

#### ConstrBuilderArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for constraint builder object.

#### ConstrBuilderArray::Size()

Get the number of constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of constraint builder objects.

### 25.5.11 Column

COPT column object. A column consists of a list of constraints and associated coefficients. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients.

#### Column::Column()

Constructor of column.

**Synopsis**

```
Column()
```

#### Column::AddColumn()

Add a column to self.

**Synopsis**

```
void AddColumn(const Column &col, double mult)
```

**Arguments**

`col`: column object to be added.

`mult`: multiply constant.

#### Column::AddTerm()

Add a term to column object.

**Synopsis**

```
void AddTerm(const Constraint &constr, double coeff)
```

**Arguments**

`constr`: a constraint for new term.

`coeff`: coefficient for new term.

#### Column::AddTerms()

Add terms to column object.

**Synopsis**

```
int AddTerms(
    const ConstrArray &constrs,
    double *pCoeff,
    int len)
```

**Arguments**

`constrs`: constraints for added terms.

`pCoeff`: coefficients for added terms.

`len`: number of terms to be added.

**Return**

number of added terms.

### Column::Clear()

Clear all terms.

**Synopsis**

```
void Clear()
```

### Column::Clone()

Deep copy column object.

**Synopsis**

```
Column Clone()
```

**Return**

cloned column object.

### Column::GetCoeff()

Get coefficient from the i-th term in column object.

**Synopsis**

```
double GetCoeff(int i)
```

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term in column object.

### Column::GetConstr()

Get constraint from the i-th term in column object.

**Synopsis**

```
Constraint GetConstr(int i)
```

**Arguments**

`i`: index of the term.

**Return**

constraint of the i-th term in column object.

### Column::Remove()

Remove i-th term from column object.

**Synopsis**

```
void Remove(int i)
```

**Arguments**

`i`: index of the term to be removed.

#### Column::Remove()

Remove the term associated with constraint from column object.

**Synopsis**

```
bool Remove(const Constraint &constr)
```

**Arguments**

`constr`: a constraint whose term should be removed.

**Return**

true if constraint exits in column object.

#### Column::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for Column object.

#### Column::Size()

Get number of terms in column object.

**Synopsis**

```
int Size()
```

**Return**

number of terms.

### 25.5.12 ColumnArray

COPT column array object. To store and access a set of C++ *Column* objects, Cardinal Optimizer provides C++ ColumnArray class, which defines the following methods.

#### ColumnArray::Clear()

Clear all column objects.

**Synopsis**

```
void Clear()
```

#### ColumnArray::GetColumn()

Get i-th column object.

**Synopsis**

```
Column &GetColumn(int i)
```

**Arguments**

`i`: index of the column.

**Return**

column object with index i.

#### ColumnArray::PushBack()

Add a column object to column array.

**Synopsis**

```
void PushBack(const Column &col)
```

**Arguments**

`col`: a column object.

#### ColumnArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for linear expression object.

#### ColumnArray::Size()

Get the number of column objects.

**Synopsis**

```
int Size()
```

**Return**

number of column objects.

### 25.5.13 Sos

COPT SOS constraint object. SOS constraints are always associated with a particular model. User creates an SOS constraint object by adding an SOS constraint to a model, rather than by using constructor of Sos class.

An SOS constraint can be type 1 or 2 (`COPT_SOS_TYPE1` or `COPT_SOS_TYPE2`).

#### Sos::GetIdx()

Get the index of SOS constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of SOS constraint.

#### Sos::GetIIS()

Get IIS status of the SOS constraint.

**Synopsis**

```
int GetIIS()
```

**Return**

IIS status.

#### Sos::Remove()

Remove the SOS constraint from model.

**Synopsis**

```
void Remove()
```

### 25.5.14 SosArray

COPT SOS constraint array object. To store and access a set of C++ *Sos* objects, Cardinal Optimizer provides C++ SosArray class, which defines the following methods.

#### SosArray::GetSos()

Get i-th SOS constraint object.

**Synopsis**

```
Sos &GetSos(int i)
```

**Arguments**

`i`: index of the SOS constraint.

**Return**

SOS constraint object with index i.

#### SosArray::PushBack()

Add a SOS constraint object to SOS constraint array.

**Synopsis**

```
void PushBack(const Sos &sos)
```

**Arguments**

`sos`: a SOS constraint object.

#### SosArray::Size()

Get the number of SOS constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of SOS constraint objects.

### 25.5.15 SosBuilder

COPT SOS constraint builder object. To help building an SOS constraint, given the SOS type, a set of variables and associated weights, Cardinal Optimizer provides C++ SosBuilder class, which defines the following methods.

#### SosBuilder::GetSize()

Get number of terms in SOS constraint.

**Synopsis**

```
int GetSize()
```

**Return**

number of terms.

#### SosBuilder::GetType()

Get type of SOS constraint.

**Synopsis**

```
int GetType()
```

**Return**

type of SOS constraint.

### SosBuilder::GetVar()

Get variable from the i-th term in SOS constraint.

**Synopsis**

```
Var GetVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

variable of the i-th term in SOS constraint.

### SosBuilder::GetVars()

Get variables of all terms in SOS constraint.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variable array object.

### SosBuilder::GetWeight()

Get weight from the i-th term in SOS constraint.

**Synopsis**

```
double GetWeight(int i)
```

**Arguments**

`i`: index of the term.

**Return**

weight of the i-th term in SOS constraint.

### SosBuilder::GetWeights()

Get weights of all terms in SOS constraint.

**Synopsis**

```
double GetWeights()
```

**Return**

pointer to array of weights.

#### SosBuilder::Set()

Set variables and weights of SOS constraint.

**Synopsis**

```
void Set(
    const VarArray &vars,
    const double *pWeights,
    int type)
```

**Arguments**

`vars`: variable array object.

`pWeights`: pointer to array of weights.

`type`: type of SOS constraint.

### 25.5.16 SosBuilderArray

COPT SOS constraint builder array object. To store and access a set of C++ *SosBuilder* objects, Cardinal Optimizer provides C++ SosBuilderArray class, which defines the following methods.

#### SosBuilderArray::GetBuilder()

Get i-th SOS constraint builder object.

**Synopsis**

```
SosBuilder &GetBuilder(int i)
```

**Arguments**

`i`: index of the SOS constraint builder.

**Return**

SOS constraint builder object with index i.

#### SosBuilderArray::PushBack()

Add a SOS constraint builder object to SOS constraint builder array.

**Synopsis**

```
void PushBack(const SosBuilder &builder)
```

**Arguments**

`builder`: a SOS constraint builder object.

#### SosBuilderArray::Size()

Get the number of SOS constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of SOS constraint builder objects.

### 25.5.17 GenConstr

COPT general constraint object. General constraints are always associated with a particular model. User creates a general constraint object by adding a general constraint to a model, rather than by using constructor of GenConstr class.

#### GenConstr::GetIdx()

Get the index of the general constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of the general constraint.

#### GenConstr::GetIIS()

Get IIS status of the general constraint.

**Synopsis**

```
int GetIIS()
```

**Return**

IIS status.

#### GenConstr::GetName()

Get name of general constraint.

**Synopsis**

```
const char *GetName()
```

**Return**

the name of general constraint.

#### GenConstr::Remove()

Remove the general constraint from model.

**Synopsis**

```
void Remove()
```

#### GenConstr::SetName()

Set name of general constraint.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: new name to set.

### 25.5.18 GenConstrArray

COPT general constraint array object. To store and access a set of C++ *GenConstr* objects, Cardinal Optimizer provides C++ GenConstrArray class, which defines the following methods.

#### GenConstrArray::GetGenConstr()

Get i-th general constraint object.

**Synopsis**

```
GenConstr &GetGenConstr(int i)
```

**Arguments**

`i`: index of the general constraint.

**Return**

general constraint object with index i.

#### GenConstrArray::PushBack()

Add a general constraint object to general constraint array.

**Synopsis**

```
void PushBack(const GenConstr &constr)
```

**Arguments**

`constr`: a general constraint object.

#### GenConstrArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for general constraint objects.

#### GenConstrArray::Size()

Get the number of general constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of general constraint objects.

### 25.5.19 GenConstrBuilder

COPT general constraint builder object. To help building a general constraint, given a binary variable and associated value, a linear expression and constraint sense, Cardinal Optimizer provides C++ GenConstrBuilder class, which defines the following methods.

#### GenConstrBuilder::GetBinVal()

Get binary value associated with general constraint.

**Synopsis**

```
int GetBinVal()
```

**Return**

binary value.

#### GenConstrBuilder::GetBinVar()

Get binary variable associated with general constraint.

**Synopsis**

```
Var GetBinVar()
```

**Return**

binary vaiable object.

### GenConstrBuilder::GetExpr()

Get expression associated with general constraint.

**Synopsis**

```
const Expr &GetExpr()
```

**Return**

expression object.

### GenConstrBuilder::GetIndType()

Get type of general constraint.

**Synopsis**

```
int GetIndType()
```

**Return**

type of GenConstr, COPT_INDICATOR_IF, COPT_INDICATOR_ONLYIF and COPT_INDICATOR_IFANDONLYIF.

### GenConstrBuilder::GetSense()

Get sense associated with general constraint.

**Synopsis**

```
char GetSense()
```

**Return**

constraint sense.

### GenConstrBuilder::Set()

Set binary variable, binary value, expression, sense and type of general constraint.

**Synopsis**

```
void Set(
    Var bvar,
    int bval,
    const Expr &expr,
    char sense,
    int type)
```

**Arguments**

`bvar`: binary variable.

`bval`: binary value.

`expr`: expression object.

`sense`: general constraint sense.

`type`: type of general constraint with default value of COPT_INDICATOR_IF.

### 25.5.20 GenConstrBuilderArray

COPT general constraint builder array object. To store and access a set of C++ *GenConstrBuilder* objects, Cardinal Optimizer provides C++ GenConstrBuilderArray class, which defines the following methods.

#### GenConstrBuilderArray::GetBuilder()

Get i-th general constraint builder object.

**Synopsis**

```
GenConstrBuilder &GetBuilder(int i)
```

**Arguments**

`i`: index of the general constraint builder.

**Return**

general constraint builder object with index i.

#### GenConstrBuilderArray::PushBack()

Add a general constraint builder object to general constraint builder array.

**Synopsis**

```
void PushBack(const GenConstrBuilder &builder)
```

**Arguments**

`builder`: a general constraint builder object.

#### GenConstrBuilderArray::Size()

Get the number of general constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of general constraint builder objects.

### 25.5.21 Cone

COPT cone constraint object. Cone constraints are always associated with a particular model. User creates a cone constraint object by adding a cone constraint to a model, rather than by using constructor of Cone class.

A cone constraint can be regular or rotated (`COPT_CONE_QUAD` or `COPT_CONE_RQUAD`).

#### Cone::GetIdx()

Get the index of a cone constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of a cone constraint.

#### Cone::Remove()

Remove the cone constraint from model.

**Synopsis**

```
void Remove()
```

### 25.5.22 ConeArray

COPT cone constraint array object. To store and access a set of C++ *Cone* objects, Cardinal Optimizer provides C++ ConeArray class, which defines the following methods.

#### ConeArray::GetCone()

Get i-th cone constraint object.

**Synopsis**

```
Cone &GetCone(int i)
```

**Arguments**

`i`: index of the cone constraint.

**Return**

cone constraint object with index i.

#### ConeArray::PushBack()

Add a cone constraint object to cone constraint array.

**Synopsis**

```
void PushBack(const Cone &cone)
```

**Arguments**

`cone`: a cone constraint object.

#### ConeArray::Size()

Get the number of cone constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of cone constraint objects.

### 25.5.23 ConeBuilder

COPT cone constraint builder object. To help building a cone constraint, given the cone type and a set of variables, Cardinal Optimizer provides C++ ConeBuilder class, which defines the following methods.

#### ConeBuilder::GetSize()

Get number of vars in a cone constraint.

**Synopsis**

```
int GetSize()
```

**Return**

number of vars.

#### ConeBuilder::GetType()

Get type of a cone constraint.

**Synopsis**

```
int GetType()
```

**Return**

type of a cone constraint.

#### ConeBuilder::GetVar()

Get the i-th variable in a cone constraint.

**Synopsis**

```
Var GetVar(int i)
```

**Arguments**

`i`: index of vars in a cone constraint.

**Return**

the i-th variable in a cone constraint.

#### ConeBuilder::GetVars()

Get all variables in a cone constraint.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variable array object.

#### ConeBuilder::Set()

Set variables of a cone constraint.

**Synopsis**

```
void Set(const VarArray &vars, int type)
```

**Arguments**

`vars`: variable array object.

`type`: type of cone constraint.

### 25.5.24 ConeBuilderArray

COPT cone constraint builder array object. To store and access a set of C++ *ConeBuilder* objects, Cardinal Optimizer provides C++ ConeBuilderArray class, which defines the following methods.

#### ConeBuilderArray::GetBuilder()

Get i-th cone constraint builder object.

**Synopsis**

```
ConeBuilder &GetBuilder(int i)
```

**Arguments**

`i`: index of the cone constraint builder.

**Return**

cone constraint builder object with index i.

#### ConeBuilderArray::PushBack()

Add a cone constraint builder object to cone constraint builder array.

**Synopsis**

```
void PushBack(const ConeBuilder &builder)
```

**Arguments**

`builder`: a cone constraint builder object.

#### ConeBuilderArray::Size()

Get the number of cone constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of cone constraint builder objects.

### 25.5.25 ExpCone

COPT exponential cone constraint object. ExpCone constraints are always associated with a particular model. User creates an exponential cone constraint object by adding an expcone constraint to a model, rather than by using constructor of ExpCone class.

#### ExpCone::GetIdx()

Get the index of an exponential cone constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of an exponential cone constraint.

#### ExpCone::Remove()

Remove the exponential cone constraint from model.

**Synopsis**

```
void Remove()
```

### 25.5.26 ExpConeArray

COPT exponential cone constraint array object. To store and access a set of C++ *ExpCone* objects, Cardinal Optimizer provides C++ ExpConeArray class, which defines the following methods.

#### ExpConeArray::GetCone()

Get i-th exponential cone constraint object.

**Synopsis**

```
ExpCone &GetCone(int i)
```

**Arguments**

`i`: index of the exponential cone constraint.

**Return**

exponential cone constraint object with index i.

#### ExpConeArray::PushBack()

Add an exponential cone constraint object to exponential cone constraint array.

**Synopsis**

```
void PushBack(const ExpCone &cone)
```

**Arguments**

`cone`: an exponential constraint object.

#### ExpConeArray::Size()

Get the number of exponential cone constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of exponential cone constraint objects.

### 25.5.27 ExpConeBuilder

COPT exponential cone constraint builder object. To help build an exponential cone constraint, given the expcone type and a set of variables, Cardinal Optimizer provides C++ ExpConeBuilder class, which defines the following methods.

#### ExpConeBuilder::GetSize()

Get number of vars in an exponential cone constraint.

**Synopsis**

```
int GetSize()
```

**Return**

number of vars.

#### ExpConeBuilder::GetType()

Get type of an exponential cone constraint.

**Synopsis**

```
int GetType()
```

**Return**

type of an exponential cone constraint.

#### ExpConeBuilder::GetVar()

Get the i-th variable in an exponential cone constraint.

**Synopsis**

```
Var GetVar(int i)
```

**Arguments**

`i`: index of vars in an exponential cone constraint.

**Return**

the i-th variable in an exponential cone constraint.

#### ExpConeBuilder::GetVars()

Get all variables in an exponential cone constraint.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variable array object.

#### ExpConeBuilder::Set()

Set variables of an exponential cone constraint.

**Synopsis**

```
void Set(const VarArray &vars, int type)
```

**Arguments**

`vars`: variable array object.

`type`: type of exponential cone constraint.

### 25.5.28 ExpConeBuilderArray

COPT exponential cone constraint builder array object. To store and access a set of C++ *ExpConeBuilder* objects, Cardinal Optimizer provides C++ ExpConeBuilderArray class, which defines the following methods.

#### ExpConeBuilderArray::GetBuilder()

Get i-th exponential cone constraint builder object.

**Synopsis**

```
ExpConeBuilder &GetBuilder(int i)
```

**Arguments**

`i`: index of the exponential cone constraint builder.

**Return**

exponential cone constraint builder object with index i.

#### ExpConeBuilderArray::PushBack()

Add an exponential cone constraint builder object to exponential cone constraint builder array.

**Synopsis**

```
void PushBack(const ExpConeBuilder &builder)
```

**Arguments**

`builder`: an exponential cone constraint builder object.

#### ExpConeBuilderArray::Size()

Get the number of exponential cone constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of exponential cone constraint builder objects.

### 25.5.29 AffineCone

The *AffineCone* class in COPT encapsulates operations related to affine cone constraints. The following methods are provided:

#### AffineCone::GetIdx()

Get the index of an affine cone constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of an affine cone constraint.

#### AffineCone::GetName()

Get name of affine cone constraint.

**Synopsis**

```
const char *GetName()
```

**Return**

name of affine cone constraint.

#### AffineCone::Remove()

Remove the affine cone constraint from model.

**Synopsis**

```
void Remove()
```

#### AffineCone::SetName()

Set name of affine cone constraint.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: name of affine cone constraint.

### 25.5.30 AffineConeArray

To facilitate user operations on a group of C++ *AffineCone* objects, the C++ interface of COPT introduces the *AffineConeArray* class. The following methods are provided:

#### AffineConeArray::GetCone()

Get i-th affine cone constraint object.

**Synopsis**

```
AffineCone &GetCone(int i)
```

**Arguments**

`i`: index of the affine cone constraint.

**Return**

affine cone constraint object with index i.

#### AffineConeArray::PushBack()

Add an affine cone constraint object to affine cone constraint array.

**Synopsis**

```
void PushBack(const AffineCone &cone)
```

**Arguments**

`cone`: an affine constraint object.

#### AffineConeArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for affine cone constraint objects.

#### AffineConeArray::Size()

Get the number of affine cone constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of affine cone constraint objects.

### 25.5.31 AffineConeBuilder

The *AffineConeBuilder* class in COPT encapsulates the builder for constructing affine cone constraints. The following methods are provided:

#### AffineConeBuilder::GetExpr()

Get the i-th linear expression in an affine cone constraint.

**Synopsis**

```
const Expr &GetExpr(int i)
```

**Arguments**

`i`: index of linear expression in an affine cone constraint.

**Return**

the i-th linear expression in an affine cone constraint.

#### AffineConeBuilder::GetExprs()

Get all linear expressions in an affine cone constraint.

**Synopsis**

```
MLinExpr<1> GetExprs()
```

**Return**

MLinExpr object.

#### AffineConeBuilder::GetPsdExpr()

Get the i-th PSD expression in an affine cone constraint.

**Synopsis**

```
const PsdExpr &GetPsdExpr(int i)
```

**Arguments**

`i`: index of PSD expression in an affine cone constraint.

**Return**

the i-th PSD expression in an affine cone constraint.

#### AffineConeBuilder::GetPsdExprs()

Get all PSD expressions in an affine cone constraint.

**Synopsis**

```
const MPsdExpr<1> &GetPsdExprs()
```

**Return**

MPsdExpr object.

#### AffineConeBuilder::GetSize()

Get number of expressions in an affine cone constraint.

**Synopsis**

```
int GetSize()
```

**Return**

number of expressions.

#### AffineConeBuilder::GetType()

Get type of an affine cone constraint.

**Synopsis**

```
int GetType()
```

**Return**

type of an affine cone constraint.

#### AffineConeBuilder::HasPsdTerm()

Check whether affine cone has PSD terms.

**Synopsis**

```
bool HasPsdTerm()
```

**Return**

flag to indicate whether affine cone has PSD terms.

#### AffineConeBuilder::Set()

Set linear expressions of an affine cone constraint.

**Synopsis**

```
void Set(const MLinExpr<1> &exprs, int type)
```

**Arguments**

`exprs`: 1-dimensional MLinExpr object.

`type`: type of affine cone constraint.

#### AffineConeBuilder::Set()

Set PSD expressions of an affine cone constraint.

**Synopsis**

```
void Set(const MPsdExpr<1> &exprs, int type)
```

**Arguments**

`exprs`: 1-dimensional MPsdExpr object.

`type`: type of affine cone constraint.

### 25.5.32 AffineConeBuilderArray

To facilitate user operations on a group of C++ *AffineConeBuilder* objects, the C++ interface of COPT introduces the *AffineConeBuilderArray* class. The following methods are provided:

#### AffineConeBuilderArray::GetBuilder()

Get i-th affine cone constraint builder object.

**Synopsis**

```
AffineConeBuilder &GetBuilder(int i)
```

**Arguments**

`i`: index of the affine cone constraint builder.

**Return**

affine cone constraint builder object with index i.

#### AffineConeBuilderArray::PushBack()

Add an affine cone constraint builder object to affine cone constraint builder array.

**Synopsis**

```
void PushBack(const AffineConeBuilder &builder)
```

**Arguments**

`builder`: an affine cone constraint builder object.

#### AffineConeBuilderArray::Size()

Get the number of affine cone constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of affine cone constraint builder objects.

### 25.5.33 QuadExpr

COPT quadratic expression object. A quadratic expression consists of a linear expression, a list of variable pairs and associated coefficients of quadratic terms. Quadratic expressions are used to build quadratic constraints and objectives.

#### QuadExpr::QuadExpr()

Constructor of a quadratic expression with a constant.

**Synopsis**

```
QuadExpr(double constant)
```

**Arguments**

`constant`: constant value in quadratic expression object.

#### QuadExpr::QuadExpr()

Constructor of a quadratic expression with one term.

**Synopsis**

```
QuadExpr(const Var &var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

#### QuadExpr::QuadExpr()

Constructor of a quadratic expression with a linear expression.

**Synopsis**

```
QuadExpr(const Expr &expr)
```

**Arguments**

`expr`: input linear expression.

### QuadExpr::QuadExpr()

Constructor of a quadratic expression with two linear expression.

**Synopsis**

```
QuadExpr(const Expr &expr, const Var &var)
```

**Arguments**

`expr`: one linear expression.

`var`: another variable.

### QuadExpr::QuadExpr()

Constructor of a quadratic expression with two linear expression.

**Synopsis**

```
QuadExpr(const Expr &left, const Expr &right)
```

**Arguments**

`left`: one linear expression.

`right`: another linear expression.

### QuadExpr::AddConstant()

Add constant for the expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### QuadExpr::AddLinExpr()

Add a linear expression to self.

**Synopsis**

```
void AddLinExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### QuadExpr::AddQuadExpr()

Add a quadratic expression to self.

**Synopsis**

```
void AddQuadExpr(const QuadExpr &expr, double mult)
```

**Arguments**

`expr`: quadratic expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### QuadExpr::AddTerm()

Add a linear term to expression object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

### QuadExpr::AddTerm()

Add a quadratic term to expression object.

**Synopsis**

```
void AddTerm(
    const Var &var1,
    const Var &var2,
    double coeff)
```

**Arguments**

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

### QuadExpr::AddTerms()

Add linear terms to expression object.

**Synopsis**

```
int AddTerms(
    const VarArray &vars,
    double *pCoeff,
    int len)
```

**Arguments**

`vars`: variables for added linear terms.

`pCoeff`: coefficient array for added linear terms.

`len`: length of coefficient array.

**Return**

number of added linear terms.

### QuadExpr::AddTerms()

Add quadratic terms to expression object.

**Synopsis**

```
int AddTerms(
    const VarArray &vars1,
    const VarArray &vars2,
    double *pCoeff,
    int len)
```

**Arguments**

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`pCoeff`: coefficient array for added quadratic terms.

`len`: length of coefficient array.

**Return**

number of added quadratic terms.

### QuadExpr::Clear()

Clear quadratic expression object.

**Synopsis**

```
void Clear()
```

### QuadExpr::Clone()

Deep copy quadratic expression object.

**Synopsis**

```
QuadExpr Clone()
```

**Return**

cloned quadratic expression object.

### QuadExpr::Evaluate()

Evaluate quadratic expression after solving

**Synopsis**

```
double Evaluate()
```

**Return**

value of quadratic expression

#### QuadExpr::GetCoeff()

Get coefficient from the i-th term in quadratic expression.

**Synopsis**

```
double GetCoeff(int i)
```

**Arguments**

`i`: index of the quadratic term.

**Return**

coefficient of the i-th quadratic term in quadratic expression object.

#### QuadExpr::GetConstant()

Get constant in quadratic expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in quadratic expression.

#### QuadExpr::GetLinExpr()

Get linear expression in quadratic expression.

**Synopsis**

```
Expr &GetLinExpr()
```

**Return**

linear expression object.

#### QuadExpr::GetVar1()

Get the first variable from the i-th term in quadratic expression.

**Synopsis**

```
Var &GetVar1(int i)
```

**Arguments**

`i`: index of the term.

**Return**

the first variable of the i-th term in quadratic expression object.

### QuadExpr::GetVar2()

Get the second variable from the i-th term in quadratic expression.

**Synopsis**

```
Var &GetVar2(int i)
```

**Arguments**

`i`: index of the term.

**Return**

the second variable of the i-th term in quadratic expression object.

### QuadExpr::operator*=()

Multiply a constant to self.

**Synopsis**

```
void operator*=(double c)
```

**Arguments**

`c`: constant multiplier.

### QuadExpr::operator*()

Multiply constant and return new expression.

**Synopsis**

```
QuadExpr operator*(double c)
```

**Arguments**

`c`: constant multiplier.

**Return**

result expression.

### QuadExpr::operator*()

Multiply a variable and return new nonlinear expression.

**Synopsis**

```
NlExpr operator*(const Var &var)
```

**Arguments**

`var`: a variable as multiplier.

**Return**

result nonlinear expression.

### QuadExpr::operator*()

Multiply a linear expression and return new nonlinear expression.

**Synopsis**

```
NlExpr operator*(const Expr &expr)
```

**Arguments**

`expr`: a linear expression as multiplier.

**Return**

result nonlinear expression.

### QuadExpr::operator*()

Multiply a quadratic expression and return new nonlinear expression.

**Synopsis**

```
NlExpr operator*(const QuadExpr &expr)
```

**Arguments**

`expr`: a quadratic expression as multiplier.

**Return**

result nonlinear expression.

### QuadExpr::operator/()

Divided by a constant and return new quadratic expression.

**Synopsis**

```
QuadExpr operator/(double c)
```

**Arguments**

`c`: constant divisor.

**Return**

result expression.

### QuadExpr::operator/()

Divided by a variable and return new nonlinear expression.

**Synopsis**

```
NlExpr operator/(const Var &var)
```

**Arguments**

`var`: a variable as divisor.

**Return**

result nonlinear expression.

### QuadExpr::operator/()

Divided by a linear expression and return new nonlinear expression.

**Synopsis**

```
NlExpr operator/(const Expr &other)
```

**Arguments**

`other`: a linear expression as divisor.

**Return**

result nonlinear expression.

### QuadExpr::operator/()

Divided by a quadratic expression and return new nonlinear expression.

**Synopsis**

```
NlExpr operator/(const QuadExpr &other)
```

**Arguments**

`other`: a quadratic expression as divisor.

**Return**

result nonlinear expression.

### QuadExpr::operator+=()

Add an expression to self.

**Synopsis**

```
void operator+=(const QuadExpr &expr)
```

**Arguments**

`expr`: expression to be added.

### QuadExpr::operator+()

Add expression and return new expression.

**Synopsis**

```
QuadExpr operator+(const QuadExpr &other)
```

**Arguments**

`other`: other expression to add.

**Return**

result expression.

### QuadExpr::operator-=()

Subtract an expression from self.

**Synopsis**

```
void operator-=(const QuadExpr &expr)
```

**Arguments**

`expr`: expression to be subtracted.

### QuadExpr::operator-()

Subtract expression and return new expression.

**Synopsis**

```
QuadExpr operator-(const QuadExpr &other)
```

**Arguments**

`other`: other expression to subtract.

**Return**

result expression.

### QuadExpr::Remove()

Remove i-th term from expression object.

**Synopsis**

```
void Remove(int i)
```

**Arguments**

`i`: index of the term to be removed.

### QuadExpr::Remove()

Remove the term associated with variable from expression.

**Synopsis**

```
void Remove(const Var &var)
```

**Arguments**

`var`: a variable whose term should be removed.

### QuadExpr::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(size_t n)
```

**Arguments**

`n`: minimum capacity for quadratic expression object.

#### QuadExpr::SetCoeff()

Set coefficient of the i-th term in quadratic expression.

**Synopsis**

```
void SetCoeff(int i, double val)
```

**Arguments**

`i`: index of the quadratic term.

`val`: coefficient of the term.

#### QuadExpr::SetConstant()

Set constant for the expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### QuadExpr::Size()

Get number of terms in expression.

**Synopsis**

```
size_t Size()
```

**Return**

number of terms.

### 25.5.34 QConstraint

COPT quadratic constraint object. Quadratic constraints are always associated with a particular model. User creates a quadratic constraint object by adding a quadratic constraint to a model, rather than by using constructor of QConstraint class.

#### QConstraint::Get()

Get information value of the quadratic constraint. Support related quadratic information.

**Synopsis**

```
double Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of the information being queried.

**Return**

value of information.

#### QConstraint::GetIdx()

Get index of the quadratic constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

the index of the quadratic constraint.

#### QConstraint::GetName()

Get name of the quadratic constraint.

**Synopsis**

```
const char *GetName()
```

**Return**

the name of the quadratic constraint.

#### QConstraint::GetRhs()

Get rhs of quadratic constraint.

**Synopsis**

```
double GetRhs()
```

**Return**

rhs of quadratic constraint.

#### QConstraint::GetSense()

Get sense of quadratic constraint.

**Synopsis**

```
char GetSense()
```

**Return**

sense of quadratic constraint.

#### QConstraint::Remove()

Remove this quadratic constraint from model.

**Synopsis**

```
void Remove()
```

#### QConstraint::Set()

Set information value of the quadratic constraint. Support related quadratic information.

**Synopsis**

```
void Set(const char *szInfo, double value)
```

**Arguments**

`szInfo`: name of the information.

`value`: new information value.

#### QConstraint::SetName()

Set name of a quadratic constraint.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: the name to set.

#### QConstraint::SetRhs()

Set rhs of quadratic constraint.

**Synopsis**

```
void SetRhs(double rhs)
```

**Arguments**

`rhs`: rhs of quadratic constraint.

#### QConstraint::SetSense()

Set sense of quadratic constraint.

**Synopsis**

```
void SetSense(char sense)
```

**Arguments**

`sense`: sense of quadratic constraint.

### 25.5.35 QConstrArray

COPT quadratic constraint array object. To store and access a set of C++ *QConstraint* objects, Cardinal Optimizer provides C++ QConstrArray class, which defines the following methods.

#### QConstrArray::GetQConstr()

Get i-th quadratic constraint object.

**Synopsis**

```
QConstraint &GetQConstr(int idx)
```

**Arguments**

`idx`: index of the quadratic constraint.

**Return**

quadratic constraint object with index i.

#### QConstrArray::PushBack()

Add a quadratic constraint object to constraint array.

**Synopsis**

```
void PushBack(const QConstraint &constr)
```

**Arguments**

`constr`: a quadratic constraint object.

#### QConstrArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for quadratic constraint objects.

#### QConstrArray::Size()

Get the number of quadratic constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of quadratic constraint objects.

### 25.5.36 QConstrBuilder

COPT quadratic constraint builder object. To help building a quadratic constraint, given a quadratic expression, constraint sense and right-hand side value, Cardinal Optimizer provides C++ QConstrBuilder class, which defines the following methods.

#### QConstrBuilder::GetQuadExpr()

Get expression associated with quadratic constraint.

**Synopsis**

```
const QuadExpr &GetQuadExpr()
```

**Return**

quadratic expression object.

#### QConstrBuilder::GetSense()

Get sense associated with quadratic constraint.

**Synopsis**

```
char GetSense()
```

**Return**

quadratic constraint sense.

#### QConstrBuilder::Set()

Set detail of a quadratic constraint to its builder object.

**Synopsis**

```
void Set(
    const QuadExpr &expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the quadratic constraint.

`sense`: quadratic constraint sense.

`rhs`: constant of right side of quadratic constraint.

### 25.5.37 QConstrBuilderArray

COPT quadratic constraint builder array object. To store and access a set of C++ *QConstrBuilder* objects, Cardinal Optimizer provides C++ QConstrBuilderArray class, which defines the following methods.

#### QConstrBuilderArray::GetBuilder()

Get i-th quadratic constraint builder object.

**Synopsis**

```
QConstrBuilder &GetBuilder(int idx)
```

**Arguments**

`idx`: index of the quadratic constraint builder.

**Return**

quadratic constraint builder object with index i.

#### QConstrBuilderArray::PushBack()

Add a quadratic constraint builder object to quadratic constraint builder array.

**Synopsis**

```
void PushBack(const QConstrBuilder &builder)
```

**Arguments**

`builder`: a quadratic constraint builder object.

#### QConstrBuilderArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for quadratic constraint builder object.

#### QConstrBuilderArray::Size()

Get the number of quadratic constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of quadratic constraint builder objects.

### 25.5.38 PsdVar

COPT PSD variable object. PSD variables are always associated with a particular model. User creates a PSD variable object by adding a PSD variable to model, rather than by constructor of PsdVar class.

#### PsdVar::Get()

Get information values of PSD variable.

**Synopsis**

```
double Get(const char *szInfo, int sz)
```

**Arguments**

`szInfo`: name of information.

`sz`: length of the output array.

**Return**

output array of information values.

### PsdVar::Get()

Get information values of PSD variable.

**Synopsis**

```
NdArray<double, 2> Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of information.

**Return**

2-dimensional NdArray with related information values.

### PsdVar::GetDim()

Get dimension of PSD variable.

**Synopsis**

```
int GetDim()
```

**Return**

dimension of PSD variable.

### PsdVar::GetIdx()

Get index of PSD variable.

**Synopsis**

```
int GetIdx()
```

**Return**

index of PSD variable.

### PsdVar::GetLen()

Get length of PSD variable.

**Synopsis**

```
int GetLen()
```

**Return**

length of PSD variable.

### PsdVar::GetName()

Get name of PSD variable.

**Synopsis**

```
const char *GetName()
```

**Return**

name of PSD variable.

#### PsdVar::GetSize()

Get size or length of PSD variable.

**Synopsis**

```
int GetSize()
```

**Return**

Size of PSD variable.

#### PsdVar::Remove()

Remove PSD variable from model.

**Synopsis**

```
void Remove()
```

#### PsdVar::SetName()

Set name of PSD variable.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: name of PSD variable.

### 25.5.39 PsdVarArray

COPT PSD variable array object. To store and access a set of *PsdVar* objects, Cardinal Optimizer provides PsdVarArray class, which defines the following methods.

#### PsdVarArray::GetPsdVar()

Get idx-th PSD variable object.

**Synopsis**

```
PsdVar &GetPsdVar(int idx)
```

**Arguments**

`idx`: index of the PSD variable.

**Return**

PSD variable object with index idx.

#### PsdVarArray::PushBack()

Add a PSD variable object to PSD variable array.

**Synopsis**

```
void PushBack(const PsdVar &var)
```

**Arguments**

`var`: a PSD variable object.

#### PsdVarArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD variable object.

#### PsdVarArray::Size()

Get the number of PSD variable objects.

**Synopsis**

```
int Size()
```

**Return**

number of PSD variable objects.

### 25.5.40 PsdExpr

COPT PSD expression object. A PSD expression consists of a linear expression, a list of PSD variables and associated coefficient matrices of PSD terms. PSD expressions are used to build PSD constraints and objectives.

#### PsdExpr::PsdExpr()

Constructor of a PSD expression with a constant.

**Synopsis**

```
PsdExpr(double constant)
```

**Arguments**

`constant`: constant value in PSD expression object.

#### PsdExpr::PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(const Var &var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

#### PsdExpr::PsdExpr()

Constructor of a PSD expression with a linear expression.

**Synopsis**

```
PsdExpr(const Expr &expr)
```

**Arguments**

`expr`: input linear expression.

#### PsdExpr::PsdExpr()

Constructor of a PSD expression with a MExpression object.

**Synopsis**

```
PsdExpr(const MExpression &expr)
```

**Arguments**

`expr`: input MExpression object.

#### PsdExpr::PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(const PsdVar &var, const SymMatrix &mat)
```

**Arguments**

`var`: PSD variable for the added term.

`mat`: coefficient matrix for the added term.

#### PsdExpr::PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(const PsdVar &var, const SymMatExpr &expr)
```

**Arguments**

`var`: PSD variable for the added term.

`expr`: coefficient expression of symmetric matrices for the added term.

### PsdExpr::AddConstant()

Add constant to the PSD expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: value to be added.

### PsdExpr::AddLinExpr()

Add a linear expression to self.

**Synopsis**

```
void AddLinExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### PsdExpr::AddMExpr()

Add MExpression to PsdExpr object.

**Synopsis**

```
void AddMExpr(const MExpression &expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### PsdExpr::AddPsdExpr()

Add a PSD expression to self.

**Synopsis**

```
void AddPsdExpr(const PsdExpr &expr, double mult)
```

**Arguments**

`expr`: PSD expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### PsdExpr::AddTerm()

Add a linear term to PSD expression object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

#### PsdExpr::AddTerm()

Add a PSD term to PSD expression object.

**Synopsis**

```
void AddTerm(const PsdVar &var, const SymMatrix &mat)
```

**Arguments**

`var`: PSD variable of new PSD term.

`mat`: coefficient matrix of new PSD term.

#### PsdExpr::AddTerm()

Add a PSD term to PSD expression object.

**Synopsis**

```
void AddTerm(const PsdVar &var, const SymMatExpr &expr)
```

**Arguments**

`var`: PSD variable of new PSD term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

#### PsdExpr::AddTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
int AddTerms(
    const VarArray &vars,
    double *pCoeff,
    int len)
```

**Arguments**

`vars`: variables for added linear terms.

`pCoeff`: coefficient array for added linear terms.

`len`: length of coefficient array.

**Return**

number of added linear terms.

#### PsdExpr::AddTerms()

Add PSD terms to PSD expression object.

**Synopsis**

```
int AddTerms(const PsdVarArray &vars, const SymMatrixArray &mats)
```

**Arguments**

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrices for added PSD terms.

**Return**

number of added PSD terms.

### PsdExpr::Clone()

Deep copy PSD expression object.

**Synopsis**

```
PsdExpr Clone()
```

**Return**

cloned PSD expression object.

### PsdExpr::Evaluate()

evaluate PSD expression after solving

**Synopsis**

```
double Evaluate()
```

**Return**

value of PSD expression

### PsdExpr::GetCoeff()

Get coefficient from the i-th term in PSD expression.

**Synopsis**

```
SymMatExpr &GetCoeff(int i)
```

**Arguments**

`i`: index of the PSD term.

**Return**

coefficient of the i-th PSD term in PSD expression object.

### PsdExpr::GetConstant()

Get constant in PSD expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in PSD expression.

#### PsdExpr::GetLinExpr()

Get linear expression in PSD expression.

**Synopsis**

```
Expr &GetLinExpr()
```

**Return**

linear expression object.

#### PsdExpr::GetPsdVar()

Get the PSD variable from the i-th term in PSD expression.

**Synopsis**

```
PsdVar &GetPsdVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

the first variable of the i-th term in PSD expression object.

#### PsdExpr::operator*=()

Multiply a constant to self.

**Synopsis**

```
void operator*=(double c)
```

**Arguments**

`c`: constant multiplier.

#### PsdExpr::operator*()

Multiply constant and return new expression.

**Synopsis**

```
PsdExpr operator*(double c)
```

**Arguments**

`c`: constant multiplier.

**Return**

result expression.

### PsdExpr::operator+=()

Add an expression to self.

**Synopsis**

```
void operator+=(const PsdExpr &expr)
```

**Arguments**

`expr`: expression to be added.

### PsdExpr::operator+()

Add expression and return new expression.

**Synopsis**

```
PsdExpr operator+(const PsdExpr &other)
```

**Arguments**

`other`: other expression to add.

**Return**

result expression.

### PsdExpr::operator-=()

Subtract an expression from self.

**Synopsis**

```
void operator-=(const PsdExpr &expr)
```

**Arguments**

`expr`: expression to be subtracted.

### PsdExpr::operator-()

Subtract expression and return new expression.

**Synopsis**

```
PsdExpr operator-(const PsdExpr &other)
```

**Arguments**

`other`: other expression to subtract.

**Return**

result expression.

### PsdExpr::Remove()

Remove i-th term from PSD expression object.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### PsdExpr::Remove()

Remove the term associated with variable from PSD expression.

**Synopsis**

```
void Remove(const Var &var)
```

**Arguments**

`var`: a variable whose term should be removed.

### PsdExpr::Remove()

Remove the term associated with PSD variable from PSD expression.

**Synopsis**

```
void Remove(const PsdVar &var)
```

**Arguments**

`var`: a PSD variable whose term should be removed.

### PsdExpr::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(size_t n)
```

**Arguments**

`n`: minimum capacity for PSD expression object.

### PsdExpr::SetCoeff()

Set coefficient matrix of the i-th term in PSD expression.

**Synopsis**

```
void SetCoeff(int i, const SymMatrix &mat)
```

**Arguments**

`i`: index of the PSD term.

`mat`: coefficient matrix of the term.

#### PsdExpr::SetConstant()

Set constant for the PSD expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### PsdExpr::Size()

Get number of PSD terms in expression.

**Synopsis**

```
size_t Size()
```

**Return**

number of PSD terms.

### 25.5.41 PsdConstraint

COPT PSD constraint object. PSD constraints are always associated with a particular model. User creates a PSD constraint object by adding a PSD constraint to model, rather than by constructor of PsdConstraint class.

#### PsdConstraint::Get()

Get information value of the PSD constraint. Support related PSD information.

**Synopsis**

```
double Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of queried information.

**Return**

value of information.

#### PsdConstraint::GetIdx()

Get index of the PSD constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

the index of the PSD constraint.

#### PsdConstraint::GetName()

Get name of the PSD constraint.

**Synopsis**

```
const char *GetName()
```

**Return**

the name of the PSD constraint.

#### PsdConstraint::Remove()

Remove this PSD constraint from model.

**Synopsis**

```
void Remove()
```

#### PsdConstraint::Set()

Set information value of the PSD constraint. Support related PSD information.

**Synopsis**

```
void Set(const char *szInfo, double value)
```

**Arguments**

`szInfo`: name of queried information.

`value`: new information value.

#### PsdConstraint::SetName()

Set name of a PSD constraint.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: the name to set.

### 25.5.42 PsdConstrArray

COPT PSD constraint array object. To store and access a set of *PsdConstraint* objects, Cardinal Optimizer provides PsdConstrArray class, which defines the following methods.

#### PsdConstrArray::GetPsdConstr()

Get idx-th PSD constraint object.

**Synopsis**

```
PsdConstraint &GetPsdConstr(int idx)
```

**Arguments**

`idx`: index of the PSD constraint.

**Return**

PSD constraint object with index idx.

#### PsdConstrArray::PushBack()

Add a PSD constraint object to PSD constraint array.

**Synopsis**

```
void PushBack(const PsdConstraint &constr)
```

**Arguments**

`constr`: a PSD constraint object.

#### PsdConstrArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD constraint objects.

#### PsdConstrArray::Size()

Get the number of PSD constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of PSD constraint objects.

### 25.5.43 PsdConstrBuilder

COPT PSD constraint builder object. To help building a PSD constraint, given a PSD expression, constraint sense and right-hand side value, Cardinal Optimizer provides PsdConstrBuilder class, which defines the following methods.

#### PsdConstrBuilder::GetPsdExpr()

Get expression associated with PSD constraint.

**Synopsis**

```
const PsdExpr &GetPsdExpr()
```

**Return**

PSD expression object.

### PsdConstrBuilder::GetRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of the constraint.

### PsdConstrBuilder::GetSense()

Get sense associated with PSD constraint.

**Synopsis**

```
char GetSense()
```

**Return**

PSD constraint sense.

### PsdConstrBuilder::Set()

Set detail of a PSD constraint to its builder object.

**Synopsis**

```
void Set(
    const PsdExpr &expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the PSD constraint.

`sense`: PSD constraint sense, other than COPT_RANGE.

`rhs`: constant at right side of the PSD constraint.

### PsdConstrBuilder::SetRange()

Set a range constraint to its builder.

**Synopsis**

```
void SetRange(const PsdExpr &expr, double range)
```

**Arguments**

`expr`: PSD expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must be greater than 0.

### 25.5.44 PsdConstrBuilderArray

COPT PSD constraint builder array object. To store and access a set of *PsdConstrBuilder* objects, Cardinal Optimizer provides PsdConstrBuilderArray class, which defines the following methods.

#### PsdConstrBuilderArray::GetBuilder()

Get idx-th PSD constraint builder object.

**Synopsis**

```
PsdConstrBuilder &GetBuilder(int idx)
```

**Arguments**

`idx`: index of the PSD constraint builder.

**Return**

PSD constraint builder object with index idx.

#### PsdConstrBuilderArray::PushBack()

Add a PSD constraint builder object to PSD constraint builder array.

**Synopsis**

```
void PushBack(const PsdConstrBuilder &builder)
```

**Arguments**

`builder`: a PSD constraint builder object.

#### PsdConstrBuilderArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD constraint builder object.

#### PsdConstrBuilderArray::Size()

Get the number of PSD constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of PSD constraint builder objects.

### 25.5.45 LmiConstraint

COPT LMI constraint object. LMI constraints are always associated with a particular model. User creates a LMI constraint object by adding a LMI constraint to model, rather than by constructor of LmiConstraint class.

#### LmiConstraint::Get()

Get information values of LMI expression.

**Synopsis**

```
double Get(const char *szInfo, int len)
```

**Arguments**

`szInfo`: name of queried information.

`len`: length of output array.

**Return**

output list of information values.

#### LmiConstraint::GetDim()

Get dimension of LMI constraint.

**Synopsis**

```
int GetDim()
```

**Return**

dimension of LMI constraint.

#### LmiConstraint::GetIdx()

Get index of LMI constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of LMI constraint.

#### LmiConstraint::GetLen()

Get length of LMI constraint.

**Synopsis**

```
int GetLen()
```

**Return**

length of LMI constraint.

#### LmiConstraint::GetName()

Get name of LMI constraint.

**Synopsis**

```
const char *GetName()
```

**Return**

the name of LMI constraint.

#### LmiConstraint::Remove()

Remove this LMI constraint from model.

**Synopsis**

```
void Remove()
```

#### LmiConstraint::SetName()

Set name of LMI constraint.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: new name to set.

#### LmiConstraint::SetRhs()

Set constant term of LMI constraint.

**Synopsis**

```
void SetRhs(const SymMatrix &mat)
```

**Arguments**

`mat`: new symmetric matrix for constant term.

### 25.5.46 LmiConstrArray

COPT LMI constraint array object. To store and access a set of *LmiConstraint* objects, Cardinal Optimizer provides LmiConstrArray class, which defines the following methods.

#### LmiConstrArray::GetLmiConstr()

Get idx-th LMI constraint object.

**Synopsis**

```
LmiConstraint &GetLmiConstr(int idx)
```

**Arguments**

`idx`: index of the LMI constraint.

**Return**

LMI constraint object with index idx.

#### LmiConstrArray::PushBack()

Add a LMI constraint to LMI constraint array.

**Synopsis**

```
void PushBack(const LmiConstraint &constr)
```

**Arguments**

`constr`: LMI constraint object.

#### LmiConstrArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: capacity number of LMI constraint objects.

#### LmiConstrArray::Size()

Get the number of LMI constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of LMI constraint objects.

### 25.5.47 LmiExpr

COPT LMI expression object. A LMI expression consists of a list of variables, associated coefficient matrices of LMI term, and constant matrices. LMI expressions are used to build LMI constraints.

#### LmiExpr::LmiExpr()

Default constructor of a LMI expression.

**Synopsis**

```
LmiExpr()
```

#### LmiExpr::LmiExpr()

Constructor of LMI expression with given symmetric matrix.

**Synopsis**

```
LmiExpr(const SymMatrix &mat)
```

**Arguments**

`mat`: symmetric matrix as constant term of LMI expression.

### LmiExpr::LmiExpr()

Constructor of LMI expression with given matrix expression.

**Synopsis**

```
LmiExpr(const SymMatExpr &expr)
```

**Arguments**

`expr`: matrix expression as constant term of LMI expression.

### LmiExpr::LmiExpr()

Constructor of LMI expression with one term.

**Synopsis**

```
LmiExpr(const Var &var, const SymMatrix &mat)
```

**Arguments**

`var`: variable of the added term.

`mat`: coefficient matrix of the added term.

### LmiExpr::LmiExpr()

Constructor of LMI expression with one term.

**Synopsis**

```
LmiExpr(const Var &var, const SymMatExpr &expr)
```

**Arguments**

`var`: variable of the added term.

`expr`: coefficient expression of symmetric matrices for the added term.

### LmiExpr::AddConstant()

Add to constant term of LMI expression.

**Synopsis**

```
void AddConstant(const SymMatExpr &expr)
```

**Arguments**

`expr`: matrix expression object added to constant term.

### LmiExpr::AddLmiExpr()

Add a LMI expression to self.

**Synopsis**

```
void AddLmiExpr(const LmiExpr &expr, double mult)
```

**Arguments**

`expr`: LMI expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

#### LmiExpr::AddTerm()

Add a term to LMI expression.

**Synopsis**

```
void AddTerm(const Var &var, const SymMatrix &mat)
```

**Arguments**

`var`: variable of new LMI term.

`mat`: coefficient matrix object of new LMI term.

#### LmiExpr::AddTerm()

Add a term to LMI expression.

**Synopsis**

```
void AddTerm(const Var &var, const SymMatExpr &expr)
```

**Arguments**

`var`: variable of new LMI term.

`expr`: coefficient expression object of symmetric matrices of new LMI term.

#### LmiExpr::AddTerms()

Add LMI terms to LMI expression.

**Synopsis**

```
int AddTerms(const VarArray &vars, const SymMatrixArray &mats)
```

**Arguments**

`vars`: variables of added LMI terms.

`mats`: coefficient matrix objects for added LMI terms.

**Return**

number of added LMI terms.

#### LmiExpr::Clone()

Deep copy LMI expression.

**Synopsis**

```
LmiExpr Clone()
```

**Return**

cloned LMI expression object.

### LmiExpr::GetCoeff()

Get coefficient from the i-th term in LMI expression.

**Synopsis**

```
SymMatExpr &GetCoeff(int i)
```

**Arguments**

`i`: index of the LMI term.

**Return**

coefficient matrix expression object of the i-th LMI term in LMI expression.

### LmiExpr::GetConstant()

Get constant term in LMI expression.

**Synopsis**

```
SymMatExpr &GetConstant()
```

**Return**

matrix expression object in LMI expression.

### LmiExpr::GetVar()

Get variable from the i-th term in LMI expression.

**Synopsis**

```
Var &GetVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

variable of the i-th term in LMI expression object.

### LmiExpr::operator*=()

Multiply a double constant to self.

**Synopsis**

```
void operator*=(double c)
```

**Arguments**

`c`: constant multiplier.

#### LmiExpr::operator*()

Multiply double constant and return new expression.

**Synopsis**

```
LmiExpr operator*(double c)
```

**Arguments**

`c`: constant multiplier.

**Return**

result expression.

#### LmiExpr::operator+=()

Add a symmetric matrix or LMI expression to self.

**Synopsis**

```
void operator+=(const LmiExpr &expr)
```

**Arguments**

`expr`: symmetric matrix or LMI expression to be added.

#### LmiExpr::operator+()

Add a symmetric matrix or LMI expression and return new LMI expression.

**Synopsis**

```
LmiExpr operator+(const LmiExpr &other)
```

**Arguments**

`other`: other symmetric matrix or LMI expression to add.

**Return**

result expression.

#### LmiExpr::operator-=()

Subtract a symmetric matrix or LMI expression from self.

**Synopsis**

```
void operator-=(const LmiExpr &expr)
```

**Arguments**

`expr`: symmetric matrix or LMI expression to be subtracted.

### LmiExpr::operator-()

Subtract a symmetric matrix or LMI expression and return new expression.

**Synopsis**

```
LmiExpr operator-(const LmiExpr &other)
```

**Arguments**

`other`: other symmetric matrix or LMI expression to subtract.

**Return**

result expression.

### LmiExpr::Remove()

Remove i-th term from LMI expression.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### LmiExpr::Remove()

Remove the term associated with given variable from LMI expression.

**Synopsis**

```
void Remove(const Var &var)
```

**Arguments**

`var`: a variable whose term should be removed.

### LmiExpr::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(size_t n)
```

**Arguments**

`n`: capacity number of LMI expression.

### LmiExpr::SetCoeff()

Set coefficient matrix of the i-th term in LMI expression.

**Synopsis**

```
void SetCoeff(int i, const SymMatrix &mat)
```

**Arguments**

`i`: index of the LMI term.

`mat`: new coefficient matrix object.

#### LmiExpr::SetConstant()

Set constant term of the LMI expression.

**Synopsis**

```
void SetConstant(const SymMatrix &mat)
```

**Arguments**

`mat`: new matrix object.

#### LmiExpr::Size()

Get number of terms in LMI expression.

**Synopsis**

```
size_t Size()
```

**Return**

number of LMI terms.

### 25.5.48 SymMatrix

COPT symmetric matrix object. Symmetric matrices are always associated with a particular model. User creates a symmetric matrix object by adding a symmetric matrix to model, rather than by constructor of SymMatrix class.

Symmetric matrices are used as coefficient matrices of PSD terms in PSD expressions, PSD constraints or PSD objectives.

#### SymMatrix::GetDim()

Get the dimension of a symmetric matrix.

**Synopsis**

```
int GetDim()
```

**Return**

dimension of a symmetric matrix.

#### SymMatrix::GetIdx()

Get the index of a symmetric matrix.

**Synopsis**

```
int GetIdx()
```

**Return**

index of a symmetric matrix.

### 25.5.49 SymMatrixArray

COPT symmetric matrix object. To store and access a set of *SymMatrix* objects, Cardinal Optimizer provides SymMatrixArray class, which defines the following methods.

#### SymMatrixArray::GetMatrix()

Get i-th SymMatrix object.

**Synopsis**

```
SymMatrix &GetMatrix(int idx)
```

**Arguments**

`idx`: index of the SymMatrix object.

**Return**

SymMatrix object with index idx.

#### SymMatrixArray::PushBack()

Add a SymMatrix object to SymMatrix array.

**Synopsis**

```
void PushBack(const SymMatrix &mat)
```

**Arguments**

`mat`: a SymMatrix object.

#### SymMatrixArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for symmetric matrix object.

#### SymMatrixArray::Size()

Get the number of SymMatrix objects.

**Synopsis**

```
int Size()
```

**Return**

number of SymMatrix objects.

### 25.5.50 SymMatExpr

COPT symmetric matrix expression object. A symmetric matrix expression is a linear combination of symmetric matrices, which is still a symmetric matrix. However, by doing so, we are able to delay computing the final matrix until setting PSD constraints or PSD objective.

#### SymMatExpr::SymMatExpr()

Constructor of a symmetric matrix expression.

**Synopsis**

```
SymMatExpr()
```

#### SymMatExpr::SymMatExpr()

Constructor of a symmetric matrix expression with one term.

**Synopsis**

```
SymMatExpr(const SymMatrix &mat, double coeff)
```

**Arguments**

`mat`: symmetric matrix of the added term.

`coeff`: optional, coefficient for the added term. Its default value is 1.0.

#### SymMatExpr::AddSymMatExpr()

Add a symmetric matrix expression to self.

**Synopsis**

```
void AddSymMatExpr(const SymMatExpr &expr, double mult)
```

**Arguments**

`expr`: symmetric matrix expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

#### SymMatExpr::AddTerm()

Add a term to symmetric matrix expression object.

**Synopsis**

```
bool AddTerm(const SymMatrix &mat, double coeff)
```

**Arguments**

`mat`: symmetric matrix of the new term.

`coeff`: coefficient of the new term.

**Return**

True if the term is added successfully.

### SymMatExpr::AddTerms()

Add multiple terms to expression object.

**Synopsis**

```
int AddTerms(
    const SymMatrixArray &mats,
    double *pCoeff,
    int len)
```

**Arguments**

`mats`: symmetric matrix array object for added terms.

`pCoeff`: coefficient array for added terms.

`len`: length of coefficient array.

**Return**

Number of added terms. If negative, fail to add one of terms.

### SymMatExpr::Clone()

Deep copy symmetric matrix expression object.

**Synopsis**

```
SymMatExpr Clone()
```

**Return**

cloned expression object.

### SymMatExpr::GetCoeff()

Get coefficient of the i-th term in expression object.

**Synopsis**

```
double GetCoeff(int i)
```

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term.

### SymMatExpr::GetDim()

Get dimension of symmetric matrix in expression.

**Synopsis**

```
int GetDim()
```

**Return**

dimension of symmetric matrix.

#### SymMatExpr::GetSymMat()

Get symmetric matrix of the i-th term in expression object.

**Synopsis**

```
SymMatrix &GetSymMat(int i)
```

**Arguments**

`i`: index of the term.

**Return**

the symmetric matrix of the i-th term.

#### SymMatExpr::operator*=()

Multiply a constant to self.

**Synopsis**

```
void operator*=(double c)
```

**Arguments**

`c`: constant multiplier.

#### SymMatExpr::operator*()

Multiply constant and return new expression.

**Synopsis**

```
SymMatExpr operator*(double c)
```

**Arguments**

`c`: constant multiplier.

**Return**

result expression.

#### SymMatExpr::operator+=()

Add a symmetric matrix expression to self.

**Synopsis**

```
void operator+=(const SymMatExpr &expr)
```

**Arguments**

`expr`: symmetric matrix expression to be added.

### SymMatExpr::operator+()

Add expression and return new expression.

**Synopsis**

```
SymMatExpr operator+(const SymMatExpr &other)
```

**Arguments**

`other`: other expression to add.

**Return**

result expression.

### SymMatExpr::operator-=()

Subtract a symmetric matrix expression from self.

**Synopsis**

```
void operator-=(const SymMatExpr &expr)
```

**Arguments**

`expr`: symmetric matrix to be subtracted.

### SymMatExpr::operator-()

Subtract expression and return new expression.

**Synopsis**

```
SymMatExpr operator-(const SymMatExpr &other)
```

**Arguments**

`other`: other expression to subtract.

**Return**

result expression.

### SymMatExpr::Remove()

Remove i-th term from expression object.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

#### SymMatExpr::Remove()

Remove the term associated with the symmetric matrix.

**Synopsis**

```
void Remove(const SymMatrix &mat)
```

**Arguments**

`mat`: a symmetric matrix whose term should be removed.

#### SymMatExpr::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(size_t n)
```

**Arguments**

`n`: minimum capacity for expression object.

#### SymMatExpr::SetCoeff()

Set coefficient for the i-th term in expression object.

**Synopsis**

```
void SetCoeff(int i, double val)
```

**Arguments**

`i`: index of the term.

`val`: coefficient of the term.

#### SymMatExpr::Size()

Get number of terms in expression.

**Synopsis**

```
size_t Size()
```

**Return**

number of terms.

### 25.5.51 MVar

COPT multi-dimensional variable object. It is used to construct multi-dimensional variables and support operations with the built-in multi-dimensional array *NdArray* in COPT. It can be created by calling the method `addMVar` of *Model*. The following methods are provided:

### MVar::MVar()

Construct a MVar object with the given shape, filling with the given variable.

**Synopsis**

```
MVar(const Shape<N> &shp, const Var &var)
```

**Arguments**

`shp`: shape of MVar.

`var`: Variable object.

### MVar::MVar()

Construct a MVar object with the given shape, filling with an array of variables.

**Synopsis**

```
MVar(const Shape<N> &shp, const VarArray &vars)
```

**Arguments**

`shp`: shape of MVar.

`vars`: an array of variables.

### MVar::Clone()

Clone MVar object.

**Synopsis**

```
MVar Clone()
```

**Return**

new MVar object.

### MVar::Diagonal()

Get diagonals of MVar object.

**Synopsis**

```
MVar<N - 1> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MVar.

`axis2`: 2nd axis of MVar.

**Return**

(N-1)-dimensional diagonals.

### MVar::Expand()

Expand shape of MVar object.

**Synopsis**

```
MVar<N + 1> Expand(int axis)
```

**Arguments**

`axis`: axis of MVar.

**Return**

MVar object of (N+1)-dimensional shape.

### MVar::Flatten()

Flatten a MVar object to a 1-dimensional shape.

**Synopsis**

```
MVar<1> Flatten()
```

**Return**

a MVar object collapsed into one dimension.

### MVar::Get()

Get values of information associated with variables in MVar object.

**Synopsis**

```
NdArray<double, N> Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of information.

**Return**

multi-dimensional array of information of variables.

### MVar::GetBasis()

Get basis of variables in MVar object.

**Synopsis**

```
NdArray<int, N> GetBasis()
```

**Return**

multi-dimensional array of basis of variables.

### MVar::GetDim()

Get i-th dimension of MVar object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MVar::GetIdx()

Get indexes of variables in MVar object.

**Synopsis**

```
NdArray<int, N> GetIdx()
```

**Return**

multi-dimensional array of indexes of variables.

### MVar::GetLowerIIS()

Get IIS status of lower bound of variables in MVar object.

**Synopsis**

```
NdArray<int, N> GetLowerIIS()
```

**Return**

multi-dimensional array of IIS status of lower bounds of variables.

### MVar::GetND()

Get number of dimensions of MVar object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MVar::GetShape()

Get shape of MVar object.

**Synopsis**

```
Shape<N> GetShape()
```

**Return**

shape object.

### MVar::GetSize()

Get size of MVar object.

**Synopsis**

```
size_t GetSize()
```

**Return**

number of vars.

### MVar::GetType()

Get types of variables in MVar object.

**Synopsis**

```
NdArray<char, N> GetType()
```

**Return**

multi-dimensional array of types of variables.

### MVar::GetUpperIIS()

Get IIS status of upper bound of variables in MVar object.

**Synopsis**

```
NdArray<int, N> GetUpperIIS()
```

**Return**

multi-dimensional array of IIS status of upper bounds of variables.

### MVar::Item()

Get variable of given index from MVar object.

**Synopsis**

```
Var &Item(size_t idx)
```

**Arguments**

`idx`: index of var.

**Return**

Var object.

### MVar::operator[]()

Get variable of given index from MVar object.

**Synopsis**

```
Var &operator[](size_t idx)
```

**Arguments**

`idx`: index of var.

**Return**

Var object.

### MVar::operator[]()

Get sub-arrays of MVar object, given view object.

**Synopsis**

```
MVar operator[](const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MVar object.

### MVar::Pick()

Given a list of indexes, get variables from MVar object.

**Synopsis**

```
MVar<1> Pick(const NdArray<int, 1> &indexes)
```

**Arguments**

`indexes`: indexes of elements.

**Return**

one-dimensional array of desired variables.

### MVar::Pick()

Given a list of indexes, get variables from MVar object.

**Synopsis**

```
MVar<1> Pick(const NdArray<int, 2> &idxrows)
```

**Arguments**

`idxrows`: indexes in format of 2-dimensional array, where each row is position of element.

**Return**

one-dimensional array of desired variables.

### MVar::Repeat()

Repeat each element of MVar along given axis.

**Synopsis**

```
MVar<N> Repeat(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of MVar.

**Return**

new MVar object.

### MVar::RepeatBlock()

Repeat an MVar a number of times along given axis.

**Synopsis**

```
MVar<N> RepeatBlock(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of MVar.

**Return**

new MVar object.

### MVar::Represent()

String representation of MVar object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

### MVar::Reshape()

Reshape MVar object to new shape.

**Synopsis**

```
template <int M> MVar<M> Reshape(const Shape<M> &shape)
```

**Arguments**

`shape`: new shape of M-dimensions.

**Return**

M-dimensional MVar object.

### MVar::Set()

Set values of information associated with variables in MVar object.

**Synopsis**

```
void Set(const char *szInfo, double val)
```

**Arguments**

`szInfo`: name of information.

`val`: value of information.

### MVar::Set()

Set values of information associated with variables in MVar object.

**Synopsis**

```
void Set(const char *szInfo, const NdArray<double, N> &vals)
```

**Arguments**

`szInfo`: name of information.

`vals`: multi-dimensional array of values of information.

### MVar::SetItem()

Set variable of given index to MVar object.

**Synopsis**

```
void SetItem(size_t idx, const Var &var)
```

**Arguments**

`idx`: index of element.

`var`: Var object.

### MVar::Squeeze()

Remove axis of length 1 from shape of MVar object.

**Synopsis**

```
MVar<N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: axis of MVar, where the length is 1.

**Return**

MVar object of (N-1)-dimensional shape.

### MVar::Stack()

Stack with other MVar object along given axis.

**Synopsis**

```
MVar<N> Stack(const MVar<N> &other, int axis)
```

**Arguments**

`other`: a MVar object.

`axis`: an axis of MVar.

**Return**

the result MVar object.

#### MVar::Sum()

Sum of all variables in MVar object.

**Synopsis**

```
MLinExpr<0> Sum()
```

**Return**

sum in zero dimension.

#### MVar::Sum()

Sum of variables at given axis of MVar object.

**Synopsis**

```
MLinExpr<N - 1> Sum(int axis)
```

**Arguments**

`axis`: axis of MVar.

**Return**

MLinExpr object in (N-1)-dimension.

#### MVar::Transpose()

Perform matrix transpose of MVar object.

**Synopsis**

```
MVar<N> Transpose()
```

**Return**

transposed MVar object.

### 25.5.52 MConstr

COPT multi-dimensional linear constraint object. It can be created by calling the method `addMConstr` of *Model*. The following methods are provided:

#### MConstr::MConstr()

Construct a MConstr object with the given shape, filling with the given constraint.

**Synopsis**

```
MConstr(const Shape<N> &shp, const Constraint &con)
```

**Arguments**

`shp`: shape of MConstr.

`con`: Constraint object.

### MConstr::MConstr()

Construct a MConstr object with the given shape, filling with an array of constraints.

**Synopsis**

```
MConstr(const Shape<N> &shp, const ConstrArray &cons)
```

**Arguments**

`shp`: shape of MConstr.

`cons`: an array of constraints.

### MConstr::Clone()

Clone MConstr object.

**Synopsis**

```
MConstr Clone()
```

**Return**

new MConstr object.

### MConstr::Diagonal()

Get diagonals of MConstr object.

**Synopsis**

```
MConstr<N - 1> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MConstr.

`axis2`: 2nd axis of MConstr.

**Return**

(N-1)-dimensional diagonals.

### MConstr::Expand()

Expand shape of MConstr object.

**Synopsis**

```
MConstr<N + 1> Expand(int axis)
```

**Arguments**

`axis`: axis of MConstr.

**Return**

MConstr object of (N+1)-dimensional shape.

### MConstr::Flatten()

Flatten a MConstr object to a 1-dimensional shape.

**Synopsis**

```
MConstr<1> Flatten()
```

**Return**

a MConstr object collapsed into one dimension.

### MConstr::Get()

Get values of information associated with constraints in MConstr object.

**Synopsis**

```
NdArray<double, N> Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of information.

**Return**

multi-dimensional array of information of constraints.

### MConstr::GetBasis()

Get basis of constraints in MConstr object.

**Synopsis**

```
NdArray<int, N> GetBasis()
```

**Return**

multi-dimensional array of basis of constraints.

### MConstr::GetDim()

Get i-th dimension of MConstr object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MConstr::GetIdx()

Get index of constraints in MConstr object.

**Synopsis**

```
NdArray<int, N> GetIdx()
```

**Return**

multi-dimensional array of indexes of constraints.

### MConstr::GetLowerIIS()

Get IIS status of lower bound of constraints in MConstr object.

**Synopsis**

```
NdArray<int, N> GetLowerIIS()
```

**Return**

multi-dimensional array of IIS status of lower bounds of constraints.

### MConstr::GetND()

Get number of dimensions of MConstr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MConstr::GetShape()

Get shape of MConstr object.

**Synopsis**

```
Shape<N> GetShape()
```

**Return**

shape object.

### MConstr::GetSize()

Get size of MConstr object.

**Synopsis**

```
size_t GetSize()
```

**Return**

number of constraints.

### MConstr::GetUpperIIS()

Get IIS status of upper bound of constraints in MConstr object.

**Synopsis**

```
NdArray<int, N> GetUpperIIS()
```

**Return**

multi-dimensional array of IIS status of upper bounds of constraints.

### MConstr::Item()

Get constraint of given index from MConstr object.

**Synopsis**

```
Constraint &Item(size_t idx)
```

**Arguments**

`idx`: index of constraint.

**Return**

Constraint object.

### MConstr::operator[]()

Get constraint of given index from MConstr object.

**Synopsis**

```
Constraint &operator[](size_t idx)
```

**Arguments**

`idx`: index of constraint.

**Return**

Constraint object.

### MConstr::operator[]()

Get constraints of given view from MConstr object.

**Synopsis**

```
MConstr operator[](const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

new MConstr object.

### MConstr::Pick()

Given a list of indexes, get constraints from MConstr object.

**Synopsis**

```
MConstr<1> Pick(const NdArray<int, 1> &indexes)
```

**Arguments**

`indexes`: indexes of elements.

**Return**

one-dimensional array of desired constraints.

### MConstr::Pick()

Given a list of indexes, get constraints from MConstr object.

**Synopsis**

```
MConstr<1> Pick(const NdArray<int, 2> &idxrows)
```

**Arguments**

`idxrows`: indexes in format of 2-dimensional array, where each row is position of element.

**Return**

one-dimensional array of desired constraints.

### MConstr::Represent()

String representation of MConstr object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

### MConstr::Reshape()

Reshape MConstr object to new shape.

**Synopsis**

```
template <int M> MConstr<M> Reshape(const Shape<M> &shape)
```

**Arguments**

`shape`: new shape of M-dimensions.

**Return**

M-dimensional MConstr object.

### MConstr::Set()

Set values of information associated with constraints in MConstr object.

**Synopsis**

```
void Set(const char *szInfo, double val)
```

**Arguments**

`szInfo`: name of information.

`val`: value of information.

### MConstr::Set()

Set values of information associated with constraints in MConstr object.

**Synopsis**

```
void Set(const char *szInfo, const NdArray<double, N> &vals)
```

**Arguments**

`szInfo`: name of information.

`vals`: multi-dimensional array of values of information.

### MConstr::SetItem()

Set constraint of given index to MConstr object.

**Synopsis**

```
void SetItem(size_t idx, const Constraint &con)
```

**Arguments**

`idx`: index of element.

`con`: constraint object.

### MConstr::Squeeze()

Remove axis of length 1 from shape of MConstr object.

**Synopsis**

```
MConstr<N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: axis of MConstr, where the length is 1.

**Return**

MConstr object of (N-1)-dimensional shape.

#### MConstr::Stack()

Stack with other MConstr object along given axis.

**Synopsis**

```
MConstr<N> Stack(const MConstr<N> &other, int axis)
```

**Arguments**

`other`: a MConstr object.

`axis`: an axis of MConstr.

**Return**

the result MConstr object.

#### MConstr::Transpose()

Perform matrix transpose of MConstr object.

**Synopsis**

```
MConstr<N> Transpose()
```

**Return**

transposed MConstr object.

### 25.5.53 MConstrBuilder

COPT builder object of multi-dimensional linear constraints. It is used to generate multi-dimensional linear constraints and support operations with the built-in multi-dimensional array *NdArray* in COPT. It is recommended to create MConstrBuilder object by comparing two objects, one of which should be *MVar* object or *MLinExpr* object, by comparison operators. The following methods are provided:

#### MConstrBuilder::MConstrBuilder()

Construct a MConstrBuilder object with the given shape.

**Synopsis**

```
MConstrBuilder(const Shape<N> &shp)
```

**Arguments**

`shp`: shape of MConstrBuilder.

#### MConstrBuilder::Flatten()

Flatten a MConstrBuilder object to a 1-dimensional shape.

**Synopsis**

```
MConstrBuilder<1> Flatten()
```

**Return**

a MConstrBuilder object collapsed into one dimension.

### MConstrBuilder::GetExpr()

Get N-dimensional linear expressions associated with N-dimensional constraints.

**Synopsis**

```
const MLinExpr<N> &GetExpr()
```

**Return**

MLinExpr object.

### MConstrBuilder::GetND()

Get number of dimensions of MConstrBuilder object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MConstrBuilder::GetRange()

Get range from lower bound to upper bound of N-dimensional range constraints.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of range constraints.

### MConstrBuilder::GetSense()

Get sense associated with N-dimensional constraints.

**Synopsis**

```
char GetSense()
```

**Return**

constraint sense.

### MConstrBuilder::Set()

Set N-dimensional constraints to its builder object.

**Synopsis**

```
void Set(
    const MLinExpr<N> &expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: MLinExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of constraints.

### MConstrBuilder::Set()

Set N-dimensional constraints to its builder object.

**Synopsis**

```
template <class T> void Set(
    const MLinExpr<N> &expr,
    char sense,
    const NdArray<T, N> &rhs)
```

**Arguments**

`expr`: MLinExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: N-dimensional constants at right side of constraints.

### MConstrBuilder::Set()

Set N-dimensional constraints to its builder object.

**Synopsis**

```
template <int M> void Set(
    const MLinExpr<N> &expr,
    char sense,
    const MVar<M> &rhs)
```

**Arguments**

`expr`: MLinExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MVar object at right side of constraints.

### MConstrBuilder::Set()

Set N-dimensional constraints to its builder object.

**Synopsis**

```
template <int M> void Set(
    const MLinExpr<N> &expr,
    char sense,
    const MLinExpr<M> &rhs)
```

**Arguments**

`expr`: MLinExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MLinExpr object at right side of constraints.

#### MConstrBuilder::SetRange()

Set N-dimensional range constraints to its builder object.

**Synopsis**

```
void SetRange(const MLinExpr<N> &expr, double range)
```

**Arguments**

`expr`: MLinExpr object.

`range`: length from lower bound to upper bound of the constraint. Must be greater than 0.

### 25.5.54 MExpression

The MExpression class is a generalized version of *Expr*. It represents a linear expression and supports most of methods in Expr class. In addition, it supports linear combination of multi-dimensional objects, such as *MVar* object and *NdArray* object. The following methods are provided:

#### MExpression::MExpression()

Construct a MExpression object with the given constant.

**Synopsis**

```
MExpression(double constant)
```

**Arguments**

`constant`: constant number.

#### MExpression::MExpression()

Construct a MExpression object with the given linear expression.

**Synopsis**

```
MExpression(const Expr &expr)
```

**Arguments**

`expr`: a linear expression.

#### MExpression::MExpression()

Construct a MExpression object with the given variable.

**Synopsis**

```
MExpression(const Var &var)
```

**Arguments**

`var`: variable object.

### MExpression::AddConstant()

Add constant for the expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### MExpression::AddExpr()

Add a linear expression to MExpression object.

**Synopsis**

```
void AddExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

### MExpression::AddMExpr()

Add MExpression to MExpression object.

**Synopsis**

```
void AddMExpr(const MExpression &expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MExpression::AddTerm()

Add a linear term to MExpression object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

### MExpression::Clone()

Clone MExpression object.

**Synopsis**

```
MExpression Clone()
```

**Return**

new MExpression object.

#### MExpression::Evaluate()

Evaluate MExpression after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of MExpression object.

#### MExpression::GetConstant()

Get constant in expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in expression.

#### MExpression::Represent()

String representation of MExpression object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

#### MExpression::SetConstant()

Set constant for the expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### 25.5.55 MLinExpr

COPT multi-dimensional linear expression object. It is used to construct multi-dimensional linear expressions and perform operations with the built-in multi-dimensional array *NdArray* in COPT. Its elements are *MExpression* objects. It can be created by linear combination of *MVar* objects. The following methods are provided:

### MLinExpr::MLinExpr()

Construct a MLinExpr object with the given shape and a constant.

**Synopsis**

```
MLinExpr(const Shape<N> &shp, double constant)
```

**Arguments**

`shp`: shape of MLinExpr.

`constant`: constant number.

### MLinExpr::MLinExpr()

Construct a MLinExpr object with the given shape and a linear expression.

**Synopsis**

```
MLinExpr(const Shape<N> &shp, const Expr &expr)
```

**Arguments**

`shp`: shape of MLinExpr.

`expr`: a linear expression.

### MLinExpr::MLinExpr()

Construct a MLinExpr object with the given shape and a MExpression object.

**Synopsis**

```
MLinExpr(const Shape<N> &shp, const MExpression &expr)
```

**Arguments**

`shp`: shape of MLinExpr.

`expr`: a MExpression object.

### MLinExpr::AddConstant()

Add constant to each expression in MLinExpr object.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### MLinExpr::AddConstant()

Add constants to each expression in MLinExpr object.

**Synopsis**

```
template <class T> void AddConstant(const NdArray<T, N> &constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

#### MLinExpr::AddExpr()

Add a linear expression to each expression in MLinExpr object.

**Synopsis**

```
void AddExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

#### MLinExpr::AddMExpr()

Add MExpression to each expression in MLinExpr object.

**Synopsis**

```
void AddMExpr(const MExpression &expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

#### MLinExpr::AddMLinExpr()

Add linear expressions to MLinExpr object.

**Synopsis**

```
void AddMLinExpr(const MLinExpr<N> &exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MLinExpr object.

`mult`: the same multiplier for added linear expressions, default value is 1.0.

#### MLinExpr::AddTerm()

Add a linear term to MLinExpr object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

### MLinExpr::AddTerms()

Add terms to expressions in MLinExpr object.

**Synopsis**

```
void AddTerms(const MVar<N> &vars, double mult)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`mult`: the same coefficient for added terms, default value 1.0.

### MLinExpr::AddTerms()

Add terms to expressions in MLinExpr object.

**Synopsis**

```
void AddTerms(const MVar<N> &vars, const NdArray<double, N> &coeffs)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`coeffs`: N-dimension NdArray object of coefficients for added terms.

### MLinExpr::Clear()

Clear MLinExpr object.

**Synopsis**

```
void Clear()
```

### MLinExpr::Clone()

Clone MLinExpr object.

**Synopsis**

```
MLinExpr Clone()
```

**Return**

new MLinExpr object.

### MLinExpr::Diagonal()

Get diagonals of MLinExpr object.

**Synopsis**

```
MLinExpr<N - 1> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MLinExpr.

`axis2`: 2nd axis of MLinExpr.

**Return**

(N-1)-dimensional diagonals.

### MLinExpr::Evaluate()

Evaluate MLinExpr object after solving.

**Synopsis**

```
NdArray<double, N> Evaluate()
```

**Return**

NdArray object storing value of each linear expression.

### MLinExpr::Expand()

Expand shape of MLinExpr object.

**Synopsis**

```
MLinExpr<N + 1> Expand(int axis)
```

**Arguments**

`axis`: axis of MLinExpr.

**Return**

MLinExpr object of (N+1)-dimensional shape.

### MLinExpr::Flatten()

Flatten a MLinExpr object to a 1-dimensional shape.

**Synopsis**

```
MLinExpr<1> Flatten()
```

**Return**

a MLinExpr object collapsed into one dimension.

### MLinExpr::GetDim()

Get i-th dimension of MLinExpr object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MLinExpr::GetND()

Get number of dimensions of MLinExpr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MLinExpr::GetShape()

Get shape of MLinExpr object.

**Synopsis**

```
Shape<N> GetShape()
```

**Return**

shape object.

### MLinExpr::GetSize()

Get size of MLinExpr object.

**Synopsis**

```
size_t GetSize()
```

**Return**

number of MExpressions.

### MLinExpr::Item()

Get expression of given index from MLinExpr object.

**Synopsis**

```
MExpression &Item(size_t idx)
```

**Arguments**

`idx`: index of expression.

**Return**

expression object.

### MLinExpr::Item()

Get sub-arrays of MLinExpr object, given view object.

**Synopsis**

```
MLinExpr Item(const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MLinExpr object.

#### MLinExpr::operator[]()

Get expression of given index from MLinExpr object.

**Synopsis**

```
MExpression &operator[](size_t i)
```

**Arguments**

`i`: index of expression.

**Return**

expression object.

#### MLinExpr::operator[]()

Get constraints of given view from MLinExpr object.

**Synopsis**

```
MLinExpr operator[](const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

new MLinExpr object.

#### MLinExpr::Pick()

Given a list of indexes, get linear expressions from MLinExpr object.

**Synopsis**

```
MLinExpr<1> Pick(const NdArray<int, 1> &indexes)
```

**Arguments**

`indexes`: indexes of elements.

**Return**

one-dimensional array of desired linear expressions.

#### MLinExpr::Pick()

Given a list of indexes, get linear expressions from MLinExpr object.

**Synopsis**

```
MLinExpr<1> Pick(const NdArray<int, 2> &idxrows)
```

**Arguments**

`idxrows`: indexes in format of 2-dimensional array, where each row is position of element.

**Return**

one-dimensional array of desired linear expressions.

### MLinExpr::Repeat()

Repeat each element of MLinExpr along given axis.

**Synopsis**

```
MLinExpr<N> Repeat(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of MLinExpr.

**Return**

new MLinExpr object.

### MLinExpr::RepeatBlock()

Repeat an MLinExpr a number of times along given axis.

**Synopsis**

```
MLinExpr<N> RepeatBlock(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of MLinExpr.

**Return**

new MLinExpr object.

### MLinExpr::Represent()

String representation of MLinExpr object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

### MLinExpr::Reshape()

Reshape MLinExpr object to new shape.

**Synopsis**

```
template <int M> MLinExpr<M> Reshape(const Shape<M> &shape)
```

**Arguments**

`shape`: new shape of M-dimensions.

**Return**

M-dimensional MLinExpr object.

### MLinExpr::SetItem()

Set expression of given index to MLinExpr object.

**Synopsis**

```
void SetItem(size_t idx, const MExpression &expr)
```

**Arguments**

`idx`: index of element.

`expr`: MExpression object.

### MLinExpr::Squeeze()

Remove axis of length 1 from shape of MLinExpr object.

**Synopsis**

```
MLinExpr<N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: axis of MLinExpr, where the length is 1.

**Return**

MLinExpr object of (N-1)-dimensional shape.

### MLinExpr::Stack()

Stack with other MLinExpr object along given axis.

**Synopsis**

```
MLinExpr<N> Stack(const MLinExpr<N> &other, int axis)
```

**Arguments**

`other`: a MLinExpr object.

`axis`: an axis of MLinExpr.

**Return**

the result MLinExpr object.

### MLinExpr::Stack()

Stack with other MVar object along given axis.

**Synopsis**

```
MLinExpr<N> Stack(const MVar<N> &other, int axis)
```

**Arguments**

`other`: a MVar object.

`axis`: an axis of MLinExpr.

**Return**

the result MLinExpr object.

### MLinExpr::Stack()

Stack with other NdArray object along given axis.

**Synopsis**

```
template <class T> MLinExpr<N> Stack(const NdArray<T, N> &other, int
axis)
```

**Arguments**

`other`: a NdArray object.

`axis`: an axis of MLinExpr.

**Return**

the result MLinExpr object.

### MLinExpr::SubConstant()

Subtract constants from each expression in MLinExpr object.

**Synopsis**

```
template <class T> void SubConstant(const NdArray<T, N> &constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

### MLinExpr::Sum()

Sum of all expressions in MLinExpr object.

**Synopsis**

```
MLinExpr<0> Sum()
```

**Return**

sum in zero dimension.

### MLinExpr::Sum()

Sum of expressions at given axis of MLinExpr object.

**Synopsis**

```
MLinExpr<N - 1> Sum(int axis)
```

**Arguments**

`axis`: axis of MLinExpr

**Return**

MLinExpr object in (N-1)-dimension.

#### MLinExpr::Transpose()

Perform matrix transpose of MLinExpr object.

**Synopsis**

```
MLinExpr<N> Transpose()
```

**Return**

transposed MLinExpr object.

### 25.5.56 MPsdConstr Class

The *MPsdConstr* class in COPT represents multi-dimensional semidefinite constraints. It is generated through the methods `addConstrs` or `addConstr` of *Model*. The following member methods are provided:

#### MPsdConstr::MPsdConstr()

Construct a MPsdConstr object with the given shape, filling with the given PSD constraint.

**Synopsis**

```
MPsdConstr(const Shape<N> &shp, const PsdConstraint &con)
```

**Arguments**

`shp`: shape of MPsdConstr.

`con`: PSD constraint object.

#### MPsdConstr::MPsdConstr()

Construct a MPsdConstr object with the given shape, filling with an array of PSD constraints.

**Synopsis**

```
MPsdConstr(const Shape<N> &shp, const PsdConstrArray &cons)
```

**Arguments**

`shp`: shape of MPsdConstr.

`cons`: an array of PSD constraints.

#### MPsdConstr::Clone()

Clone MPsdConstr object.

**Synopsis**

```
MPsdConstr Clone()
```

**Return**

new MPsdConstr object.

### MPsdConstr::Diagonal()

Get diagonals of MPsdConstr object.

**Synopsis**

```
MPsdConstr<N - 1> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MPsdConstr.

`axis2`: 2nd axis of MPsdConstr.

**Return**

(N-1)-dimensional diagonals.

### MPsdConstr::Expand()

Expand shape of MPsdConstr object.

**Synopsis**

```
MPsdConstr<N + 1> Expand(int axis)
```

**Arguments**

`axis`: axis of MPsdConstr.

**Return**

MPsdConstr object of (N+1)-dimensional shape.

### MPsdConstr::Flatten()

Flatten a MPsdConstr object to a 1-dimensional shape.

**Synopsis**

```
MPsdConstr<1> Flatten()
```

**Return**

a MPsdConstr object collapsed into one dimension.

### MPsdConstr::Get()

Get values of information associated with PSD constraints in MPsdConstr object.

**Synopsis**

```
NdArray<double, N> Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of information.

**Return**

multi-dimensional array of information of PSD constraints.

### MPsdConstr::GetDim()

Get i-th dimension of MPsdConstr object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MPsdConstr::GetIdx()

Get index of PSD constraints in MPsdConstr object.

**Synopsis**

```
NdArray<int, N> GetIdx()
```

**Return**

multi-dimensional array of indexes of PSD constraints.

### MPsdConstr::GetND()

Get number of dimensions of MPsdConstr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MPsdConstr::GetShape()

Get shape of MPsdConstr object.

**Synopsis**

```
Shape<N> GetShape()
```

**Return**

shape object.

### MPsdConstr::GetSize()

Get size of MPsdConstr object.

**Synopsis**

```
size_t GetSize()
```

**Return**

number of QConstraints

### MPsdConstr::Item()

Get PSD constraint of given index from MPsdConstr object.

**Synopsis**

```
PsdConstraint &Item(size_t idx)
```

**Arguments**

`idx`: index of PSD constraint.

**Return**

PsdConstraint object.

### MPsdConstr::operator[]()

Get PSD constraint of given index from MPsdConstr object.

**Synopsis**

```
PsdConstraint &operator[](size_t idx)
```

**Arguments**

`idx`: index of PSD constraint.

**Return**

PsdConstraint object.

### MPsdConstr::operator[]()

Get constraints of given view from MPsdConstr object.

**Synopsis**

```
MPsdConstr operator[](const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

new MPsdConstr object.

### MPsdConstr::Pick()

Given a list of indexes, get PSD constraints from MPsdConstr object.

**Synopsis**

```
MPsdConstr<1> Pick(const NdArray<int, 1> &indexes)
```

**Arguments**

`indexes`: indexes of elements.

**Return**

one-dimensional array of desired PSD constraints.

### MPsdConstr::Pick()

Given a list of indexes, get PSD constraints from MPsdConstr object.

**Synopsis**

```
MPsdConstr<1> Pick(const NdArray<int, 2> &idxrows)
```

**Arguments**

`idxrows`: indexes in format of 2-dimensional array, where each row is position of element.

**Return**

one-dimensional array of desired PSD constraints.

### MPsdConstr::Represent()

String representation of MPsdConstr object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

### MPsdConstr::Reshape()

Reshape MPsdConstr object to new shape.

**Synopsis**

```
template <int M> MPsdConstr<M> Reshape(const Shape<M> &shape)
```

**Arguments**

`shape`: new shape of M-dimensions.

**Return**

M-dimensional MPsdConstr object.

### MPsdConstr::Set()

Set values of information associated with PSD constraints in MPsdConstr object.

**Synopsis**

```
void Set(const char *szInfo, const NdArray<double, N> &vals)
```

**Arguments**

`szInfo`: name of information.

`vals`: multi-dimensional array of values of information.

### MPsdConstr::Set()

Set values of information associated with PSD constraints in MPsdConstr object.

**Synopsis**

```
void Set(const char *szInfo, double val)
```

**Arguments**

`szInfo`: name of information.

`val`: value of information.

### MPsdConstr::SetItem()

Set PSD constraint of given index to MPsdConstr object.

**Synopsis**

```
void SetItem(size_t idx, const PsdConstraint &con)
```

**Arguments**

`idx`: index of element.

`con`: PSD constraint object.

### MPsdConstr::Squeeze()

Remove axis of length 1 from shape of MPsdConstr object.

**Synopsis**

```
MPsdConstr<N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: axis of MPsdConstr, where the length is 1.

**Return**

MPsdConstr object of (N-1)-dimensional shape.

### MPsdConstr::Stack()

Stack with other MPsdConstr object along given axis.

**Synopsis**

```
MPsdConstr<N> Stack(const MPsdConstr<N> &other, int axis)
```

**Arguments**

`other`: a MPsdConstr object.

`axis`: an axis of MPsdConstr.

**Return**

the result MPsdConstr object.

#### MPsdConstr::Transpose()

Perform matrix transpose of MPsdConstr object.

**Synopsis**

```
MPsdConstr<N> Transpose()
```

**Return**

transposed MPsdConstr object.

### 25.5.57 MPsdConstrBuilder

The *MPsdConstrBuilder* class in COPT serves as a builder for multi-dimensional semidefinite constraints. It is used to generate multi-dimensional semidefinite constraints and supports operations with the built-in multi-dimensional array *NdArray*. An *MPsdConstrBuilder* object can be created through comparison operations between two objects, one of which can be an *MPsdExpr Class* object. The following member methods are provided:

#### MPsdConstrBuilder::MPsdConstrBuilder()

Construct a MPsdConstrBuilder object with the given shape.

**Synopsis**

```
MPsdConstrBuilder(const Shape<N> &shp)
```

**Arguments**

`shp`: shape of MPsdConstrBuilder.

#### MPsdConstrBuilder::Flatten()

Flatten a MPsdConstrBuilder object to a 1-dimensional shape.

**Synopsis**

```
MPsdConstrBuilder<1> Flatten()
```

**Return**

a MPsdConstrBuilder object collapsed into one dimension.

#### MPsdConstrBuilder::GetND()

Get number of dimensions of MPsdConstrBuilder object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MPsdConstrBuilder::GetPsdExpr()

Get N-dimensional PSD expressions associated with N-dimensional PSD constraints.

**Synopsis**

```
const MPsdExpr<N> &GetPsdExpr()
```

**Return**

MPsdExpr object.

### MPsdConstrBuilder::GetRange()

Get range from lower bound to upper bound of N-dimensional range PSD constraints.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of range PSD constraints.

### MPsdConstrBuilder::GetSense()

Get sense associated with N-dimensional PSD constraints.

**Synopsis**

```
char GetSense()
```

**Return**

PSD constraint sense.

### MPsdConstrBuilder::Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
template <int M> void Set(
    const MPsdExpr<N> &exprs,
    char sense,
    const MPsdExpr<M> &rhs)
```

**Arguments**

`exprs`: MPsdExpr object

`sense`: PSD constraint sense other than COPT_RANGE.

`rhs`: MPsdExpr object at right side of PSD constraints.

### MPsdConstrBuilder::Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
void Set(
    const MPsdExpr<N> &exprs,
    char sense,
    double rhs)
```

**Arguments**

`exprs`: MPsdExpr object

`sense`: PSD constraint sense other than COPT_RANGE.

`rhs`: constant of right side of PSD constraints.

### MPsdConstrBuilder::Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
template <class T> void Set(
    const MPsdExpr<N> &exprs,
    char sense,
    const NdArray<T, N> &rhs)
```

**Arguments**

`exprs`: MPsdExpr object

`sense`: PSD constraint sense other than COPT_RANGE.

`rhs`: N-dimensional constants at right side of PSD constraints.

### MPsdConstrBuilder::Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
template <int M> void Set(
    const MPsdExpr<N> &exprs,
    char sense,
    const MVar<M> &rhs)
```

**Arguments**

`exprs`: MPsdExpr object

`sense`: PSD constraint sense other than COPT_RANGE.

`rhs`: MVar object at right side of PSD constraints.

#### MPsdConstrBuilder::Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
template <int M> void Set(
    const MPsdExpr<N> &exprs,
    char sense,
    const MLinExpr<M> &rhs)
```

**Arguments**

`exprs`: MPsdExpr object

`sense`: PSD constraint sense other than COPT_RANGE.

`rhs`: MLinExpr object at right side of PSD constraints.

#### MPsdConstrBuilder::SetRange()

Set N-dimensional range PSD constraints to its builder object.

**Synopsis**

```
void SetRange(const MPsdExpr<N> &exprs, double range)
```

**Arguments**

`exprs`: MPsdExpr object.

`range`: length from lower bound to upper bound of PSD constraint. Must be greater than 0.

### 25.5.58 MPsdExpr Class

The *MPsdExpr* class in COPT represents multi-dimensional semidefinite expressions. It is used to construct multi-dimensional semidefinite expressions and perform operations with the built-in multi-dimensional array *NdArray* in COPT.

The elements of *MPsdExpr* are either *PsdExpr* objects or their multi-dimensional linear combinations.

The following member methods are provided:

#### MPsdExpr::MPsdExpr()

Construct a MPsdExpr object with the given shape and a constant.

**Synopsis**

```
MPsdExpr(const Shape<N> &shp, double constant)
```

**Arguments**

`shp`: shape of MPsdExpr.

`constant`: constant number, default value is 0.0.

#### MPsdExpr::MPsdExpr()

Construct a MPsdExpr object with the given shape and a PSD expression.

**Synopsis**

```
MPsdExpr(const Shape<N> &shp, const PsdExpr &expr)
```

**Arguments**

`shp`: shape of MPsdExpr.

`expr`: a PSD expression.

#### MPsdExpr::AddConstant()

Add constant to each quadratic expression in MPsdExpr object.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### MPsdExpr::AddConstant()

Add constants to each PSD expression in MPsdExpr object.

**Synopsis**

```
template <class T> void AddConstant(const NdArray<T, N> &constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

#### MPsdExpr::AddLinExpr()

Add a linear expression to each PsdExpr in MPsdExpr object.

**Synopsis**

```
void AddLinExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

#### MPsdExpr::AddMExpr()

Add MExpression to each PSD expression in MPsdExpr object.

**Synopsis**

```
void AddMExpr(const MExpression &expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MPsdExpr::AddMLinExpr()

Add linear expressions to MPsdExpr object.

**Synopsis**

```
void AddMLinExpr(const MLinExpr<N> &exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MLinExpr object.

`mult`: the same multiplier for added linear expressions, default value is 1.0.

### MPsdExpr::AddMPsdExpr()

Add PSD expressions to MPsdExpr object.

**Synopsis**

```
void AddMPsdExpr(const MPsdExpr<N> &exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MPsdExpr object.

`mult`: the same multiplier for added PSD expressions, default value is 1.0.

### MPsdExpr::AddPsdExpr()

Add a PSD expression to each PSD expression in MPsdExpr object.

**Synopsis**

```
void AddPsdExpr(const PsdExpr &expr, double mult)
```

**Arguments**

`expr`: PSD expression object.

`mult`: the multiplier of PSD expression, default value is 1.0.

### MPsdExpr::AddTerm()

Add a linear term to MPsdExpr object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

### MPsdExpr::AddTerm()

Add a PSD term to MPsdExpr object.

**Synopsis**

```
void AddTerm(const PsdVar &var, const SymMatrix &mat)
```

**Arguments**

`var`: PSD variable of new PSD term.

`mat`: coefficient matrix of new PSD term.

### MPsdExpr::AddTerm()

Add a PSD term to MPsdExpr object.

**Synopsis**

```
void AddTerm(const PsdVar &var, const SymMatExpr &expr)
```

**Arguments**

`var`: PSD variable of new PSD term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

### MPsdExpr::AddTerms()

Add terms to PSD expressions in MPsdExpr object.

**Synopsis**

```
void AddTerms(const MVar<N> &vars, double mult)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`mult`: the same coefficient for added terms, default value 1.0.

### MPsdExpr::AddTerms()

Add terms to PSD expressions in MPsdExpr object.

**Synopsis**

```
void AddTerms(const MVar<N> &vars, const NdArray<double, N> &coeffs)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`coeffs`: N-dimension NdArray object of coefficients for added terms.

### MPsdExpr::Clear()

Clear MPsdExpr object.

**Synopsis**

```
void Clear()
```

### MPsdExpr::Clone()

Clone MPsdExpr object.

**Synopsis**

```
MPsdExpr Clone()
```

**Return**

new MPsdExpr object.

### MPsdExpr::Diagonal()

Get diagonals of MPsdExpr object.

**Synopsis**

```
MPsdExpr<N - 1> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MPsdExpr.

`axis2`: 2nd axis of MPsdExpr.

**Return**

(N-1)-dimensional diagonals.

### MPsdExpr::Evaluate()

Evaluate MPsdExpr object after solving.

**Synopsis**

```
NdArray<double, N> Evaluate()
```

**Return**

NdArray object storing value of each PSD expression.

### MPsdExpr::Expand()

Expand shape of MPsdExpr object.

**Synopsis**

```
MPsdExpr<N + 1> Expand(int axis)
```

**Arguments**

`axis`: axis of MPsdExpr.

**Return**

MPsdExpr object of (N+1)-dimensional shape.

### MPsdExpr::Flatten()

Flatten a MPsdExpr object to a 1-dimensional shape.

**Synopsis**

```
MPsdExpr<1> Flatten()
```

**Return**

a MPsdExpr object collapsed into one dimension.

### MPsdExpr::GetDim()

Get i-th dimension of MPsdExpr object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MPsdExpr::GetND()

Get number of dimensions of MPsdExpr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MPsdExpr::GetShape()

Get shape of MPsdExpr object.

**Synopsis**

```
Shape<N> GetShape()
```

**Return**

shape object.

### MPsdExpr::GetSize()

Get size of MPsdExpr object.

**Synopsis**

```
size_t GetSize()
```

**Return**

number of PSD expressions.

### MPsdExpr::Item()

Get PSD expression of given index from MPsdExpr object.

**Synopsis**

```
PsdExpr &Item(size_t idx)
```

**Arguments**

`idx`: index of PSD expression.

**Return**

PSD expression object.

### MPsdExpr::Item()

Get sub-arrays of MPsdExpr object, given view object.

**Synopsis**

```
MPsdExpr Item(const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MPsdExpr object.

### MPsdExpr::operator[]()

Get PSD expression of given index from MPsdExpr object.

**Synopsis**

```
PsdExpr &operator[](size_t i)
```

**Arguments**

`i`: index of PSD expression.

**Return**

PSD expression object.

### MPsdExpr::operator[]()

Get PSD expressions of given view from MPsdExpr object.

**Synopsis**

```
MPsdExpr operator[](const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

new MPsdExpr object.

### MPsdExpr::Pick()

Given a list of indexes, get PSD expressions from MPsdExpr object.

**Synopsis**

```
MPsdExpr<1> Pick(const NdArray<int, 1> &indexes)
```

**Arguments**

`indexes`: indexes of elements.

**Return**

one-dimensional array of desired PSD expressions.

### MPsdExpr::Pick()

Given a list of indexes, get PSD expressions from MPsdExpr object.

**Synopsis**

```
MPsdExpr<1> Pick(const NdArray<int, 2> &idxrows)
```

**Arguments**

`idxrows`: indexes in format of 2-dimensional array, where each row is position of element.

**Return**

one-dimensional array of desired PSD expressions.

### MPsdExpr::Repeat()

Repeat each element of MPsdExpr along given axis.

**Synopsis**

```
MPsdExpr<N> Repeat(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of MPsdExpr.

**Return**

new MPsdExpr object.

### MPsdExpr::RepeatBlock()

Repeat an MPsdExpr a number of times along given axis.

**Synopsis**

```
MPsdExpr<N> RepeatBlock(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of MPsdExpr.

**Return**

new MPsdExpr object.

### MPsdExpr::Represent()

String representation of MPsdExpr object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

### MPsdExpr::Reshape()

Reshape MPsdExpr object to new shape.

**Synopsis**

```
template <int M> MPsdExpr<M> Reshape(const Shape<M> &shape)
```

**Arguments**

`shape`: new shape of M-dimensions.

**Return**

M-dimensional MPsdExpr object.

### MPsdExpr::SetItem()

Set PSD expression of given index to MPsdExpr object.

**Synopsis**

```
void SetItem(size_t idx, const PsdExpr &expr)
```

**Arguments**

`idx`: index of element.

`expr`: PSD expression object.

### MPsdExpr::Squeeze()

Remove axis of length 1 from shape of MPsdExpr object.

**Synopsis**

```
MPsdExpr<N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: axis of MPsdExpr, where the length is 1.

**Return**

MPsdExpr object of (N-1)-dimensional shape.

### MPsdExpr::Stack()

Stack with other MPsdExpr object along given axis.

**Synopsis**

```
MPsdExpr<N> Stack(const MPsdExpr<N> &other, int axis)
```

**Arguments**

`other`: a MPsdExpr object.

`axis`: an axis of MPsdExpr.

**Return**

the result MPsdExpr object.

### MPsdExpr::Stack()

Stack with other MLinExpr object along given axis.

**Synopsis**

```
MPsdExpr<N> Stack(const MLinExpr<N> &other, int axis)
```

**Arguments**

`other`: a MLinExpr object.

`axis`: an axis of MPsdExpr.

**Return**

the result MPsdExpr object.

### MPsdExpr::Stack()

Stack with other MVar object along given axis.

**Synopsis**

```
MPsdExpr<N> Stack(const MVar<N> &other, int axis)
```

**Arguments**

`other`: a MVar object.

`axis`: an axis of MPsdExpr.

**Return**

the result MPsdExpr object.

### MPsdExpr::Stack()

Stack with other NdArray object along given axis.

**Synopsis**

```
template <class T> MPsdExpr<N> Stack(const NdArray<T, N> &other, int
axis)
```

**Arguments**

`other`: a NdArray object.

`axis`: an axis of MPsdExpr.

**Return**

the result MPsdExpr object.

### MPsdExpr::SubConstant()

Subtract constants from each PSD expression in MPsdExpr object.

**Synopsis**

```
template <class T> void SubConstant(const NdArray<T, N> &constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

### MPsdExpr::Sum()

Sum of all quadratic expressions in MPsdExpr object.

**Synopsis**

```
MPsdExpr<0> Sum()
```

**Return**

sum in zero dimension.

#### MPsdExpr::Sum()

Sum of quadratic expressions at given axis of MPsdExpr object.

**Synopsis**

```
MPsdExpr<N - 1> Sum(int axis)
```

**Arguments**

`axis`: axis of MPsdExpr.

**Return**

MPsdExpr object in (N-1)-dimension.

#### MPsdExpr::Transpose()

Perform matrix transpose of MPsdExpr object.

**Synopsis**

```
MPsdExpr<N> Transpose()
```

**Return**

transposed MPsdExpr object.

### 25.5.59 MQConstr

The MQConstr class is a COPT multi-dimensional quadratic constraint object. It can be created by calling the method `addMQConstr` of *Model*. The following methods are provided:

#### MQConstr::MQConstr()

Construct a MQConstr object with the given shape, filling with the given quadratic constraint.

**Synopsis**

```
MQConstr(const Shape<N> &shp, const QConstraint &con)
```

**Arguments**

`shp`: shape of MQConstr.

`con`: Quadratic constraint object.

#### MQConstr::MQConstr()

Construct a MQConstr object with the given shape, filling with an array of quadratic constraints.

**Synopsis**

```
MQConstr(const Shape<N> &shp, const QConstrArray &cons)
```

**Arguments**

`shp`: shape of MQConstr.

`cons`: an array of quadratic constraints.

### MQConstr::Clone()

Clone MQConstr object.

**Synopsis**

```
MQConstr Clone()
```

**Return**

new MQConstr object.

### MQConstr::Diagonal()

Get diagonals of MQConstr object.

**Synopsis**

```
MQConstr<N - 1> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MQConstr.

`axis2`: 2nd axis of MQConstr.

**Return**

(N-1)-dimensional diagonals.

### MQConstr::Expand()

Expand shape of MQConstr object.

**Synopsis**

```
MQConstr<N + 1> Expand(int axis)
```

**Arguments**

`axis`: axis of MQConstr.

**Return**

MQConstr object of (N+1)-dimensional shape.

### MQConstr::Flatten()

Flatten a MQConstr object to a 1-dimensional shape.

**Synopsis**

```
MQConstr<1> Flatten()
```

**Return**

a MQConstr object collapsed into one dimension.

### MQConstr::Get()

Get values of information associated with quadratic constraints in MQConstr object.

**Synopsis**

```
NdArray<double, N> Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of information.

**Return**

multi-dimensional array of information of quadratic constraints.

### MQConstr::GetDim()

Get i-th dimension of MQConstr object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MQConstr::GetIdx()

Get index of quadratic constraints in MQConstr object.

**Synopsis**

```
NdArray<int, N> GetIdx()
```

**Return**

multi-dimensional array of indexes of quadratic constraints.

### MQConstr::GetND()

Get number of dimensions of MQConstr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MQConstr::GetRhs()

Get RHS of quadratic constraints in MQConstr object.

**Synopsis**

```
NdArray<double, N> GetRhs()
```

**Return**

multi-dimensional array of RHS of quadratic constraints.

### MQConstr::GetSense()

Get senses of quadratic constraints in MQConstr object.

**Synopsis**

```
NdArray<char, N> GetSense()
```

**Return**

multi-dimensional array of senses of quadratic constraints.

### MQConstr::GetShape()

Get shape of MQConstr object.

**Synopsis**

```
Shape<N> GetShape()
```

**Return**

shape object.

### MQConstr::GetSize()

Get size of MQConstr object.

**Synopsis**

```
size_t GetSize()
```

**Return**

number of QConstraints

### MQConstr::Item()

Get quadratic constraint of given index from MQConstr object.

**Synopsis**

```
QConstraint &Item(size_t idx)
```

**Arguments**

`idx`: index of quadratic constraint.

**Return**

QConstraint object.

### MQConstr::operator[]()

Get quadratic constraint of given index from MQConstr object.

**Synopsis**

```
QConstraint &operator[](size_t idx)
```

**Arguments**

`idx`: index of quadratic constraint.

**Return**

QConstraint object.

### MQConstr::operator[]()

Get constraints of given view from MQConstr object.

**Synopsis**

```
MQConstr operator[](const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

new MQConstr object.

### MQConstr::Pick()

Given a list of indexes, get quadratic constraints from MQConstr object.

**Synopsis**

```
MQConstr<1> Pick(const NdArray<int, 1> &indexes)
```

**Arguments**

`indexes`: indexes of elements.

**Return**

one-dimensional array of desired quadratic constraints.

### MQConstr::Pick()

Given a list of indexes, get quadratic constraints from MQConstr object.

**Synopsis**

```
MQConstr<1> Pick(const NdArray<int, 2> &idxrows)
```

**Arguments**

`idxrows`: indexes in format of 2-dimensional array, where each row is position of element.

**Return**

one-dimensional array of desired quadratic constraints.

### MQConstr::Represent()

String representation of MQConstr object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

### MQConstr::Reshape()

Reshape MQConstr object to new shape.

**Synopsis**

```
template <int M> MQConstr<M> Reshape(const Shape<M> &shape)
```

**Arguments**

`shape`: new shape of M-dimensions.

**Return**

M-dimensional MQConstr object.

### MQConstr::Set()

Set values of information associated with quadratic constraints in MQConstr object.

**Synopsis**

```
void Set(const char *szInfo, double val)
```

**Arguments**

`szInfo`: name of information.

`val`: value of information.

### MQConstr::Set()

Set values of information associated with quadratic constraints in MQConstr object.

**Synopsis**

```
void Set(const char *szInfo, const NdArray<double, N> &vals)
```

**Arguments**

`szInfo`: name of information.

`vals`: multi-dimensional array of values of information.

### MQConstr::SetItem()

Set quadratic constraint of given index to MQConstr object.

**Synopsis**

```
void SetItem(size_t idx, const QConstraint &con)
```

**Arguments**

`idx`: index of element.

`con`: quadratic constraint object.

### MQConstr::Squeeze()

Remove axis of length 1 from shape of MQConstr object.

**Synopsis**

```
MQConstr<N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: axis of MQConstr, where the length is 1.

**Return**

MQConstr object of (N-1)-dimensional shape.

### MQConstr::Stack()

Stack with other MQConstr object along given axis.

**Synopsis**

```
MQConstr<N> Stack(const MQConstr<N> &other, int axis)
```

**Arguments**

`other`: a MQConstr object.

`axis`: an axis of MQConstr.

**Return**

the result MQConstr object.

### MQConstr::Transpose()

Perform matrix transpose of MQConstr object.

**Synopsis**

```
MQConstr<N> Transpose()
```

**Return**

transposed MQConstr object.

### 25.5.60 MQConstrBuilder

The MQConstrBuilder class is a COPT builder object of multi-dimensional quadratic constraints. It is used to generate multi-dimensional quadratic constraints and supports operations with the built-in multi-dimensional array *NdArray* in COPT. It can be created by comparing two objects, one of which should be *MQuadExpr* object, by comparison operators. The following methods are provided:

#### MQConstrBuilder::MQConstrBuilder()

Construct a MQConstrBuilder object with the given shape.

**Synopsis**

```
MQConstrBuilder(const Shape<N> &shp)
```

**Arguments**

`shp`: shape of MQConstrBuilder.

#### MQConstrBuilder::Flatten()

Flatten a MQConstrBuilder object to a 1-dimensional shape.

**Synopsis**

```
MQConstrBuilder<1> Flatten()
```

**Return**

a MQConstrBuilder object collapsed into one dimension.

#### MQConstrBuilder::GetND()

Get number of dimensions of MQConstrBuilder object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

#### MQConstrBuilder::GetQuadExpr()

Get N-dimensional quadratic expressions associated with N-dimensional quadratic constraints.

**Synopsis**

```
const MQuadExpr<N> &GetQuadExpr()
```

**Return**

MQuadExpr object.

### MQConstrBuilder::GetSense()

Get sense associated with N-dimensional quadratic constraints.

**Synopsis**

```
char GetSense()
```

**Return**

quadratic constraint sense.

### MQConstrBuilder::Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
void Set(
    const MQuadExpr<N> &expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of quadratic constraints.

### MQConstrBuilder::Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
template <class T> void Set(
    const MQuadExpr<N> &expr,
    char sense,
    const NdArray<T, N> &rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: N-dimensional constants at right side of quadratic constraints.

### MQConstrBuilder::Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
template <int M> void Set(
    const MQuadExpr<N> &expr,
    char sense,
    const MVar<M> &rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MVar object at right side of quadratic constraints.

#### MQConstrBuilder::Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
template <int M> void Set(
    const MQuadExpr<N> &expr,
    char sense,
    const MLinExpr<M> &rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MLinExpr object at right side of quadratic constraints.

#### MQConstrBuilder::Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
template <int M> void Set(
    const MQuadExpr<N> &expr,
    char sense,
    const MQuadExpr<M> &rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MQuadExpr object at right side of quadratic constraints.

### 25.5.61 MQExpression

The MQExpression class is a generalized version of *QuadExpr*. It represents a quadratic expression and supports most of methods in QuadExpr class. In addition, it supports quadratic combination of multi-dimensional objects, such as *MVar* object and *MLinExpr* object. The following methods are provided:

#### MQExpression::MQExpression()

Construct a MQExpression object with the given constant.

**Synopsis**

```
MQExpression(double constant)
```

**Arguments**

`constant`: constant number.

#### MQExpression::MQExpression()

Construct a MQExpression object with the given quadratic expression.

**Synopsis**

```
MQExpression(const QuadExpr &expr)
```

**Arguments**

`expr`: a quadratic expression.

#### MQExpression::AddConstant()

Add constant for the expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### MQExpression::AddExpr()

Add a linear expression to MQExpression object.

**Synopsis**

```
void AddExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

#### MQExpression::AddMExpr()

Add MExpression to MQExpression object.

**Synopsis**

```
void AddMExpr(const MExpression &expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MQExpression::AddMQExpr()

Add MQExpression to MQExpression object.

**Synopsis**

```
void AddMQExpr(const MQExpression &expr, double mult)
```

**Arguments**

`expr`: MQExpression object.

`mult`: the multiplier of MQExpression, default value is 1.0.

### MQExpression::AddQuadExpr()

Add a quadratic expression to MQExpression object.

**Synopsis**

```
void AddQuadExpr(const MExpression &left, const MExpression &right)
```

**Arguments**

`left`: left MExpression object.

`right`: right MExpression object.

### MQExpression::AddQuadExpr()

Add a quadratic expression to MQExpression object.

**Synopsis**

```
void AddQuadExpr(const QuadExpr &expr, double mult)
```

**Arguments**

`expr`: quadratic expression object.

`mult`: the multiplier of quadratic expression, default value is 1.0.

### MQExpression::AddQuadExpr()

Add a quadratic expression to MQExpression object.

**Synopsis**

```
void AddQuadExpr(const MExpression &expr, const Var &var)
```

**Arguments**

`expr`: MExpression object.

`var`: Var object.

#### MQExpression::AddQuadExpr()

Add a quadratic expression to MQExpression object.

**Synopsis**

```
void AddQuadExpr(const MExpression &left, const Expr &right)
```

**Arguments**

`left`: MExpression object.

`right`: Expr object.

#### MQExpression::AddTerm()

Add a linear term to MQExpression object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

#### MQExpression::AddTerm()

Add a quadratic term to MQExpression object.

**Synopsis**

```
void AddTerm(
    const Var &var1,
    const Var &var2,
    double coeff)
```

**Arguments**

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

#### MQExpression::Clone()

Clone MQExpression object.

**Synopsis**

```
MQExpression Clone()
```

**Return**

new MQExpression object.

#### MQExpression::Evaluate()

Evaluate MQExpression after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of MQExpression object.

#### MQExpression::GetConstant()

Get constant in expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in expression.

#### MQExpression::Represent()

String representation of MQExpression object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

### 25.5.62 MQuadExpr

COPT multi-dimensional quadratic expression object.It is used to construct multi-dimensional quadratic expressions and perform operations with the multi-dimensional array built in COPT *NdArray* . Its elements are *MQExpression* objects. It can be created by quadratic combination of *MVar* objects. The following methods are provided:

#### MQuadExpr::MQuadExpr()

Construct a MQuadExpr object with the given shape and a constant.

**Synopsis**

```
MQuadExpr(const Shape<N> &shp, double constant)
```

**Arguments**

`shp`: shape of MQuadExpr.

`constant`: constant number.

#### MQuadExpr::MQuadExpr()

Construct a MQuadExpr object with the given shape and a quadratic expression.

**Synopsis**

```
MQuadExpr(const Shape<N> &shp, const QuadExpr &expr)
```

**Arguments**

`shp`: shape of MQuadExpr.

`expr`: a quadratic expression.

#### MQuadExpr::MQuadExpr()

Construct a MQuadExpr object with the given shape and a MQExpression object.

**Synopsis**

```
MQuadExpr(const Shape<N> &shp, const MQExpression &expr)
```

**Arguments**

`shp`: shape of MQuadExpr.

`expr`: a MQExpression object.

#### MQuadExpr::AddConstant()

Add constant to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### MQuadExpr::AddConstant()

Add constants to each quadratic expression in MQuadExpr object.

**Synopsis**

```
template <class T> void AddConstant(const NdArray<T, N> &constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

#### MQuadExpr::AddExpr()

Add a linear expression to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

### MQuadExpr::AddMExpr()

Add MExpression to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddMExpr(const MExpression &expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MQuadExpr::AddMLinExpr()

Add linear expressions to MQuadExpr object.

**Synopsis**

```
void AddMLinExpr(const MLinExpr<N> &exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MLinExpr object.

`mult`: the same multiplier for added linear expressions, default value is 1.0.

### MQuadExpr::AddMQExpr()

Add MQExpression to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddMQExpr(const MQExpression &expr, double mult)
```

**Arguments**

`expr`: MQExpression object.

`mult`: the multiplier of MQExpression, default value is 1.0.

### MQuadExpr::AddMQuadExpr()

Add quadratic expressions to MQuadExpr object.

**Synopsis**

```
void AddMQuadExpr(const MQuadExpr<N> &exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MQuadExpr object.

`mult`: the same multiplier for added quadratic expressions, default value is 1.0.

#### MQuadExpr::AddQuadExpr()

Add a quadratic expression to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddQuadExpr(const QuadExpr &expr, double mult)
```

**Arguments**

`expr`: quadratic expression object.

`mult`: the multiplier of quadratic expression, default value is 1.0.

#### MQuadExpr::AddTerm()

Add a linear term to MQuadExpr object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

#### MQuadExpr::AddTerm()

Add a quadratic term to MQuadExpr object.

**Synopsis**

```
void AddTerm(
    const Var &var1,
    const Var &var2,
    double coeff)
```

**Arguments**

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

#### MQuadExpr::AddTerms()

Add terms to quadratic expressions in MQuadExpr object.

**Synopsis**

```
void AddTerms(const MVar<N> &vars, double mult)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`mult`: the same coefficient for added terms, default value 1.0.

### MQuadExpr::AddTerms()

Add terms to quadratic expressions in MQuadExpr object.

**Synopsis**

```
void AddTerms(const MVar<N> &vars, const NdArray<double, N> &coeffs)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`coeffs`: N-dimension NdArray object of coefficients for added terms.

### MQuadExpr::Clear()

Clear MQuadExpr object.

**Synopsis**

```
void Clear()
```

### MQuadExpr::Clone()

Clone MQuadExpr object.

**Synopsis**

```
MQuadExpr Clone()
```

**Return**

new MQuadExpr object.

### MQuadExpr::Diagonal()

Get diagonals of MQuadExpr object.

**Synopsis**

```
MQuadExpr<N - 1> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MQuadExpr.

`axis2`: 2nd axis of MQuadExpr.

**Return**

(N-1)-dimensional diagonals.

### MQuadExpr::Evaluate()

Evaluate MQuadExpr object after solving.

**Synopsis**

```
NdArray<double, N> Evaluate()
```

**Return**

NdArray object storing value of each quadratic expression.

### MQuadExpr::Expand()

Expand shape of MQuadExpr object.

**Synopsis**

```
MQuadExpr<N + 1> Expand(int axis)
```

**Arguments**

`axis`: axis of MQuadExpr.

**Return**

MQuadExpr object of (N+1)-dimensional shape.

### MQuadExpr::Flatten()

Flatten a MQuadExpr object to a 1-dimensional shape.

**Synopsis**

```
MQuadExpr<1> Flatten()
```

**Return**

a MQuadExpr object collapsed into one dimension.

### MQuadExpr::GetDim()

Get i-th dimension of MQuadExpr object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MQuadExpr::GetND()

Get number of dimensions of MQuadExpr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MQuadExpr::GetShape()

Get shape of MQuadExpr object.

**Synopsis**

```
Shape<N> GetShape()
```

**Return**

shape object.

### MQuadExpr::GetSize()

Get size of MQuadExpr object.

**Synopsis**

```
size_t GetSize()
```

**Return**

number of MQExpressions.

### MQuadExpr::Item()

Get quadratic expression of given index from MQuadExpr object.

**Synopsis**

```
MQExpression &Item(size_t idx)
```

**Arguments**

`idx`: index of quadratic expression.

**Return**

quadratic expression object.

### MQuadExpr::Item()

Get sub-arrays of MQuadExpr object, given view object.

**Synopsis**

```
MQuadExpr Item(const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MQuadExpr object.

#### MQuadExpr::operator[]()

Get quadratic expression of given index from MQuadExpr object.

**Synopsis**

```
MQExpression &operator[](size_t i)
```

**Arguments**

`i`: index of quadratic expression.

**Return**

quadratic expression object.

#### MQuadExpr::operator[]()

Get constraints of given view from MQuadExpr object.

**Synopsis**

```
MQuadExpr operator[](const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

new MQuadExpr object.

#### MQuadExpr::Pick()

Given a list of indexes, get quadratic expressions from MQuadExpr object.

**Synopsis**

```
MQuadExpr<1> Pick(const NdArray<int, 1> &indexes)
```

**Arguments**

`indexes`: indexes of elements.

**Return**

one-dimensional array of desired quadratic expressions.

#### MQuadExpr::Pick()

Given a list of indexes, get quadratic expressions from MQuadExpr object.

**Synopsis**

```
MQuadExpr<1> Pick(const NdArray<int, 2> &idxrows)
```

**Arguments**

`idxrows`: indexes in format of 2-dimensional array, where each row is position of element.

**Return**

one-dimensional array of desired quadratic expressions.

### MQuadExpr::Repeat()

Repeat each element of MQuadExpr along given axis.

**Synopsis**

```
MQuadExpr<N> Repeat(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of MQuadExpr.

**Return**

new MQuadExpr object.

### MQuadExpr::RepeatBlock()

Repeat an MQuadExpr a number of times along given axis.

**Synopsis**

```
MQuadExpr<N> RepeatBlock(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of MQuadExpr.

**Return**

new MQuadExpr object.

### MQuadExpr::Represent()

String representation of MQuadExpr object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation.

**Return**

string object.

### MQuadExpr::Reshape()

Reshape MQuadExpr object to new shape.

**Synopsis**

```
template <int M> MQuadExpr<M> Reshape(const Shape<M> &shape)
```

**Arguments**

`shape`: new shape of M-dimensions.

**Return**

M-dimensional MQuadExpr object.

#### MQuadExpr::SetItem()

Set quadratic expression of given index to MQuadExpr object.

**Synopsis**

```
void SetItem(size_t idx, const MQExpression &expr)
```

**Arguments**

`idx`: index of element.

`expr`: MQExpression object.

#### MQuadExpr::Squeeze()

Remove axis of length 1 from shape of MQuadExpr object.

**Synopsis**

```
MQuadExpr<N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: axis of MQuadExpr, where the length is 1.

**Return**

MQuadExpr object of (N-1)-dimensional shape.

#### MQuadExpr::Stack()

Stack with other MQuadExpr object along given axis.

**Synopsis**

```
MQuadExpr<N> Stack(const MQuadExpr<N> &other, int axis)
```

**Arguments**

`other`: a MQuadExpr object.

`axis`: an axis of MQuadExpr.

**Return**

the result MQuadExpr object.

#### MQuadExpr::Stack()

Stack with other MLinExpr object along given axis.

**Synopsis**

```
MQuadExpr<N> Stack(const MLinExpr<N> &other, int axis)
```

**Arguments**

`other`: a MLinExpr object.

`axis`: an axis of MQuadExpr.

**Return**

the result MQuadExpr object.

### MQuadExpr::Stack()

Stack with other MVar object along given axis.

**Synopsis**

```
MQuadExpr<N> Stack(const MVar<N> &other, int axis)
```

**Arguments**

`other`: a MVar object.

`axis`: an axis of MQuadExpr.

**Return**

the result MQuadExpr object.

### MQuadExpr::Stack()

Stack with other NdArray object along given axis.

**Synopsis**

```
template <class T> MQuadExpr<N> Stack(const NdArray<T, N> &other,
int axis)
```

**Arguments**

`other`: a NdArray object.

`axis`: an axis of MQuadExpr.

**Return**

the result MQuadExpr object.

### MQuadExpr::SubConstant()

Subtract constants from each quadratic expression in MQuadExpr object.

**Synopsis**

```
template <class T> void SubConstant(const NdArray<T, N> &constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

### MQuadExpr::Sum()

Sum of all quadratic expressions in MQuadExpr object.

**Synopsis**

```
MQuadExpr<0> Sum()
```

**Return**

sum in zero dimension.

#### MQuadExpr::Sum()

Sum of quadratic expressions at given axis of MQuadExpr object.

**Synopsis**

```
MQuadExpr<N - 1> Sum(int axis)
```

**Arguments**

`axis`: axis of MQuadExpr.

**Return**

MQuadExpr object in (N-1)-dimension.

#### MQuadExpr::Transpose()

Perform matrix transpose of MQuadExpr object.

**Synopsis**

```
MQuadExpr<N> Transpose()
```

**Return**

transposed MQuadExpr object.

### 25.5.63 NlExpr Class

The `NlExpr` class represents nonlinear expressions in COPT. The following methods are provided:

#### NlExpr::NlExpr()

Constructor of a nonlinear expression with a constant.

**Synopsis**

```
NlExpr(double constant)
```

**Arguments**

`constant`: constant value in nonlinear expression object.

#### NlExpr::NlExpr()

Constructor of a nonlinear expression with one linear term.

**Synopsis**

```
NlExpr(const Var &var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

### NlExpr::NlExpr()

Constructor of a nonlinear expression with a linear expression.

**Synopsis**

```
NlExpr(const Expr &expr)
```

**Arguments**

`expr`: linear expression.

### NlExpr::NlExpr()

Constructor of a nonlinear expression with a quadratic expression.

**Synopsis**

```
NlExpr(const QuadExpr &expr)
```

**Arguments**

`expr`: quadratic expression.

### NlExpr::AddConstant()

Add constant to the nonlinear expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: value to be added.

### NlExpr::AddLinExpr()

Add a linear expression to self.

**Synopsis**

```
void AddLinExpr(const Expr &expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### NlExpr::AddNlExpr()

Add a nonlinear expression to self.

**Synopsis**

```
void AddNlExpr(const NlExpr &expr, double mult)
```

**Arguments**

`expr`: nonlinear expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### NlExpr::AddQuadExpr()

Add a quadratic expression to self.

**Synopsis**

```
void AddQuadExpr(const QuadExpr &expr, double mult)
```

**Arguments**

`expr`: quadratic expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### NlExpr::AddTerm()

Add a linear term to nonlinear expression object.

**Synopsis**

```
void AddTerm(const Var &var, double coeff)
```

**Arguments**

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

### NlExpr::AddTerms()

Add linear terms to nonlinear expression object.

**Synopsis**

```
int AddTerms(
    const VarArray &vars,
    double *pCoeff,
    int len)
```

**Arguments**

`vars`: variables for added linear terms.

`pCoeff`: coefficient array for added linear terms.

`len`: length of coefficient array.

**Return**

number of added linear terms.

### NlExpr::Clear()

Clear nonlinear expression object.

**Synopsis**

```
void Clear()
```

### NlExpr::Clone()

Deep copy nonlinear expression object.

**Synopsis**

```
NlExpr Clone()
```

**Return**

cloned nonlinear expression object.

### NlExpr::Evaluate()

Evaluate nonlinear expression after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of nonlinear expression.

### NlExpr::GetConstant()

Get constant in nonlinear expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in nonlinear expression.

### NlExpr::GetLinExpr()

Get linear expression in nonlinear expression.

**Synopsis**

```
Expr &GetLinExpr()
```

**Return**

linear expression object.

### NlExpr::Negate()

Negate self.

**Synopsis**

```
void Negate()
```

### NlExpr::operator*=()

Multiply a nonlinear expression to self.

**Synopsis**

```
void operator*=(const NlExpr &expr)
```

**Arguments**

`expr`: nonlinear expression for multiplication, including double, Var, Expr, QuadExpr and NlExpr.

### NlExpr::operator*()

Multiply expression and return new nonlinear expression.

**Synopsis**

```
NlExpr operator*(const NlExpr &other)
```

**Arguments**

`other`: operand of multiplication, including double, Var, Expr, QuadExpr and NlExpr.

**Return**

result expression.

### NlExpr::operator/=()

Divide a nonlinear expression by self.

**Synopsis**

```
void operator/=(const NlExpr &expr)
```

**Arguments**

`expr`: nonlinear expression for multiplication, including double, Var, Expr, QuadExpr and NlExpr.

### NlExpr::operator/()

Divide an expression and return new nonlinear expression.

**Synopsis**

```
NlExpr operator/(const NlExpr &other)
```

**Arguments**

`other`: operand of division, including double, Var, Expr, QuadExpr and NlExpr.

**Return**

result expression.

### NlExpr::operator+=()

Add an expression to self.

**Synopsis**

```
void operator+=(const NlExpr &expr)
```

**Arguments**

`expr`: nonlinear expression for addition, including double, Var, Expr, QuadExpr and NlExpr.

### NlExpr::operator+()

Add expression and return new expression.

**Synopsis**

```
NlExpr operator+(const NlExpr &other)
```

**Arguments**

`other`: operand of addition, including double, Var, Expr, QuadExpr and NlExpr.

**Return**

result expression.

### NlExpr::operator-=()

Subtract an expression from self.

**Synopsis**

```
void operator-=(const NlExpr &expr)
```

**Arguments**

`expr`: nonlinear expression for subtraction, including double, Var, Expr, QuadExpr and NlExpr.

### NlExpr::operator-()

Subtract expression and return new expression.

**Synopsis**

```
NlExpr operator-(const NlExpr &other)
```

**Arguments**

`other`: operand of subtraction, including double, Var, Expr, QuadExpr and NlExpr.

**Return**

result expression.

#### NlExpr::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(size_t n)
```

**Arguments**

`n`: minimum capacity for nonlinear expression object.

#### NlExpr::SetConstant()

Set constant for the nonlinear expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### NlExpr::Size()

Get size of tokens in nonlinear expression.

**Synopsis**

```
size_t Size()
```

**Return**

size of none-linear tokens.

### 25.5.64 NlConstraint Class

The `NlConstraint` class provides an interface for operations on nonlinear constraints in COPT. The following methods are provided:

#### NlConstraint::Get()

Get information value of the constraint. Support information of "LB", "UB" and "Slack".

**Synopsis**

```
double Get(const char *szInfo)
```

**Arguments**

`szInfo`: name of the information being queried.

**Return**

value of information.

### NlConstraint::GetIdx()

Get index of nonlinear constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

the index of nonlinear constraint.

### NlConstraint::GetName()

Get name of nonlinear constraint.

**Synopsis**

```
const char *GetName()
```

**Return**

the name of nonlinear constraint.

### NlConstraint::Remove()

Remove this nonlinear constraint from model.

**Synopsis**

```
void Remove()
```

### NlConstraint::Set()

Set information value of nonlinear constraint. Support information of "LB" and "UB".

**Synopsis**

```
void Set(const char *szInfo, double value)
```

**Arguments**

`szInfo`: name of the information.

`value`: new information value.

### NlConstraint::SetName()

Set name for nonlinear constraint.

**Synopsis**

```
void SetName(const char *szName)
```

**Arguments**

`szName`: the name to set.

### 25.5.65 NlConstrArray Class

To facilitate operations on a group of C++ *NlConstraint Class* objects, the `NlConstrArray` class is provided in the COPT C++ API. The following methods are provided:

#### NlConstrArray::GetNlConstr()

Get idx-th nonlinear constraint object.

**Synopsis**

```
NlConstraint &GetNlConstr(int idx)
```

**Arguments**

`idx`: index of the nonlinear constraint.

**Return**

nonlinear constraint object with index value.

#### NlConstrArray::PushBack()

Add a nonlinear constraint to nonlinear constraint array.

**Synopsis**

```
void PushBack(const NlConstraint &constr)
```

**Arguments**

`constr`: nonlinear constraint object.

#### NlConstrArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: capacity of nonlinear constraint objects.

#### NlConstrArray::Size()

Get the number of nonlinear constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of nonlinear constraint objects.

### 25.5.66 NlConstrBuilder Class

COPT NlConstraint builder object. To help building a nonlinear constraint, given a nonlinear expression, constraint sense and right-hand side value, Cardinal Optimizer provides C++ NlConstrBuilder class, which defines the following methods.

#### NlConstrBuilder::GetNlExpr()

Get nonlinear expression associated with constraint.

**Synopsis**

```
const NlExpr &GetNlExpr()
```

**Return**

nonlinear expression object.

#### NlConstrBuilder::GetRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of nonlinear constraint.

#### NlConstrBuilder::GetSense()

Get sense associated with nonlinear constraint.

**Synopsis**

```
char GetSense()
```

**Return**

nonlinear constraint sense.

#### NlConstrBuilder::Set()

Set detail of a nonlinear constraint to its builder object.

**Synopsis**

```
void Set(
    const NlExpr &expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: nonlinear expression object at one side of nonlinear constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of nonlinear constraint.

#### NlConstrBuilder::SetRange()

Set a range constraint to nonlinear constraint builder.

**Synopsis**

```
void SetRange(const NlExpr &expr, double range)
```

**Arguments**

`expr`: nonlinear expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of nonlinear constraint. Must be greater than 0.

### 25.5.67 NlConstrBuilderArray Class

To facilitate operations on a group of C++ *NlConstrBuilder Class* objects, the `NlConstrBuilderArray` class is provided in the COPT C++ API. The following methods are provided:

#### NlConstrBuilderArray::GetBuilder()

Get idx-th nonlinear constraint builder object.

**Synopsis**

```
NlConstrBuilder &GetBuilder(int idx)
```

**Arguments**

`idx`: index of the nonlinear constraint builder.

**Return**

nonlinear constraint builder object with index idx.

#### NlConstrBuilderArray::PushBack()

Add a nonlinear constraint builder object to nonlinear constraint builder array.

**Synopsis**

```
void PushBack(const NlConstrBuilder &builder)
```

**Arguments**

`builder`: a nonlinear constraint builder object.

#### NlConstrBuilderArray::Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for nonlinear constraint builder object.

#### NlConstrBuilderArray::Size()

Get the number of nonlinear constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of nonlinear constraint builder objects.

### 25.5.68 NL Namespace

The `NL` namespace provides common nonlinear functions for constructing nonlinear expressions. The following methods are provided:

#### NL::Abs()

Calculate absolute value of a nonlinear expression.

**Synopsis**

```
NlExpr Abs(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

#### NL::ACos()

Calculate arccosine of a nonlinear expression.

**Synopsis**

```
NlExpr ACos(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

#### NL::ACosH()

Calculate inverse hyperbolic cosine of a nonlinear expression.

**Synopsis**

```
NlExpr ACosH(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::ASin()

Calculate arcsine of a nonlinear expression.

**Synopsis**

```
NlExpr ASin(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::ASinH()

Calculate inverse hyperbolic sine of a nonlinear expression.

**Synopsis**

```
NlExpr ASinH(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::ATan()

Calculate arctangent of a nonlinear expression.

**Synopsis**

```
NlExpr ATan(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::ATan2()

Calculate two-argument arctangent of a nonlinear expression.

**Synopsis**

```
NlExpr ATan2(const NlExpr &y, const NlExpr &x)
```

**Arguments**

`y`: y coordinate as a nonlinear expression.

`x`: x coordinate as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::ATanH()

Calculate inverse hyperbolic tangent of a nonlinear expression.

**Synopsis**

```
NlExpr ATanH(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Ceil()

Calculate ceiling value of a nonlinear expression.

**Synopsis**

```
NlExpr Ceil(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Cos()

Calculate cosine of a nonlinear expression.

**Synopsis**

```
NlExpr Cos(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::CosH()

Calculate hyperbolic cosine of a nonlinear expression.

**Synopsis**

```
NlExpr CosH(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Exp()

Calculate exponential function of a nonlinear expression.

**Synopsis**

```
NlExpr Exp(const NlExpr &expo)
```

**Arguments**

`expo`: exponent as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Floor()

Calculate floor value of a nonlinear expression.

**Synopsis**

```
NlExpr Floor(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Log10()

Calculate logarithmic function of a nonlinear expression with base 10.

**Synopsis**

```
NlExpr Log10(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Log()

Calculate nature logarithmic function of a nonlinear expression.

**Synopsis**

```
NlExpr Log(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Neg()

Calculate negative value of a nonlinear expression.

**Synopsis**

```
NlExpr Neg(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Pow()

Calculate power function of a nonlinear expression.

**Synopsis**

```
NlExpr Pow(const NlExpr &base, const NlExpr &expo)
```

**Arguments**

`base`: base as a nonlinear expression.

`expo`: exponent as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Sin()

Calculate sine of a nonlinear expression.

**Synopsis**

```
NlExpr Sin(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::SinH()

Calculate hyperbolic sine of a nonlinear expression.

**Synopsis**

```
NlExpr SinH(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Sqrt()

Calculate square root of a nonlinear expression.

**Synopsis**

```
NlExpr Sqrt(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Sum()

Sum of nonlinear expressions.

**Synopsis**

```
NlExpr Sum(const std::vector<NlExpr*> &exprs)
```

**Arguments**

`exprs`: vector of nonlinear expressions.

**Return**

result as a nonlinear expression.

### NL::Sum()

Sum of nonlinear expressions.

**Synopsis**

```
NlExpr Sum(
    const NlExpr &op1,
    const NlExpr &op2,
    const NlExpr &op3)
```

**Arguments**

`op1`: first nonlinear expression.

`op2`: second nonlinear expression.

`op3`: third nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Sum()

Sum of nonlinear expressions.

**Synopsis**

```
NlExpr Sum(
    const NlExpr &op1,
    const NlExpr &op2,
    const NlExpr &op3,
    const NlExpr &op4)
```

**Arguments**

`op1`: first nonlinear expression.

`op2`: second nonlinear expression.

`op3`: third nonlinear expression.

`op4`: fourth nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::Tan()

Calculate tangent of a nonlinear expression.

**Synopsis**

```
NlExpr Tan(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL::TanH()

Calculate hyperbolic tangent of a nonlinear expression.

**Synopsis**

```
NlExpr TanH(const NlExpr &expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### 25.5.69 NdArray

The NdArray class is a built-in multi-dimensional array in COPT. It represents a table of elements of the same type, indexed by a tuple of integers. The following methods are provided:

#### NdArray::NdArray()

Construct an NdArray object with the given shape, filling with the given element.

**Synopsis**

```
NdArray(const Shape<N> &shape, const T &val)
```

**Arguments**

`shape`: shape of NdArray.

`val`: value of element.

#### NdArray::NdArray()

Construct an NdArray object with the given shape, filling with an array of type T.

**Synopsis**

```
NdArray(
    const Shape<N> &shape,
    const T *data,
    size_t sz)
```

**Arguments**

`shape`: shape of NdArray.

`data`: an array of elements.

`sz`: size of elements.

#### NdArray::NdArray()

Construct an NdArray object with the given shape and the filling function.

**Synopsis**

```
NdArray(const Shape<N> &shape)
```

**Arguments**

`shape`: shape of NdArray.

#### NdArray::Clone()

Clone NdArray object.

**Synopsis**

```
NdArray<T, N> Clone()
```

**Return**

new NdArray object.

### NdArray::Diagonal()

Get diagonals of NdArray object.

**Synopsis**

```
NdArray<T, N - 1> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of NdArray.

`axis2`: 2nd axis of NdArray.

**Return**

(N-1)-dimensional diagonals.

### NdArray::Dot()

Dot product with another NdArray object of type double.

**Synopsis**

```
double Dot(const NdArray<double, N> &other)
```

**Arguments**

`other`: another NdArray object of type double.

**Return**

dot product value.

### NdArray::Dot()

Dot product with another NdArray object of type int.

**Synopsis**

```
T Dot(const NdArray<int, N> &other)
```

**Arguments**

`other`: another NdArray object of type int.

**Return**

dot product value.

### NdArray::Expand()

Expand shape of NdArray object.

**Synopsis**

```
NdArray<T, N + 1> Expand(int axis)
```

**Arguments**

`axis`: axis of NdArray.

**Return**

NdArray object in (N+1)-dimensions.

### NdArray::Fill()

Fill NdArray object with given value.

**Synopsis**

```
void Fill(const T &val)
```

**Arguments**

`val`: new value.

### NdArray::Flatten()

Flatten an NdArray object to a 1-dimensional shape.

**Synopsis**

```
NdArray<T, 1> Flatten()
```

**Return**

An NdArray object collapsed into one dimension.

### NdArray::GetDim()

Get i-th dimension of NdArray object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### NdArray::GetND()

Get number of dimensions of NdArray object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### NdArray::GetShape()

Get shape of NdArray object.

**Synopsis**

```
const Shape<N> &GetShape()
```

**Return**

shape object.

### NdArray::GetSize()

Get size of NdArray object.

**Synopsis**

```
size_t GetSize()
```

**Return**

number of elements.

### NdArray::Item()

Get element of given index from NdArray object.

**Synopsis**

```
T &Item(size_t idx)
```

**Arguments**

`idx`: index of element.

**Return**

value of element.

### NdArray::Item()

Get sub-array of NdArray object, given View object.

**Synopsis**

```
NdArray<T, N> Item(const View &view)
```

**Arguments**

`view`: View object.

**Return**

sub NdArray without copying underlying data.

#### NdArray::operator[]()

Get element of given index from NdArray object.

**Synopsis**

```
T &operator[](size_t idx)
```

**Arguments**

`idx`: index of element.

**Return**

value of element.

#### NdArray::operator[]()

Get constraints of given view from NdArray object.

**Synopsis**

```
NdArray<T, N> operator[](const View &view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

new NdArray object.

#### NdArray::Pick()

Given a list of indexes, get elements from NdArray object.

**Synopsis**

```
NdArray<T, 1> Pick(const NdArray<int, 1> &indexes)
```

**Arguments**

`indexes`: indexes of elements.

**Return**

one-dimensional array of desired elements.

#### NdArray::Pick()

Given a list of indexes, get elements from NdArray object.

**Synopsis**

```
NdArray<T, 1> Pick(const NdArray<int, 2> &idxrows)
```

**Arguments**

`idxrows`: indexes in format of 2-dimensional array, where each row is position of element.

**Return**

one-dimensional array of desired elements.

### NdArray::Prod()

Product of all elements in NdArray object.

**Synopsis**

```
T Prod()
```

**Return**

product value.

### NdArray::Prod()

Product of elements at given axis of NdArray object.

**Synopsis**

```
NdArray<T, N - 1> Prod(int axis)
```

**Arguments**

`axis`: axis of NdArray.

**Return**

(N-1)-dimensional NdArray object.

### NdArray::Repeat()

Repeat each element of an array along given axis.

**Synopsis**

```
NdArray<T, N> Repeat(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of NdArray.

**Return**

new NdArray object.

### NdArray::RepeatBlock()

Repeat an array a number of times along given axis.

**Synopsis**

```
NdArray<T, N> RepeatBlock(size_t repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of NdArray.

**Return**

new NdArray object.

#### NdArray::Represent()

String representation of NdArray object.

**Synopsis**

```
std::string Represent(size_t maxlen)
```

**Arguments**

`maxlen`: max length of representation string.

**Return**

representation string object.

#### NdArray::Reshape()

Reshape NdArray object to new shape.

**Synopsis**

```
template <int M> NdArray<T, M> Reshape(const Shape<M> &shape)
```

**Arguments**

`shape`: new shape of M-dimensions.

**Return**

M-dimensional NdArray object.

#### NdArray::SetItem()

Set element of given index to NdArray object.

**Synopsis**

```
void SetItem(size_t idx, const T &val)
```

**Arguments**

`idx`: index of element.

`val`: value of element.

#### NdArray::Squeeze()

Remove axis of length 1 from shape of NdArray object.

**Synopsis**

```
NdArray<T, N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: axis of NdArray, where the length is 1.

**Return**

(N-1)-dimensional NdArray object.

#### NdArray::Sum()

Sum of all elements in NdArray object.

**Synopsis**

```
T Sum()
```

**Return**

sum value.

#### NdArray::Sum()

Sum of elements at given axis of NdArray object.

**Synopsis**

```
NdArray<T, N - 1> Sum(int axis)
```

**Arguments**

`axis`: axis of NdArray.

**Return**

(N-1)-dimensional NdArray object.

#### NdArray::Transpose()

Perform matrix transpose of NdArray object.

**Synopsis**

```
NdArray<T, N> Transpose()
```

**Return**

transposed NdArray object.

### 25.5.70 Shape

The Shape class encapsulates a tuple of integers, indicating the size of array along each dimension. It refers to dimensions of built-in *NdArray* in COPT. The following methods are provided:

#### Shape::Expand()

Expand shape of Shape object.

**Synopsis**

```
Shape<N + 1> Expand(int axis)
```

**Arguments**

`axis`: given axis.

**Return**

Shape object in (N+1)-dimensions.

### Shape::GetDim()

Get i-th dimension in Shape object.

**Synopsis**

```
size_t GetDim(int i)
```

**Arguments**

`i`: index of dimensions.

**Return**

the i-th dimension.

### Shape::GetND()

Get number of dimensions in Shape object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### Shape::GetSize()

Get size of Shape object.

**Synopsis**

```
size_t GetSize()
```

**Return**

size of shape.

### Shape::GetStart()

Get the i-th start position in Shape object.

**Synopsis**

```
int64_t GetStart(int i)
```

**Arguments**

`i`: index of dimensions.

**Return**

start position in i-th dimension.

### Shape::GetStride()

Get i-th stride in Shape object.

**Synopsis**

```
int64_t GetStride(int i)
```

**Arguments**

`i`: index of dimensions.

**Return**

stride in i-th dimension.

### Shape::operator!=()

Use operator '!=' to compare with other Shape object.

**Synopsis**

```
bool operator!=(const Shape<N> &other)
```

**Arguments**

`other`: other Shape object.

**Return**

True if shape is not the same.

### Shape::operator==()

Use operator '==' to compare with other Shape object.

**Synopsis**

```
bool operator==(const Shape<N> &other)
```

**Arguments**

`other`: other Shape object.

**Return**

True if shape is the same.

### Shape::Rebuild()

Rebuild Shape object, that is, keep dimensions while reset strides and starts.

**Synopsis**

```
Shape<N> Rebuild()
```

**Return**

new Shape object.

#### Shape::Represent()

String representation of Shape object.

**Synopsis**

```
std::string Represent(int type)
```

**Arguments**

`type`: 0: dimensions; 1: strides; 2: starts.

**Return**

string object.

#### Shape::Squeeze()

Remove axis of length 1 from Shape object.

**Synopsis**

```
Shape<N - 1> Squeeze(int axis)
```

**Arguments**

`axis`: given axis, where the length is 1.

**Return**

Shape object in (N-1)-dimensions.

### 25.5.71 View

The View class is used to perform slicing operations on multi-dimensional arrays. The following methods are provided:

#### View::View()

Create an empty View object. Call make_view() for your convenience.

**Synopsis**

```
View()
```

#### View::AddFull()

Create full view object at current dimension.

**Synopsis**

```
View &AddFull()
```

**Return**

View object.

### View::AddScalar()

Create view object of given index at current dimension.

**Synopsis**

```
View &AddScalar(int64_t n)
```

**Arguments**

`n`: given index.

**Return**

View object.

### View::AddSlice()

Create view object of slice at current dimension.

**Synopsis**

```
View &AddSlice(int64_t start)
```

**Arguments**

`start`: start index, inclusive.

**Return**

View object.

### View::AddSlice()

Create view object of slice at current dimension.

**Synopsis**

```
View &AddSlice(int64_t start, int64_t stop)
```

**Arguments**

`start`: start index, inclusive.

`stop`: stop index, exclusive.

**Return**

View object.

### View::AddSlice()

Create view object of slice at current dimension.

**Synopsis**

```
View &AddSlice(
    int64_t start,
    int64_t stop,
    int64_t step)
```

**Arguments**

`start`: start index, inclusive.

`stop`: stop index, exclusive.

`step`: step size between start and stop index. It can be negative.

**Return**

View object.

### View::operator()()

Create full view object at current dimension.

**Synopsis**

```
View &operator()()
```

**Return**

View object.

### View::operator()()

Create view object of slice at current dimension.

**Synopsis**

```
View &operator()(int64_t start, int64_t stop)
```

**Arguments**

`start`: start index, inclusive.

`stop`: stop index, exclusive.

**Return**

View object.

### View::operator()()

Create view object of slice at current dimension.

**Synopsis**

```
View &operator()(
    int64_t start,
    int64_t stop,
    int64_t step)
```

**Arguments**

`start`: start index, inclusive.

`stop`: stop index, exclusive.

`step`: step size between start and stop index. It can be negative.

**Return**

View object.

#### View::operator()()

Create view object of given index at current dimension.

**Synopsis**

```
View &operator()(int64_t n)
```

**Arguments**

`n`: given index.

**Return**

View object.

### 25.5.72 CallbackBase

COPT Callback abstract base object. Users must implement its virtual method `virtual void CallbackBase::callback()` to instantiate an instance, which pass to `Model::SetCallback(ICallback* pcb, int cbctx)` as the first parameter. Subclass of CallbackBase inherits the following methods:

#### CallbackBase::AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(
    const Expr &lhs,
    char sense,
    double rhs)
```

**Arguments**

`lhs`: expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

#### CallbackBase::AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(
    const Expr &lhs,
    char sense,
    const Expr &rhs)
```

**Arguments**

`lhs`: left hand side expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy constraint.

### CallbackBase::AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(const ConstrBuilder &builder)
```

**Arguments**

`builder`: builder for lazy constraint.

### CallbackBase::AddLazyConstrs()

Add lazy constraints to model.

**Synopsis**

```
void AddLazyConstrs(const ConstrBuilderArray &builders)
```

**Arguments**

`builders`: array of builders for lazy constraints.

### CallbackBase::AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(
    const Expr &lhs,
    char sense,
    double rhs)
```

**Arguments**

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

### CallbackBase::AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(
    const Expr &lhs,
    char sense,
    const Expr &rhs)
```

**Arguments**

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

### CallbackBase::AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(const ConstrBuilder &builder)
```

**Arguments**

`builder`: builder for user cut.

### CallbackBase::AddUserCuts()

Add user cuts to model.

**Synopsis**

```
void AddUserCuts(const ConstrBuilderArray &builders)
```

**Arguments**

`builders`: array of builders for user cuts.

### CallbackBase::GetDblInfo()

Get double value of given information name in callback.

**Synopsis**

```
double GetDblInfo(const char *cbinfo)
```

**Arguments**

`cbinfo`: name of callback info.

**Return**

value of desired information.

### CallbackBase::GetIncumbent()

Get best feasible solution of given variable in callback.

**Synopsis**

```
double GetIncumbent(Var &var)
```

**Arguments**

`var`: given variable.

**Return**

best feasible solution of given variable.

#### CallbackBase::GetIncumbent()

Get best feasible solution of variables in callback.

**Synopsis**

```
int GetIncumbent(VarArray &vars, double *pOut)
```

**Arguments**

`vars`: an array of variables.

`pOut`: best feasible solution of desired variables.

**Return**

the number of valid variables. If failed, return -1.

#### CallbackBase::GetIncumbent()

Get best feasible solution of all variables in callback.

**Synopsis**

```
int GetIncumbent(double *pOut, int len)
```

**Arguments**

`pOut`: optional, output best feasible solution of all variables.

`len`: the length of output array. The solution is written up to number of len.

**Return**

number of all variables. Return -1 if error occurs.

#### CallbackBase::GetIntInfo()

Get integer value of given information name in callback.

**Synopsis**

```
int GetIntInfo(const char *cbinfo)
```

**Arguments**

`cbinfo`: name of callback info.

**Return**

value of desired information.

#### CallbackBase::GetRelaxSol()

Get LP-relaxation solution of given variable in callback.

**Synopsis**

```
double GetRelaxSol(Var &var)
```

**Arguments**

`var`: given variable.

**Return**

LP-relaxation solution of given variable.

### CallbackBase::GetRelaxSol()

Get LP-relaxation solution of variables in callback.

**Synopsis**

```
int GetRelaxSol(VarArray &vars, double *pOut)
```

**Arguments**

`vars`: an array of variables.

`pOut`: LP-relaxation solution of desired variables.

**Return**

the number of valid variables. If failed, return -1.

### CallbackBase::GetRelaxSol()

Get LP-relaxation solution of all variables in callback.

**Synopsis**

```
int GetRelaxSol(double *pOut, int len)
```

**Arguments**

`pOut`: optional, output LP-relaxation solution of all variables.

`len`: the length of output array. The solution is written up to number of len.

**Return**

number of all variables. Return -1 if error occurs.

### CallbackBase::GetSolution()

Get solution of given variable in callback.

**Synopsis**

```
double GetSolution(Var &var)
```

**Arguments**

`var`: given variable.

**Return**

solution of desired variable.

### CallbackBase::GetSolution()

Get solution of variables in callback.

**Synopsis**

```
int GetSolution(VarArray &vars, double *pOut)
```

**Arguments**

`vars`: an array of variables.

`pOut`: solution of desired variables.

**Return**

the number of valid variables. If failed, return -1.

#### CallbackBase::GetSolution()

Get solution of all variables in callback.

**Synopsis**

```
int GetSolution(double *pOut, int len)
```

**Arguments**

`pOut`: optional, output solution of all variables.

`len`: the length of output array. The solution is written up to number of len.

**Return**

number of all variables. Return -1 if error occurs.

#### CallbackBase::Interrupt()

Interrupt solving problems in callback

**Synopsis**

```
void Interrupt()
```

#### CallbackBase::LoadSolution()

Load customized solution to model.

**Synopsis**

```
double LoadSolution()
```

**Return**

objective value of given solution.

#### CallbackBase::SetSolution()

Set solution of a given variable in callback.

**Synopsis**

```
void SetSolution(Var &var, double val)
```

**Arguments**

`var`: a variable object.

`val`: double value.

#### CallbackBase::SetSolution()

Set solution of variables in callback.

**Synopsis**

```
void SetSolution(
    VarArray &vars,
    const double *vals,
    int len)
```

**Arguments**

`vars`: an array of variable objects.

`vals`: an array of double values.

`len`: length of array of double values.

#### CallbackBase::Where()

Get context in callback.

**Synopsis**

```
int Where()
```

**Return**

integer value of context.

### 25.5.73 NlpCallbackBase

The NlpCallbackBase class provides an interface for defining evaluation callbacks for nonlinear optimization models in COPT.

By inheriting this class and implementing the corresponding member methods, users provide objective and constraint value evaluations, together with their first-order and second-order derivative information to COPT during nonlinear optimization.

#### NlpCallbackBase::EvalObj(INdArray<double, 1>* xdata, INdArray<double, 1>* outdata)

Evaluate objective of nonlinear model.

**Synopsis**

```
void EvalObj(
    INdArray<double, 1> *xdata,
    INdArray<double, 1> *outdata)
```

**Arguments**

`xdata`: pointer to NdArray of variable values.

`outdata`: pointer to output NdArray of objective value.

#### NlpCallbackBase::EvalGrad(INdArray<double, 1>* xdata, INdArray<double, 1>* outdata)

Evaluate objective gradient of nonlinear model.

**Synopsis**

```
void EvalGrad(
    INdArray<double, 1> *xdata,
    INdArray<double, 1> *outdata)
```

**Arguments**

`xdata`: pointer to NdArray of variable values.

`outdata`: pointer to output NdArray for objective gradient.

#### NlpCallbackBase::EvalCon(INdArray<double, 1>* xdata, INdArray<double, 1>* outdata)

Evaluate constraint values of nonlinear model.

**Synopsis**

```
void EvalCon(
    INdArray<double, 1> *xdata,
    INdArray<double, 1> *outdata)
```

**Arguments**

`xdata`: pointer to NdArray of variable values.

`outdata`: pointer to output NdArray for constraint values.

#### NlpCallbackBase::EvalJac(INdArray<double, 1>* xdata, INdArray<double, 1>* outdata)

Evaluate Jacobian values of the nonlinear constraint functions.

**Synopsis**

```
void EvalJac(
    INdArray<double, 1> *xdata,
    INdArray<double, 1> *outdata)
```

**Arguments**

`xdata`: pointer to NdArray of variable values.

`outdata`: pointer to output NdArray for nonzero Jacobian entries.

#### NlpCallbackBase::EvalHess(INdArray<double, 1>* xdata, double sigma, INdArray<double, 1>* lambdata, INdArray<double, 1>* outdata)

Evaluate Hessian values of the Lagrangian function of the nonlinear model.

**Synopsis**

```
void EvalHess(
    INdArray<double, 1> *xdata,
    double sigma,
    INdArray<double, 1> *lambdata,
    INdArray<double, 1> *outdata)
```

**Arguments**

`xdata`: pointer to NdArray of variable values.

`sigma`: weight on the objective in the Lagrangian function.

`lambdata`: pointer to NdArray of Lagrange multiplier associated with the constraint functions.

`outdata`: pointer to output NdArray for nonzero Hessian entries.

### 25.5.74 ProbBuffer

Buffer object for COPT problem. ProbBuffer object holds the (MPS) problem in string format.

#### ProbBuffer::ProbBuffer()

Constructor of ProbBuffer object.

**Synopsis**

```
ProbBuffer(int sz)
```

**Arguments**

`sz`: initial size of the problem buffer.

#### ProbBuffer::GetData()

Get string of problem in problem buffer.

**Synopsis**

```
char *GetData()
```

**Return**

string of problem in problem buffer.

#### ProbBuffer::Resize()

Resize buffer to given size, and zero-ended

**Synopsis**

```
void Resize(int sz)
```

**Arguments**

`sz`: new buffer size.

#### ProbBuffer::Size()

Get the size of problem buffer.

**Synopsis**

```
int Size()
```

**Return**

size of problem buffer.

# Chapter 26

# C# API Reference

The **Cardinal Optimizer** provides C# API library. This chapter documents all COPT C# constants and API functions for C# applications.

## 26.1 Constants

There are four types of constants defined in **Cardinal Optimizer**. They are general constants, information constants, attributes and parameters.

### 26.1.1 General Constants

For the contents of C# general constants, see *General Constants*.

General constants are defined in `Consts` class. User may refer to general constants with namespace, that is, `Copt.Consts.XXXX`.

### 26.1.2 Attributes

For the contents of C# attribute constants, see *Attributes*.

All COPT C# attributes are defined in `DblAttr` and `IntAttr` classes. User may refer to double attributes by `Copt.DblAttr.XXXX`, and integer attributes by `Copt.IntAttr.XXXX`.

In the C# API, user can get the attribute value by specifying the attribute name. The two functions of obtaining attribute values are as follows, please refer to *C# API: Model Class* for details.

- `Model.GetIntAttr()`: Get value of a COPT integer attribute.
- `Model.GetDblAttr()`: Get value of a COPT double attribute.

### 26.1.3 Information

For the content of C# information constants, see *Information*.

In the C# API, information constants are defined in the `DblInfo` class. Users can access information constants through the prefix `Copt` in the namespace (usually can be omitted) `Copt.DblInfo.`

For instance, `Copt.DblInfo.Obj` is the coefficients of variables in the objective function.

### 26.1.4 Callback Information

For the content of C# API callback information class constants, see *Callback Information*.

In the C# API, callback-related information constants are defined in the `CbInfo` class. Users can access information constants through the prefix `Copt` in the namespace (usually can be omitted) `Copt.CbInfo.`

For instance, `Copt.CbInfo.BestObj` is the current best objective.

### 26.1.5 Parameters

For the contents of C# parameters constants, see *Parameters*.

All COPT C# parameters are defined in DblParam and IntParam classes. User may refer to double parameters by `Copt.DblParam.XXXX`, and integer parameters by `Copt.IntParam.XXXX`.

In the C# API, user can get and set the parameter value by specifying the parameter name. The provided functions are as follows, please refer to *C# API: Model Class* for details.

- Get detailed information of the specified parameter (current value/max/min): `Model.GetParamInfo()`
- Get the current value of the specified integer/double parameter: `Model.GetIntParam()` / `Model.GetDblParam()`
- Set the specified integer/double parameter value: `Model.SetIntParam()` / `Model.SetDblParam()`

## 26.2 C# Modeling Classes

This chapter documents COPT C# interface. Users may refer to C# classes described below for details of how to construct and solve C# models.

### 26.2.1 Envr

Essentially, any C# application using Cardinal Optimizer should start with a COPT environment. COPT models are always associated with a COPT environment. User must create an environment object before populating models. User generally only need a single environment object in program.

#### Envr.Envr()

Constructor of COPT Envr object.

**Synopsis**

```
Envr()
```

#### Envr.Envr()

Constructor of COPT Envr object, given a license folder.

**Synopsis**

```
Envr(string licDir)
```

**Arguments**

`licDir`: directory having local license or client config file.

### Envr.Envr()

Constructor of COPT Envr object, given an Envr config object.

**Synopsis**

```
Envr(EnvrConfig config)
```

**Arguments**

`config`: Envr config object holding settings for remote connection.

### Envr.BindNumaCpu()

Bind the CPUs for the current process to a NUMA node.

**Synopsis**

```
void BindNumaCpu(int numaNode)
```

**Arguments**

`numaNode`: ID of a NUMA node.

### Envr.BindNumaMem()

Bind memory for the current process to a NUMA node (Linux only).

**Synopsis**

```
void BindNumaMem(int numaNode)
```

**Arguments**

`numaNode`: the ID of a NUMA node.

### Envr.Close()

Close remote connection and token becomes invalid for all problems in current environment. To release the environment, call Dispose() for that.

**Synopsis**

```
void Close()
```

### Envr.CreateModel()

Create a model object.

**Synopsis**

```
Model CreateModel(string name)
```

**Arguments**

`name`: customized model name.

**Return**

a model object.

#### Envr.Dispose()

Release this environment reference on the calling thread. Prefer disposing explicitly (via a using statmement) on the same thread, instead of relying on the GC finalizer thread. Note the underlying native environment is reference-counted and is destroyed when the last referring to it is released.

**Synopsis**

```
void Dispose()
```

#### Envr.GetCpuAffinity()

Get CPU affinity for the current process, which is saved in an integer array.

**Synopsis**

```
int[] GetCpuAffinity()
```

**Return**

an integer array of CPU IDs.

#### Envr.GetNumaNodeCount()

Get count of NUMA nodes.

**Synopsis**

```
int GetNumaNodeCount()
```

**Return**

count of NUMA nodes.

#### Envr.SetCpuAffinity()

Set CPU affinity with given mask string.

**Synopsis**

```
void SetCpuAffinity(string hexMask)
```

**Arguments**

`hexMask`: CPU mask string of hexadecimal characters.

### 26.2.2 EnvrConfig

If user connects to COPT remote services, such as floating token server or compute cluster, it is necessary to add config settings with EnvrConfig object.

#### EnvrConfig.EnvrConfig()

Constructor of envr config object.

**Synopsis**

```
EnvrConfig()
```

#### EnvrConfig.Set()

Set config settings in terms of name-value pair.

**Synopsis**

```
void Set(string name, string value)
```

**Arguments**

`name`: keyword of a config setting.

`value`: value of a config setting.

### 26.2.3 Model

In general, a COPT model consists of a set of variables, a (linear) objective function on these variables, a set of constraints on these variables, etc. COPT model class encapsulates all required methods for constructing a COPT model.

#### Model.Model()

Constructor of model.

**Synopsis**

```
Model(Envr env, string name)
```

**Arguments**

`env`: associated environment object.

`name`: string of model name.

#### Model.AddAffineCone()

Add an affine cone constraint to model.

**Synopsis**

```
AffineCone AddAffineCone(AffineConeBuilder builder, string name)
```

**Arguments**

`builder`: builder for new affine cone constraint.

`name`: optional, name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model.AddAffineCone()

Add an affine cone constraint to a model.

**Synopsis**

```
AffineCone AddAffineCone(
    MLinExpr expr,
    int type,
    string name)
```

**Arguments**

`expr`: 1-dimensional array of linear expressions.

`type`: type of an affine cone.

`name`: name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model.AddAffineCone()

Add an affine cone constraint to a model.

**Synopsis**

```
AffineCone AddAffineCone(
    MVar vars,
    int type,
    string name)
```

**Arguments**

`vars`: 1-dimensional array of variables.

`type`: type of an affine cone.

`name`: name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model.AddAffineCone()

Add an affine cone constraint to model.

**Synopsis**

```
AffineCone AddAffineCone(
    Expr[] exprs,
    int type,
    string name)
```

**Arguments**

`exprs`: linear expressions that participate in the affine cone constraint.

`type`: type of an affine cone constraint.

`name`: name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model.AddAffineCone()

Add an affine cone constraint to a model.

**Synopsis**

```
AffineCone AddAffineCone(
    Var[] vars,
    int type,
    string name)
```

**Arguments**

`vars`: variables that participate in the affine cone constraint.

`type`: type of an affine cone.

`name`: name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model.AddAffineCone()

Add an affine cone constraint to a model.

**Synopsis**

```
AffineCone AddAffineCone(
    MPsdExpr expr,
    int type,
    string name)
```

**Arguments**

`expr`: 1-dimensional array of PSD expressions.

`type`: type of an affine cone.

`name`: name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model.AddAffineCone()

Add an affine cone constraint to model.

**Synopsis**

```
AffineCone AddAffineCone(
    PsdExpr[] exprs,
    int type,
    string name)
```

**Arguments**

`exprs`: PSD expressions that participate in the affine cone constraint.

`type`: type of an affine cone constraint.

`name`: name of new affine cone constraint.

**Return**

new affine cone constraint object.

### Model.AddAffineCones()

Add a list of affine cone constraints to a model.

**Synopsis**

```
AffineConeArray AddAffineCones(
    MLinExpr exprs,
    int type,
    string prefix)
```

**Arguments**

`exprs`: 2-dimensional array of linear expressions.

`type`: type of an affine cone.

`prefix`: name prefix for new affine cone constraints.

**Return**

array of new affine cone constraint objects.

### Model.AddAffineCones()

Add a list of affine cone constraints to a model.

**Synopsis**

```
AffineConeArray AddAffineCones(
    MVar vars,
    int type,
    string prefix)
```

**Arguments**

`vars`: 2-dimensional array of variables.

`type`: type of affine cones.

`prefix`: name prefix for new affine cone constraints.

**Return**

array of new affine cone constraint objects.

### Model.AddAffineCones()

Add a list of affine cone constraints to a model.

**Synopsis**

```
AffineConeArray AddAffineCones(
    MPsdExpr exprs,
    int type,
    string prefix)
```

**Arguments**

`exprs`: 2-dimensional array of PSD expressions.

`type`: type of an affine cone.

`prefix`: name prefix for new affine cone constraints.

**Return**

array of new affine cone constraint objects.

### Model.AddCone()

Add a cone constraint to model.

**Synopsis**

```
Cone AddCone(
    int dim,
    int type,
    char[] pvtype,
    string prefix)
```

**Arguments**

`dim`: dimension of the cone constraint.

`type`: type of the cone constraint.

`pvtype`: type of variables in the cone.

`prefix`: optional, name prefix of variables in the cone, default value is "ConeV".

**Return**

new cone constraint object.

### Model.AddCone()

Add a cone constraint to model.

**Synopsis**

```
Cone AddCone(ConeBuilder builder)
```

**Arguments**

`builder`: builder for new cone constraint.

**Return**

new cone constraint object.

### Model.AddCone()

Add a cone constraint to model.

**Synopsis**

```
Cone AddCone(Var[] vars, int type)
```

**Arguments**

`vars`: variables that participate in the cone constraint.

`type`: type of the cone constraint.

**Return**

new cone constraint object.

### Model.AddCone()

Add a cone constraint to model.

**Synopsis**

```
Cone AddCone(VarArray vars, int type)
```

**Arguments**

`vars`: variables that participate in the cone constraint.

`type`: type of a cone constraint.

**Return**

new cone constraint object.

### Model.AddConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint AddConstr(
    Expr expr,
    char sense,
    double rhs,
    string name)
```

**Arguments**

`expr`: expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`name`: optional, name of new constraint.

**Return**

new constraint object.

### Model.AddConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint AddConstr(
    Expr expr,
    char sense,
    Var var,
    string name)
```

**Arguments**

`expr`: expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`var`: variable as right hand side for the new constraint.

`name`: optional, name of new constraint.

**Return**

new constraint object.

### Model.AddConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint AddConstr(
    Expr lhs,
    char sense,
    Expr rhs,
    string name)
```

**Arguments**

`lhs`: left hand side expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side expression for the new constraint.

`name`: optional, name of new constraint.

**Return**

new constraint object.

### Model.AddConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint AddConstr(
    Expr expr,
    double lb,
    double ub,
    string name)
```

**Arguments**

`expr`: expression for the new constraint.

`lb`: lower bound for the new constraint.

`ub`: upper bound for the new constraint

`name`: optional, name of new constraint.

**Return**

new constraint object.

### Model.AddConstr()

Add a linear constraint to a model.

**Synopsis**

```
Constraint AddConstr(ConstrBuilder builder, string name)
```

**Arguments**

`builder`: builder for the new constraint.

`name`: optional, name of new constraint.

**Return**

new constraint object.

### Model.AddConstrs()

Add linear constraints to model.

**Synopsis**

```
ConstrArray AddConstrs(
    int count,
    char[] senses,
    double[] rhss,
    string prefix)
```

**Arguments**

`count`: number of constraints added to model.

`senses`: sense array for new linear constraints, other than range sense.

`rhss`: right hand side values for new variables.

`prefix`: optional, name prefix for new constraints, default value is 'R'.

**Return**

array of new constraint objects.

### Model.AddConstrs()

Add linear constraints to a model.

**Synopsis**

```
ConstrArray AddConstrs(
    int count,
    double[] lbs,
    double[] ubs,
    string prefix)
```

**Arguments**

`count`: number of constraints added to the model.

`lbs`: lower bounds of new constraints.

`ubs`: upper bounds of new constraints.

`prefix`: optional, name prefix for new constraints, default value is 'R'.

**Return**

array of new constraint objects.

### Model.AddConstrs()

Add linear constraints to a model.

**Synopsis**

```
ConstrArray AddConstrs(ConstrBuilderArray builders, string prefix)
```

**Arguments**

`builders`: builders for new constraints.

`prefix`: optional, name prefix for new constraints, default value is 'R'.

**Return**

array of new constraint objects.

### Model.AddDenseMat()

Add a dense symmetric matrix to a model.

**Synopsis**

```
SymMatrix AddDenseMat(int dim, double[] vals)
```

**Arguments**

`dim`: dimension of the dense symmetric matrix.

`vals`: array of non-zero elements, filled in column-wise up to len or max length of symmetric matrix.

**Return**

new symmetric matrix object.

### Model.AddDenseMat()

Add a dense symmetric matrix to a model.

**Synopsis**

```
SymMatrix AddDenseMat(int dim, double val)
```

**Arguments**

`dim`: dimension of dense symmetric matrix.

`val`: value to fill dense symmetric matrix.

**Return**

new symmetric matrix object.

### Model.AddDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix AddDiagMat(int dim, double val)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

**Return**

new diagonal matrix object.

### Model.AddDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix AddDiagMat(int dim, double[] vals)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`vals`: array of values of diagonal elements.

**Return**

new diagonal matrix object.

### Model.AddDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix AddDiagMat(
    int dim,
    double val,
    int offset)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

`offset`: shift distance against diagonal line.

**Return**

new diagonal matrix object.

### Model.AddDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix AddDiagMat(
    int dim,
    double[] vals,
    int offset)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`vals`: array of values of diagonal elements.

`offset`: shift distance against diagonal line.

**Return**

new diagonal matrix object.

### Model.AddExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone AddExpCone(
    int type,
    char[] pvtype,
    string prefix)
```

**Arguments**

`type`: type of the exponential cone constraint.

`pvtype`: type of variables in the exponential cone.

`prefix`: optional, name prefix of variables in the exponential cone, default value is "ExpConeV".

**Return**

new exponential cone constraint object.

### Model.AddExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone AddExpCone(ExpConeBuilder builder)
```

**Arguments**

`builder`: builder for new exponential cone constraint.

**Return**

new exponential cone constraint object.

### Model.AddExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone AddExpCone(Var[] vars, int type)
```

**Arguments**

`vars`: variables that participate in the exponential cone constraint.

`type`: type of the exponential cone constraint.

**Return**

new exponential cone constraint object.

### Model.AddExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone AddExpCone(VarArray vars, int type)
```

**Arguments**

`vars`: variables that participate in the exponential cone constraint.

`type`: type of an exponential cone constraint.

**Return**

new exponential cone constraint object.

### Model.AddEyeMat()

Add an identity matrix to a model.

**Synopsis**

```
SymMatrix AddEyeMat(int dim)
```

**Arguments**

`dim`: dimension of identity matrix.

**Return**

new identity matrix object.

### Model.AddGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr AddGenConstrIndicator(GenConstrBuilder builder, string
name)
```

**Arguments**

`builder`: builder for the general constraint.

`name`: optional, name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model.AddGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr AddGenConstrIndicator(
    Var binvar,
    int binval,
    ConstrBuilder builder,
    int type,
    string name)
```

**Arguments**

`binvar`: binary indicator variable.

`binval`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`builder`: builder for linear constraint.

`type`: type of general constraint.

`name`: optional, name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model.AddGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr AddGenConstrIndicator(
    Var binvar,
    int binval,
    Expr expr,
    char sense,
    double rhs,
    int type,
    string name)
```

**Arguments**

`binvar`: binary indicator variable.

`binval`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`expr`: expression for new linear constraint.

`sense`: sense for new linear constraint.

`rhs`: right hand side value for new linear constraint.

`type`: type of general constraint.

`name`: optional, name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model.AddGenConstrIndicators()

Add general constraints to a model.

**Synopsis**

```
GenConstrArray AddGenConstrIndicators(GenConstrBuilderArray
builders, string prefix)
```

**Arguments**

`builders`: builders for new general constraints.

`prefix`: optional, name prefix for new general constraints.

**Return**

array of new general constraint objects.

### Model.AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(
    Expr lhs,
    char sense,
    double rhs,
    string name)
```

**Arguments**

`lhs`: expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

`name`: optional, name of lazy constraint.

### Model.AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(
    Expr lhs,
    char sense,
    Expr rhs,
    string name)
```

**Arguments**

`lhs`: left hand side expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy constraint.

`name`: optional, name of lazy constraint.

### Model.AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(ConstrBuilder builder, string name)
```

**Arguments**

`builder`: builder for lazy constraint.

`name`: optional, name of lazy constraint.

### Model.AddLazyConstrs()

Add lazy constraints to model.

**Synopsis**

```
void AddLazyConstrs(ConstrBuilderArray builders, string prefix)
```

**Arguments**

`builders`: array of builders for lazy constraints.

`prefix`: name prefix of new lazy constraints.

### Model.AddLmiConstr()

Add an LMI constraint to model.

**Synopsis**

```
LmiConstraint AddLmiConstr(LmiExpr expr, string name)
```

**Arguments**

`expr`: LMI expression for new LMI constraint.

`name`: optional, name of new LMI constraint.

**Return**

new LMI constraint object.

### Model.AddMConstr()

Add a MConstr object in N-dimensions to model.

**Synopsis**

```
MConstr AddMConstr(MConstrBuilder builder, string prefix)
```

**Arguments**

`builder`: builder for MConstr object.

`prefix`: name prefix for constraints in MConstr object.

**Return**

new MConstr object.

### Model.AddMConstr()

Add a N-dimensional MConstr object to model.

**Synopsis**

```
MConstr AddMConstr(
    MLinExpr exprs,
    char sense,
    double rhs,
    string prefix)
```

**Arguments**

`exprs`: N-dimensional MLinExpr object.

`sense`: sense for new linear constraints.

`rhs`: double value at right side of the new linear constraints.

`prefix`: name prefix for constraints in MConstr object.

**Return**

new MConstr object.

### Model.AddMPsdConstr()

Add a N-dimensional MPsdConstr object to model.

**Synopsis**

```
MPsdConstr AddMPsdConstr(MPsdConstrBuilder builder, string prefix)
```

**Arguments**

`builder`: builder for MPsdConstr object.

`prefix`: name prefix of PSD constraints in MPsdConstr object.

**Return**

new MPsdConstr object.

### Model.AddMPsdConstr()

Add a N-dimensional MPsdConstr object to model.

**Synopsis**

```
MPsdConstr AddMPsdConstr(
    MPsdExpr exprs,
    char sense,
    double rhs,
    string prefix)
```

**Arguments**

`exprs`: N-dimensional MPsdExpr object.

`sense`: sense for new PSD constraints.

`rhs`: double value at right side of the new PSD constraints.

`prefix`: name prefix of PSD constraints in MPsdConstr object.

**Return**

new MPsdConstr object.

### Model.AddMQConstr()

Add a N-dimensional MQConstr object to model.

**Synopsis**

```
MQConstr AddMQConstr(MQConstrBuilder builder, string prefix)
```

**Arguments**

`builder`: builder for MQConstr object.

`prefix`: name prefix of quadratic constraints in MQConstr object.

**Return**

new MQConstr object.

### Model.AddMQConstr()

Add a N-dimensional MQConstr object to model.

**Synopsis**

```
MQConstr AddMQConstr(
    MQuadExpr exprs,
    char sense,
    double rhs,
    string prefix)
```

**Arguments**

`exprs`: N-dimensional MQuadExpr object.

`sense`: sense for new quadratic constraints.

`rhs`: double value at right side of the new quadratic constraints.

`prefix`: name prefix of quadratic constraints in MQConstr object.

**Return**

new MQConstr object.

### Model.AddMVar()

Add a MVar object in N-dimensions to model.

**Synopsis**

```
MVar AddMVar(
    Shape shp,
    char vtype,
    string prefix)
```

**Arguments**

`shp`: shape of MVar object.

`vtype`: type of variables in MVar object.

`prefix`: name prefix of variables in MVar object.

**Return**

new MVar object.

### Model.AddMVar()

Add a MVar object in N-dimensions to model.

**Synopsis**

```
MVar AddMVar(
    Shape shp,
    double lb,
    double ub,
    double obj,
    char vtype,
    string prefix)
```

**Arguments**

`shp`: shape of MVar object.

`lb`: lower bound for variables in MVar object.

`ub`: upper bound for variables in MVar object.

`obj`: objective coefficient for variables in MVar object.

`vtype`: type of variables in MVar object.

`prefix`: name prefix of variables in MVar object.

**Return**

new MVar object.

### Model.AddMVar()

Add a MVar object in N-dimensions to model.

**Synopsis**

```
MVar AddMVar(
    Shape shp,
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    string prefix)
```

**Arguments**

`shp`: shape of MVar object.

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`prefix`: name prefix of variables in MVar object.

**Return**

new MVar object.

### Model.AddNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint AddNlConstr(
    NlExpr expr,
    char sense,
    double rhs,
    string name)
```

**Arguments**

`expr`: nonlinear expression for the new constraint.

`sense`: sense for new nonlinear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`name`: optional, name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model.AddNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint AddNlConstr(
    NlExpr lhs,
    char sense,
    NlExpr rhs,
    string name)
```

**Arguments**

`lhs`: left hand side nonlinear expression for the new constraint.

`sense`: sense for new nonlinear constraint, other than range sense.

`rhs`: right hand side nonlinear expression for the new constraint.

`name`: optional, name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model.AddNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint AddNlConstr(
    NlExpr expr,
    double lb,
    double ub,
    string name)
```

**Arguments**

`expr`: nonlinear expression for the new constraint.

`lb`: lower bound for the new nonlinear constraint.

`ub`: upper bound for the new nonlinear constraint

`name`: optional, name of new constraint.

**Return**

new nonlinear constraint object.

### Model.AddNlConstr()

Add a nonlinear constraint to a model.

**Synopsis**

```
NlConstraint AddNlConstr(NlConstrBuilder builder, string name)
```

**Arguments**

`builder`: builder for the new nonlinear constraint.

`name`: optional, name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model.AddNlConstrs()

Add nonlinear constraints to a model.

**Synopsis**

```
NlConstrArray AddNlConstrs(NlConstrBuilderArray builders, string
prefix)
```

**Arguments**

`builders`: builders for new nonlinear constraints.

`prefix`: name prefix for new constraints.

**Return**

array of new nonlinear constraint objects.

### Model.AddOnesMat()

Add a dense symmetric matrix of value one to a model.

**Synopsis**

```
SymMatrix AddOnesMat(int dim)
```

**Arguments**

`dim`: dimension of dense symmetric matrix.

**Return**

new symmetric matrix object.

### Model.AddPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint AddPsdConstr(
    PsdExpr expr,
    char sense,
    double rhs,
    string name)
```

**Arguments**

`expr`: PSD expression for new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: double value at right side of the new PSD constraint.

`name`: optional, name of new PSD constraint.

**Return**

new PSD constraint object.

### Model.AddPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint AddPsdConstr(
    PsdExpr expr,
    double lb,
    double ub,
    string name)
```

**Arguments**

`expr`: expression for new PSD constraint.

`lb`: lower bound for ew PSD constraint.

`ub`: upper bound for new PSD constraint

`name`: optional, name of new PSD constraint.

**Return**

new PSD constraint object.

### Model.AddPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint AddPsdConstr(
    PsdExpr lhs,
    char sense,
    PsdExpr rhs,
    string name)
```

**Arguments**

`lhs`: PSD expression at left side of new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: PSD expression at right side of new PSD constraint.

`name`: optional, name of new PSD constraint.

**Return**

new PSD constraint object.

### Model.AddPsdConstr()

Add a PSD constraint to a model.

**Synopsis**

```
PsdConstraint AddPsdConstr(PsdConstrBuilder builder, string name)
```

**Arguments**

`builder`: builder for new PSD constraint.

`name`: optional, name of new PSD constraint.

**Return**

new PSD constraint object.

### Model.AddPsdVar()

Add a new PSD variable to model.

**Synopsis**

```
PsdVar AddPsdVar(int dim, string name)
```

**Arguments**

`dim`: dimension of new PSD variable.

`name`: name of new PSD variable.

**Return**

PSD variable object.

### Model.AddPsdVars()

Add new PSD variables to model.

**Synopsis**

```
PsdVarArray AddPsdVars(
    int count,
    int[] dims,
    string prefix)
```

**Arguments**

`count`: number of new PSD variables.

`dims`: array of dimensions of new PSD variables.

`prefix`: name prefix of new PSD variables, default prefix is PSD_V.

**Return**

array of PSD variable objects.

### Model.AddQConstr()

Add a quadratic constraint to model.

**Synopsis**

```
QConstraint AddQConstr(
    QuadExpr expr,
    char sense,
    double rhs,
    string name)
```

**Arguments**

`expr`: quadratic expression for the new constraint.

`sense`: sense for new quadratic constraint.

`rhs`: double value at right side of the new quadratic constraint.

`name`: optional, name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model.AddQConstr()

Add a quadratic constraint to model.

**Synopsis**

```
QConstraint AddQConstr(
    QuadExpr lhs,
    char sense,
    QuadExpr rhs,
    string name)
```

**Arguments**

`lhs`: quadratic expression at left side of the new quadratic constraint.

`sense`: sense for new quadratic constraint.

`rhs`: quadratic expression at right side of the new quadratic constraint.

`name`: optional, name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model.AddQConstr()

Add a quadratic constraint to a model.

**Synopsis**

```
QConstraint AddQConstr(QConstrBuilder builder, string name)
```

**Arguments**

`builder`: builder for the new quadratic constraint.

`name`: optional, name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model.AddSos()

Add a SOS constraint to model.

**Synopsis**

```
Sos AddSos(SosBuilder builder)
```

**Arguments**

`builder`: builder for new SOS constraint.

**Return**

new SOS constraint object.

### Model.AddSos()

Add a SOS constraint to model.

**Synopsis**

```
Sos AddSos(
    Var[] vars,
    double[] weights,
    int type)
```

**Arguments**

`vars`: variables that participate in the SOS constraint.

`weights`: weights for variables in the SOS constraint.

`type`: type of SOS constraint.

**Return**

new SOS constraint object.

### Model.AddSos()

Add a SOS constraint to model.

**Synopsis**

```
Sos AddSos(
    VarArray vars,
    double[] weights,
    int type)
```

**Arguments**

`vars`: variables that participate in the SOS constraint.

`weights`: weights for variables in the SOS constraint.

`type`: type of SOS constraint.

**Return**

new SOS constraint object.

### Model.AddSparseMat()

Add a sparse symmetric matrix to a model.

**Synopsis**

```
SymMatrix AddSparseMat(
    int dim,
    int nElems,
    int[] rows,
    int[] cols,
    double[] vals)
```

**Arguments**

`dim`: dimension of the sparse symmetric matrix.

`nElems`: number of non-zero elements in the sparse symmetric matrix.

`rows`: array of row indexes of non-zero elements.

`cols`: array of col indexes of non-zero elements.

`vals`: array of values of non-zero elements.

**Return**

new symmetric matrix object.

### Model.AddSymMat()

Given a symmetric matrix expression, add results matrix to model.

**Synopsis**

```
SymMatrix AddSymMat(SymMatExpr expr)
```

**Arguments**

`expr`: symmetric matrix expression object.

**Return**

results symmetric matrix object.

### Model.AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(
    Expr lhs,
    char sense,
    double rhs,
    string name)
```

**Arguments**

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

`name`: optional, name of user cut.

### Model.AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(
    Expr lhs,
    char sense,
    Expr rhs,
    string name)
```

**Arguments**

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

`name`: optional, name of user cut.

### Model.AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(ConstrBuilder builder, string name)
```

**Arguments**

`builder`: builder for user cut.

`name`: optional, name of user cut.

### Model.AddUserCuts()

Add user cuts to model.

**Synopsis**

```
void AddUserCuts(ConstrBuilderArray builders, string prefix)
```

**Arguments**

`builders`: array of builders for user cuts.

`prefix`: name prefix of new user cuts.

### Model.AddVar()

Add a variable and the associated non-zero coefficients as column.

**Synopsis**

```
Var AddVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    string name)
```

**Arguments**

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`name`: optional, name for new variable.

**Return**

new variable object.

### Model.AddVar()

Add a variable and the associated non-zero coefficients as column.

**Synopsis**

```
Var AddVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    Column col,
    string name)
```

**Arguments**

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`col`: column object for specifying a set of constraints to which the variable belongs.

`name`: optional, name for new variable.

**Return**

new variable object.

### Model.AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    char vtype,
    string prefix)
```

**Arguments**

`count`: the number of variables to add.

`vtype`: variable types for new variables.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

**Return**

array of new variable objects.

### Model.AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    double lb,
    double ub,
    double obj,
    char vtype,
    string prefix)
```

**Arguments**

`count`: the number of variables to add.

`lb`: lower bound for new variables.

`ub`: upper bound for new variables.

`obj`: objective coefficient for new variables.

`vtype`: variable type for new variables.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

**Return**

array of new variable objects.

### Model.AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    int count,
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    string prefix)
```

**Arguments**

`count`: the number of variables to add.

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

**Return**

array of new variable objects.

### Model.AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    Column[] cols,
    string prefix)
```

**Arguments**

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`cols`: column objects for specifying a set of constraints to which each new variable belongs.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

**Return**

array of new variable objects.

### Model.AddVars()

Add new variables to model.

**Synopsis**

```
VarArray AddVars(
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    ColumnArray cols,
    string prefix)
```

**Arguments**

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`cols`: columnarray for specifying a set of constraints to which each new variable belongs.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

**Return**

array of new variable objects.

### Model.BSolve()

Given a sparse vector, perform a backward solve. The result is stored in place in the same vector.

**Synopsis**

```
int BSolve(
    NdArray<double> vals,
    NdArray<int> idxes,
    int len)
```

**Arguments**

`vals`: input values of the sparse vector; output values of the result, up to row size.

`idxes`: input indices of the sparse vector; output indices of the result, up to row size.

`len`: length of the sparse input vector.

**Return**

length of the result sparse vector.

### Model.Clear()

Clear all settings including problem itself.

**Synopsis**

```
void Clear()
```

### Model.Clone()

Deep copy COPT model.

**Synopsis**

```
Model Clone()
```

**Return**

cloned model object.

### Model.ComputeIIS()

Compute IIS for infeasible model.

**Synopsis**

```
void ComputeIIS()
```

### Model.ComputeShadowPriceRange()

Compute the shadow price range for a given constraint.

**Synopsis**

```
void ComputeShadowPriceRange(
    int iRow,
    int ifRhs,
    out double lower,
    out double upper)
```

**Arguments**

`iRow`: constraint (row) index.

`ifRhs`: for range constraint, whether to use RHS for computing shadow price.

`lower`: output lower bound of the shadow price.

`upper`: output upper bound of the shadow price.

### Model.DelNlObj()

delete nonlinear part of objective in model.

**Synopsis**

```
void DelNlObj()
```

### Model.DelObjN()

Delete linear part of specific multi-objective function in model.

**Synopsis**

```
void DelObjN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

### Model.DelPsdObj()

delete PSD part of objective in model.

**Synopsis**

```
void DelPsdObj()
```

### Model.DelQuadObj()

delete quadratic part of objective in model.

**Synopsis**

```
void DelQuadObj()
```

### Model.Dispose()

Release this model reference on the calling thread. Prefer disposing explicitly (or via a using statement) on the same thread, instead of relying on the GC finalizer thread. Note the underlying native problem is reference-counted and destroyed when the last referring to it is released.

**Synopsis**

```
void Dispose()
```

### Model.FeasRelax()

Compute feasibility relaxation for infeasible model.

**Synopsis**

```
void FeasRelax(
    VarArray vars,
    double[] colLowPen,
    double[] colUppPen,
    ConstrArray cons,
    double[] rowBndPen,
    double[] rowUppPen)
```

**Arguments**

`vars`: an array of variables.

`colLowPen`: penalties for lower bounds of variables.

`colUppPen`: penalties for upper bounds of variables.

`cons`: an array of constraints.

`rowBndPen`: penalties for right hand sides of constraints.

`rowUppPen`: penalties for upper bounds of range constraints.

### Model.FeasRelax()

Compute feasibility relaxation for infeasible model.

**Synopsis**

```
void FeasRelax(int ifRelaxVars, int ifRelaxCons)
```

**Arguments**

`ifRelaxVars`: whether to relax variables.

`ifRelaxCons`: whether to relax constraints.

### Model.FSolve()

Given a sparse vector, perform a forward solve. The result is stored in place in the same vector.

**Synopsis**

```
int FSolve(
    NdArray<double> vals,
    NdArray<int> idxes,
    int len)
```

**Arguments**

`vals`: input values of the sparse vector; output values of the result, up to row size.

`idxes`: input indices of the sparse vector; output indices of the result, up to row size.

`len`: length of the sparse input vector.

**Return**

length of the result sparse vector.

### Model.Get()

Query values of information associated with variables.

**Synopsis**

```
double[] Get(string name, Var[] vars)
```

**Arguments**

`name`: name of information.

`vars`: a list of interested variables.

**Return**

values of information.

### Model.Get()

Query values of information associated with variables.

**Synopsis**

```
double[] Get(string name, VarArray vars)
```

**Arguments**

`name`: name of information.

`vars`: array of interested variables.

**Return**

values of information.

### Model.Get()

Query values of information associated with constraints.

**Synopsis**

```
double[] Get(string name, Constraint[] constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of interested constraints.

**Return**

values of information.

### Model.Get()

Query values of information associated with constraints.

**Synopsis**

```
double[] Get(string name, ConstrArray constrs)
```

**Arguments**

`name`: name of information.

`constrs`: array of interested constraints.

**Return**

values of information.

### Model.Get()

Query values of information associated with nonlinear constraints.

**Synopsis**

```
double[] Get(string name, NlConstraint[] constrs)
```

**Arguments**

`name`: name of information.

`constrs`: array of desired nonlinear constraints.

**Return**

output array of information values.

### Model.Get()

Query values of information associated with nonlinear constraints.

**Synopsis**

```
double[] Get(string name, NlConstrArray constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired nonlinear constraints.

**Return**

output array of information values.

### Model.Get()

Query values of information associated with quadratic constraints.

**Synopsis**

```
double[] Get(string name, QConstraint[] constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of interested quadratic constraints.

**Return**

values of information.

### Model.Get()

Query values of information associated with quadratic constraints.

**Synopsis**

```
double[] Get(string name, QConstrArray constrs)
```

**Arguments**

`name`: name of information.

`constrs`: array of interested quadratic constraints.

**Return**

values of information.

### Model.Get()

Query values of information associated with PSD constraints.

**Synopsis**

```
double[] Get(string name, PsdConstraint[] constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired PSD constraints.

**Return**

output array of information values.

### Model.Get()

Query values of information associated with PSD constraints.

**Synopsis**

```
double[] Get(string name, PsdConstrArray constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired PSD constraints.

**Return**

output array of information values.

### Model.GetAffineCone()

Get an affine cone constraint of given index in model.

**Synopsis**

```
AffineCone GetAffineCone(int idx)
```

**Arguments**

`idx`: index of the desired affine cone constraint.

**Return**

the desired affine cone constraint object.

### Model.GetAffineConeBuilder()

Get builder of given affine cone constraint in model.

**Synopsis**

```
AffineConeBuilder GetAffineConeBuilder(AffineCone cone)
```

**Arguments**

`cone`: affine cone constraint.

**Return**

desired affine cone constraint builder.

### Model.GetAffineConeBuilders()

Get builders of all affine cone constraints in model.

**Synopsis**

```
AffineConeBuilderArray GetAffineConeBuilders()
```

**Return**

array object of affine cone constraint builders.

### Model.GetAffineConeBuilders()

Get builders of given affine cone constraints in model.

**Synopsis**

```
AffineConeBuilderArray GetAffineConeBuilders(AffineCone[] cones)
```

**Arguments**

`cones`: array of affine cone constraints.

**Return**

array object of desired affine cone constraint builders.

### Model.GetAffineConeBuilders()

Get builders of given affine cone constraints in model.

**Synopsis**

```
AffineConeBuilderArray GetAffineConeBuilders(AffineConeArray cones)
```

**Arguments**

`cones`: array of affine cone constraints.

**Return**

array object of desired affine cone constraint builders.

### Model.GetAffineConeByName()

Get an affine cone constraint of given name in model.

**Synopsis**

```
AffineCone GetAffineConeByName(string name)
```

**Arguments**

`name`: name of the desired affine cone constraint.

**Return**

the desired affine cone constraint object.

### Model.GetAffineCones()

Get all affine cone constraints in model.

**Synopsis**

```
AffineConeArray GetAffineCones()
```

**Return**

array object of affine cone constraints.

### Model.GetBasisHead()

Get indices of the variables that make up the current basis.

**Synopsis**

```
NdArray<int> GetBasisHead()
```

**Return**

array of basis head indices.

### Model.GetCoeff()

Get the coefficient of variable in linear constraint.

**Synopsis**

```
double GetCoeff(Constraint constr, Var var)
```

**Arguments**

`constr`: The requested constraint.

`var`: The requested variable.

**Return**

The requested coefficient.

### Model.GetCol()

Get a column object that have a list of constraints in which the variable participates.

**Synopsis**

```
Column GetCol(Var var)
```

**Arguments**

`var`: a variable object.

**Return**

a column object associated with a variable.

### Model.GetColBasis()

Get status of column basis.

**Synopsis**

```
int[] GetColBasis()
```

**Return**

basis status.

### Model.GetCone()

Get a cone constraint of given index in model.

**Synopsis**

```
Cone GetCone(int idx)
```

**Arguments**

`idx`: index of the desired cone constraint.

**Return**

the desired cone constraint object.

### Model.GetConeBuilders()

Get builders of all cone constraints in model.

**Synopsis**

```
ConeBuilderArray GetConeBuilders()
```

**Return**

array object of cone constraint builders.

### Model.GetConeBuilders()

Get builders of given cone constraints in model.

**Synopsis**

```
ConeBuilderArray GetConeBuilders(Cone[] cones)
```

**Arguments**

`cones`: array of cone constraints.

**Return**

array object of desired cone constraint builders.

### Model.GetConeBuilders()

Get builders of given cone constraints in model.

**Synopsis**

```
ConeBuilderArray GetConeBuilders(ConeArray cones)
```

**Arguments**

`cones`: array of cone constraints.

**Return**

array object of desired cone constraint builders.

### Model.GetCones()

Get all cone constraints in model.

**Synopsis**

```
ConeArray GetCones()
```

**Return**

array object of cone constraints.

### Model.GetConstr()

Get a constraint of given index in model.

**Synopsis**

```
Constraint GetConstr(int idx)
```

**Arguments**

`idx`: index of the desired constraint.

**Return**

the desired constraint object.

### Model.GetConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

**Synopsis**

```
ConstrBuilder GetConstrBuilder(Constraint constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

constraint builder object.

### Model.GetConstrBuilders()

Get builders of all constraints in model.

**Synopsis**

```
ConstrBuilderArray GetConstrBuilders()
```

**Return**

array object of constraint builders.

### Model.GetConstrByName()

Get a constraint of given name in model.

**Synopsis**

```
Constraint GetConstrByName(string name)
```

**Arguments**

`name`: name of the desired constraint.

**Return**

the desired constraint object.

### Model.GetConstrLowerIIS()

Get IIS status of lower bounds of constraints.

**Synopsis**

```
int[] GetConstrLowerIIS(ConstrArray constrs)
```

**Arguments**

`constrs`: Array of constraints.

**Return**

IIS status of lower bounds of constraints.

### Model.GetConstrLowerIIS()

Get IIS status of lower bounds of constraints.

**Synopsis**

```
int[] GetConstrLowerIIS(Constraint[] constrs)
```

**Arguments**

`constrs`: Array of constraints.

**Return**

IIS status of lower bounds of constraints.

### Model.GetConstrs()

Get all constraints in model.

**Synopsis**

```
ConstrArray GetConstrs()
```

**Return**

array object of constraints.

### Model.GetConstrUpperIIS()

Get IIS status of upper bounds of constraints.

**Synopsis**

```
int[] GetConstrUpperIIS(ConstrArray constrs)
```

**Arguments**

`constrs`: Array of constraints.

**Return**

IIS status of upper bounds of constraints.

### Model.GetConstrUpperIIS()

Get IIS status of upper bounds of constraints.

**Synopsis**

```
int[] GetConstrUpperIIS(Constraint[] constrs)
```

**Arguments**

`constrs`: Array of constraints.

**Return**

IIS status of upper bounds of constraints.

### Model.GetDblAttr()

Get value of a COPT double attribute.

**Synopsis**

```
double GetDblAttr(string attr)
```

**Arguments**

`attr`: name of double attribute.

**Return**

value of double attribute.

### Model.GetDblAttrN()

Get value of a double attribute of a multi-objective function.

**Synopsis**

```
double GetDblAttrN(int idx, string attr)
```

**Arguments**

`idx`: index of a multi-objective function.

`attr`: name of double attribute.

**Return**

value of double attribute.

### Model.GetDblParam()

Get value of a COPT double parameter.

**Synopsis**

```
double GetDblParam(string param)
```

**Arguments**

`param`: name of integer parameter.

**Return**

value of double parameter.

### Model.GetDblParamN()

Get value of a double parameter of a multi-objective function.

**Synopsis**

```
double GetDblParamN(int idx, string param)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of double parameter.

**Return**

value of double parameter.

### Model.GetExpCone()

Get an exponential cone constraint of given index in model.

**Synopsis**

```
ExpCone GetExpCone(int idx)
```

**Arguments**

`idx`: index of the desired exponential cone constraint.

**Return**

the desired exponential cone constraint object.

### Model.GetExpConeBuilders()

Get builders of all exponential cone constraints in model.

**Synopsis**

```
ExpConeBuilderArray GetExpConeBuilders()
```

**Return**

array object of exponential cone constraint builders.

### Model.GetExpConeBuilders()

Get builders of given exponential cone constraints in model.

**Synopsis**

```
ExpConeBuilderArray GetExpConeBuilders(ExpCone[] cones)
```

**Arguments**

`cones`: array of exponential cone constraints.

**Return**

array object of desired exponential cone constraint builders.

### Model.GetExpConeBuilders()

Get builders of given exponential cone constraints in model.

**Synopsis**

```
ExpConeBuilderArray GetExpConeBuilders(ExpConeArray cones)
```

**Arguments**

`cones`: array of exponential cone constraints.

**Return**

array object of desired exponential cone constraint builders.

### Model.GetExpCones()

Get all exponential cone constraints in model.

**Synopsis**

```
ExpConeArray GetExpCones()
```

**Return**

array object of exponential cone constraints.

### Model.GetGenConstr()

Get a general constraint of given index in model.

**Synopsis**

```
GenConstr GetGenConstr(int idx)
```

**Arguments**

`idx`: index of the desired general constraint.

**Return**

the desired general constraint object.

### Model.GetGenConstrByName()

Get a general constraint of given name in model.

**Synopsis**

```
GenConstr GetGenConstrByName(string name)
```

**Arguments**

`name`: name of the desired general constraint.

**Return**

the desired general constraint object.

### Model.GetGenConstrIndicator()

Get builder of given general constraint of type indicator.

**Synopsis**

```
GenConstrBuilder GetGenConstrIndicator(GenConstr indicator)
```

**Arguments**

`indicator`: a general constraint of type indicator.

**Return**

builder object of general constraint of type indicator.

### Model.GetGenConstrIndicators()

Get builders of all general constraints in model.

**Synopsis**

```
GenConstrBuilderArray GetGenConstrIndicators()
```

**Return**

array object of general constraint builders.

### Model.GetGenConstrs()

Get all general constraints in model.

**Synopsis**

```
GenConstrArray GetGenConstrs()
```

**Return**

array object of general constraints.

### Model.GetIndicatorIIS()

Get IIS status of indicator constraints.

**Synopsis**

```
int[] GetIndicatorIIS(GenConstrArray genconstrs)
```

**Arguments**

`genconstrs`: Array of indicator constraints.

**Return**

IIS status of indicator constraints.

### Model.GetIndicatorIIS()

Get IIS status of indicator constraints.

**Synopsis**

```
int[] GetIndicatorIIS(GenConstr[] genconstrs)
```

**Arguments**

`genconstrs`: Array of indicator constraints.

**Return**

IIS status of indicator constraints.

### Model.GetIntAttr()

Get value of a COPT integer attribute.

**Synopsis**

```
int GetIntAttr(string attr)
```

**Arguments**

`attr`: name of integer attribute.

**Return**

value of integer attribute.

### Model.GetIntAttrN()

Get value of an integer attribute of a multi-objective function.

**Synopsis**

```
int GetIntAttrN(int idx, string attr)
```

**Arguments**

`idx`: index of a multi-objective function.

`attr`: name of integer attribute.

**Return**

value of integer attribute.

### Model.GetIntParam()

Get value of a COPT integer parameter.

**Synopsis**

```
int GetIntParam(string param)
```

**Arguments**

`param`: name of integer parameter.

**Return**

value of integer parameter.

### Model.GetIntParamN()

Get value of an integer parameter of a multi-objective function.

**Synopsis**

```
int GetIntParamN(int idx, string param)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of integer parameter.

**Return**

value of integer parameter.

### Model.GetLmiCoeff()

Get the symmetric matrix of variable in LMI constraint.

**Synopsis**

```
SymMatrix GetLmiCoeff(LmiConstraint constr, Var var)
```

**Arguments**

`constr`: The desired LMI constraint.

`var`: The desired variable.

**Return**

The associated coefficient matrix.

### Model.GetLmiConstr()

Get LMI constraint of given index in model.

**Synopsis**

```
LmiConstraint GetLmiConstr(int idx)
```

**Arguments**

`idx`: index of desired LMI constraint.

**Return**

LMI constraint object.

### Model.GetLmiConstrByName()

Get LMI constraint of given name in model.

**Synopsis**

```
LmiConstraint GetLmiConstrByName(string name)
```

**Arguments**

`name`: name of desired LMI constraint.

**Return**

LMI constraint object.

### Model.GetLmiConstrs()

Get all LMI constraints in model.

**Synopsis**

```
LmiConstrArray GetLmiConstrs()
```

**Return**

array object of LMI constraints.

### Model.GetLmiRhs()

Get the symmetric matrix of constant of LMI constraint.

**Synopsis**

```
SymMatrix GetLmiRhs(LmiConstraint constr)
```

**Arguments**

`constr`: The desired LMI constraint.

**Return**

matrix of constant term.

### Model.GetLmiRow()

Get variables and associated symmetric matrices that participate in a LMI constraint.

**Synopsis**

```
LmiExpr GetLmiRow(LmiConstraint constr)
```

**Arguments**

`constr`: given LMI constraint object.

**Return**

LMI expression object of LMI constraint.

### Model.GetLpSolution()

Get LP solution.

**Synopsis**

```
void GetLpSolution(
    out double[] value,
    out double[] slack,
    out double[] rowDual,
    out double[] redCost)
```

**Arguments**

`value`: out, solution values.

`slack`: out, slack values.

`rowDual`: out, dual values.

`redCost`: out, reduced costs.

### Model.GetNlConstr()

Get a nonlinear constraint of given index in model.

**Synopsis**

```
NlConstraint GetNlConstr(int idx)
```

**Arguments**

`idx`: index of the desired nonlinear constraint.

**Return**

the desired nonlinear constraint object.

### Model.GetNlConstrBuilder()

Get builder of a nonlinear constraint in model, including nonlinear expression, sense and RHS.

**Synopsis**

```
NlConstrBuilder GetNlConstrBuilder(NlConstraint constr)
```

**Arguments**

`constr`: a nonlinear constraint object.

**Return**

nonlinear constraint builder object.

### Model.GetNlConstrBuilders()

Get builders of all nonlinear constraints in model.

**Synopsis**

```
NlConstrBuilderArray GetNlConstrBuilders()
```

**Return**

array object of nonlinear constraint builders.

### Model.GetNlConstrByName()

Get a nonlinear constraint of given name in model.

**Synopsis**

```
NlConstraint GetNlConstrByName(string name)
```

**Arguments**

`name`: name of the desired constraint.

**Return**

the desired nonlinear constraint object.

### Model.GetNlConstrs()

Get all nonlinear constraints in model.

**Synopsis**

```
NlConstrArray GetNlConstrs()
```

**Return**

array object of nonlinear constraints.

### Model.GetNlObjective()

Get nonlinear objective of model.

**Synopsis**

```
NlExpr GetNlObjective()
```

**Return**

a nonlinear expression object.

### Model.GetNlRow()

Get nonlinear expression of a nonlinear constraint.

**Synopsis**

```
NlExpr GetNlRow(NlConstraint constr)
```

**Arguments**

`constr`: a nonlinear constraint object.

**Return**

output object of nonlinear expression.

### Model.GetObjective()

Get linear expression of objective for model.

**Synopsis**

```
Expr GetObjective()
```

**Return**

an linear expression object.

### Model.GetObjectiveN()

Get linear expression of a multi-objective function in model.

**Synopsis**

```
Expr GetObjectiveN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

**Return**

a linear expression object.

### Model.GetObjParamN()

Get value of objective parameter of a multi-objective function.

**Synopsis**

```
double GetObjParamN(int idx, string param)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of objective parameter, including priority, weight, abstol and reltol.

**Return**

value of objective parameter.

### Model.GetParamInfo()

Get current, default, minimum, maximum of COPT integer parameter.

**Synopsis**

```
void GetParamInfo(
    string name,
    out int cur,
    out int def,
    out int min,
    out int max)
```

**Arguments**

`name`: name of integer parameter.

`cur`: out, current value of integer parameter.

`def`: out, default value of integer parameter.

`min`: out, minimum value of integer parameter.

`max`: out, maximum value of integer parameter.

### Model.GetParamInfo()

Get current, default, minimum, maximum of COPT double parameter.

**Synopsis**

```
void GetParamInfo(
    string name,
    out double cur,
    out double def,
    out double min,
    out double max)
```

**Arguments**

`name`: name of integer parameter.

`cur`: out, current value of double parameter.

`def`: out, default value of double parameter.

`min`: out, minimum value of double parameter.

`max`: out, maximum value of double parameter.

### Model.GetPoolObjVal()

Get the idx-th objective value in solution pool.

**Synopsis**

```
double GetPoolObjVal(int idx)
```

**Arguments**

`idx`: Index of solution.

**Return**

The requested objective value.

### Model.GetPoolObjValN()

Get the objective value of required multi-objective function in solution pool.

**Synopsis**

```
double GetPoolObjValN(int idx, int iSol)
```

**Arguments**

`idx`: index of a multi-objective function.

`iSol`: index of solution.

**Return**

value of required multi-objective function.

### Model.GetPoolSolution()

Get the idx-th solution in solution pool.

**Synopsis**

```
double[] GetPoolSolution(int idx, VarArray vars)
```

**Arguments**

`idx`: Index of solution.

`vars`: The requested variables.

**Return**

The requested solution.

### Model.GetPoolSolution()

Get the idx-th solution in solution pool.

**Synopsis**

```
double[] GetPoolSolution(int idx, Var[] vars)
```

**Arguments**

`idx`: Index of solution.

`vars`: The requested variables.

**Return**

The requested solution.

### Model.GetPsdCoeff()

Get the symmetric matrix of PSD variable in a PSD constraint.

**Synopsis**

```
SymMatrix GetPsdCoeff(PsdConstraint constr, PsdVar var)
```

**Arguments**

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

**Return**

The associated coefficient matrix.

### Model.GetPsdConstr()

Get a PSD constraint of given index in model.

**Synopsis**

```
PsdConstraint GetPsdConstr(int idx)
```

**Arguments**

`idx`: index of desired PSD constraint.

**Return**

PSD constraint object.

### Model.GetPsdConstrBuilder()

Get builder of a PSD constraint in model, including PSD variables, sense and associated symmetric matrices.

**Synopsis**

```
PsdConstrBuilder GetPsdConstrBuilder(PsdConstraint constr)
```

**Arguments**

`constr`: PSD constraint object.

**Return**

PSD constraint builder object.

### Model.GetPsdConstrBuilders()

Get builders of all PSD constraints in model.

**Synopsis**

```
PsdConstrBuilderArray GetPsdConstrBuilders()
```

**Return**

array object of PSD constraint builders.

### Model.GetPsdConstrByName()

Get a PSD constraint of given name in model.

**Synopsis**

```
PsdConstraint GetPsdConstrByName(string name)
```

**Arguments**

`name`: name of desired PSD constraint.

**Return**

PSD constraint object.

### Model.GetPsdConstrs()

Get all PSD constraints in model.

**Synopsis**

```
PsdConstrArray GetPsdConstrs()
```

**Return**

array object of PSD constraints.

### Model.GetPsdObjective()

Get PSD objective of model.

**Synopsis**

```
PsdExpr GetPsdObjective()
```

**Return**

a PSD expression object.

### Model.GetPsdRow()

Get PSD variables and associated symmetric matrices that participate in a PSD constraint.

**Synopsis**

```
PsdExpr GetPsdRow(PsdConstraint constr)
```

**Arguments**

`constr`: PSD constraint object.

**Return**

PSD expression object of the PSD constraint.

### Model.GetPsdSolution()

Get PSD solution.

**Synopsis**

```
void GetPsdSolution(
    out double[] psdValue,
    out double[] psdSlack,
    out double[] psdRowDual,
    out double[] psdRedCost)
```

**Arguments**

`psdValue`: out, solution of PSD variables.

`psdSlack`: out, slack of PSD constraints.

`psdRowDual`: out, dual of PSD constraints.

`psdRedCost`: out, reduced costs of PSD variables.

### Model.GetPsdVar()

Get a PSD variable of given index in model.

**Synopsis**

```
PsdVar GetPsdVar(int idx)
```

**Arguments**

`idx`: index of the desired PSD variable.

**Return**

the desired PSD variable object.

### Model.GetPsdVarByName()

Get a PSD variable of given name in model.

**Synopsis**

```
PsdVar GetPsdVarByName(string name)
```

**Arguments**

`name`: name of the desired PSD variable.

**Return**

the desired PSD variable object.

### Model.GetPsdVars()

Get all PSD variables in model.

**Synopsis**

```
PsdVarArray GetPsdVars()
```

**Return**

array object of PSD variables.

### Model.GetQConstr()

Get a quadratic constraint of given index in model.

**Synopsis**

```
QConstraint GetQConstr(int idx)
```

**Arguments**

`idx`: index of the desired quadratic constraint.

**Return**

the desired quadratic constraint object.

### Model.GetQConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

**Synopsis**

```
QConstrBuilder GetQConstrBuilder(QConstraint constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

constraint builder object.

### Model.GetQConstrBuilders()

Get builders of all constraints in model.

**Synopsis**

```
QConstrBuilderArray GetQConstrBuilders()
```

**Return**

array object of constraint builders.

### Model.GetQConstrByName()

Get a quadratic constraint of given name in model.

**Synopsis**

```
QConstraint GetQConstrByName(string name)
```

**Arguments**

`name`: name of the desired constraint.

**Return**

the desired quadratic constraint object.

### Model.GetQConstrs()

Get all quadratic constraints in model.

**Synopsis**

```
QConstrArray GetQConstrs()
```

**Return**

array object of quadratic constraints.

### Model.GetQuadObjective()

Get quadratic objective of model.

**Synopsis**

```
QuadExpr GetQuadObjective()
```

**Return**

a quadratic expression object.

### Model.GetQuadRow()

Get two variables and associated coefficients that participate in a quadratic constraint.

**Synopsis**

```
QuadExpr GetQuadRow(QConstraint constr)
```

**Arguments**

`constr`: a quadratic constraint object.

**Return**

quadratic expression object of the constraint.

### Model.GetRow()

Get variables that participate in a constraint, and the associated coefficients.

**Synopsis**

```
Expr GetRow(Constraint constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

expression object of the constraint.

### Model.GetRowBasis()

Get status of row basis.

**Synopsis**

```
int[] GetRowBasis()
```

**Return**

basis status.

### Model.GetSolution()

Get MIP solution.

**Synopsis**

```
double[] GetSolution()
```

**Return**

solution values.

### Model.GetSos()

Get a SOS constraint of given index in model.

**Synopsis**

```
Sos GetSos(int idx)
```

**Arguments**

`idx`: index of the desired SOS constraint.

**Return**

the desired SOS constraint object.

### Model.GetSosBuilders()

Get builders of all SOS constraints in model.

**Synopsis**

```
SosBuilderArray GetSosBuilders()
```

**Return**

array object of SOS constraint builders.

### Model.GetSosBuilders()

Get builders of given SOS constraints in model.

**Synopsis**

```
SosBuilderArray GetSosBuilders(Sos[] soss)
```

**Arguments**

`soss`: array of SOS constraints.

**Return**

array object of desired SOS constraint builders.

### Model.GetSosBuilders()

Get builders of given SOS constraints in model.

**Synopsis**

```
SosBuilderArray GetSosBuilders(SosArray soss)
```

**Arguments**

`soss`: array of SOS constraints.

**Return**

array object of desired SOS constraint builders.

### Model.GetSOSIIS()

Get IIS status of SOS constraints.

**Synopsis**

```
int[] GetSOSIIS(SosArray soss)
```

**Arguments**

`soss`: Array of SOS constraints.

**Return**

IIS status of SOS constraints.

### Model.GetSOSIIS()

Get IIS status of SOS constraints.

**Synopsis**

```
int[] GetSOSIIS(Sos[] soss)
```

**Arguments**

`soss`: Array of SOS constraints.

**Return**

IIS status of SOS constraints.

### Model.GetSoss()

Get all SOS constraints in model.

**Synopsis**

```
SosArray GetSoss()
```

**Return**

array object of SOS constraints.

### Model.GetSymMat()

Get a symmetric matrix of given index in model.

**Synopsis**

```
SymMatrix GetSymMat(int idx)
```

**Arguments**

`idx`: index of the desired symmetric matrix.

**Return**

the desired symmetric matrix object.

### Model.GetVar()

Get a variable of given index in model.

**Synopsis**

```
Var GetVar(int idx)
```

**Arguments**

`idx`: index of the desired variable.

**Return**

the desired variable object.

### Model.GetVarByName()

Get a variable of given name in model.

**Synopsis**

```
Var GetVarByName(string name)
```

**Arguments**

`name`: name of the desired variable.

**Return**

the desired variable object.

### Model.GetVarLowerIIS()

Get IIS status of lower bounds of variables.

**Synopsis**

```
int[] GetVarLowerIIS(VarArray vars)
```

**Arguments**

`vars`: Array of variables.

**Return**

IIS status of lower bounds of variables.

### Model.GetVarLowerIIS()

Get IIS status of lower bounds of variables.

**Synopsis**

```
int[] GetVarLowerIIS(Var[] vars)
```

**Arguments**

`vars`: Array of variables.

**Return**

IIS status of lower bounds of variables.

### Model.GetVars()

Get all variables in model.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variable array object.

### Model.GetVarUpperIIS()

Get IIS status of upper bounds of variables.

**Synopsis**

```
int[] GetVarUpperIIS(VarArray vars)
```

**Arguments**

`vars`: Array of variables.

**Return**

IIS status of upper bounds of variables.

### Model.GetVarUpperIIS()

Get IIS status of upper bounds of variables.

**Synopsis**

```
int[] GetVarUpperIIS(Var[] vars)
```

**Arguments**

`vars`: Array of variables.

**Return**

IIS status of upper bounds of variables.

### Model.Interrupt()

Interrupt optimization of current problem.

**Synopsis**

```
void Interrupt()
```

### Model.LoadMipStart()

Load final initial values of variables to the problem.

**Synopsis**

```
void LoadMipStart()
```

### Model.LoadTuneParam()

Load specified tuned parameters into model.

**Synopsis**

```
void LoadTuneParam(int idx)
```

**Arguments**

`idx`: Index of tuned parameters.

### Model.Read()

Read problem, solution, basis, MIP start or COPT parameters from file.

**Synopsis**

```
void Read(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadBasis()

Read basis from file.

**Synopsis**

```
void ReadBasis(string filename)
```

**Arguments**

`filename`: an input file name

### Model.ReadBin()

Read problem in COPT binary format from file.

**Synopsis**

```
void ReadBin(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadCbf()

Read problem in CBF format from file.

**Synopsis**

```
void ReadCbf(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadJsonSol()

Read solution in format of JSON from file.

**Synopsis**

```
void ReadJsonSol(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadLp()

Read problem in LP format from file.

**Synopsis**

```
void ReadLp(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadMps()

Read problem in MPS format from file.

**Synopsis**

```
void ReadMps(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadMst()

Read MIP start information from file.

**Synopsis**

```
void ReadMst(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadOrd()

Read branching order from file.

**Synopsis**

```
void ReadOrd(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadParam()

Read COPT parameters from file.

**Synopsis**

```
void ReadParam(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadSdpa()

Read problem in SDPA format from file.

**Synopsis**

```
void ReadSdpa(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadSol()

Read solution from file.

**Synopsis**

```
void ReadSol(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.ReadTune()

Read tuning parameters from file.

**Synopsis**

```
void ReadTune(string filename)
```

**Arguments**

`filename`: an input file name.

### Model.Remove()

Remove an array of variables from model.

**Synopsis**

```
void Remove(Var[] vars)
```

**Arguments**

`vars`: a list of variables.

### Model.Remove()

Remove an array of variables from model.

**Synopsis**

```
void Remove(VarArray vars)
```

**Arguments**

`vars`: array of variables.

### Model.Remove()

Remove a list of constraints from model.

**Synopsis**

```
void Remove(Constraint[] constrs)
```

**Arguments**

`constrs`: a list of constraints.

### Model.Remove()

Remove a list of constraints from model.

**Synopsis**

```
void Remove(ConstrArray constrs)
```

**Arguments**

`constrs`: an array of constraints.

### Model.Remove()

Remove an array of nonlinear constraints from model.

**Synopsis**

```
void Remove(NlConstraint[] constrs)
```

**Arguments**

`constrs`: array of nonlinear constraints.

### Model.Remove()

Remove a list of nonlinear constraints from model.

**Synopsis**

```
void Remove(NlConstrArray constrs)
```

**Arguments**

`constrs`: array object of nonlinear constraints.

### Model.Remove()

Remove a list of SOS constraints from model.

**Synopsis**

```
void Remove(Sos[] soss)
```

**Arguments**

`soss`: a list of SOS constraints.

### Model.Remove()

Remove a list of SOS constraints from model.

**Synopsis**

```
void Remove(SosArray soss)
```

**Arguments**

`soss`: an array of SOS constraints.

### Model.Remove()

Remove a list of cone constraints from model.

**Synopsis**

```
void Remove(Cone[] cones)
```

**Arguments**

`cones`: a list of cone constraints.

### Model.Remove()

Remove a list of cone constraints from model.

**Synopsis**

```
void Remove(ConeArray cones)
```

**Arguments**

`cones`: an array of cone constraints.

### Model.Remove()

Remove a list of exponential cone constraints from model.

**Synopsis**

```
void Remove(ExpCone[] cones)
```

**Arguments**

`cones`: a list of exponential cone constraints.

### Model.Remove()

Remove a list of exponential cone constraints from model.

**Synopsis**

```
void Remove(ExpConeArray cones)
```

**Arguments**

`cones`: an array of exponential cone constraints.

### Model.Remove()

Remove a list of affine cone constraints from model.

**Synopsis**

```
void Remove(AffineCone[] cones)
```

**Arguments**

`cones`: a list of affine cone constraints.

### Model.Remove()

Remove an array of affine cone constraints from model.

**Synopsis**

```
void Remove(AffineConeArray cones)
```

**Arguments**

`cones`: an array of affine cone constraints.

### Model.Remove()

Remove a list of general constraints from model.

**Synopsis**

```
void Remove(GenConstr[] genConstrs)
```

**Arguments**

`genConstrs`: a list of general constraints.

### Model.Remove()

Remove a list of general constraints from model.

**Synopsis**

```
void Remove(GenConstrArray genConstrs)
```

**Arguments**

`genConstrs`: an array of general constraints.

### Model.Remove()

Remove a list of quadratic constraints from model.

**Synopsis**

```
void Remove(QConstraint[] qconstrs)
```

**Arguments**

`qconstrs`: an array of quadratic constraints.

### Model.Remove()

Remove a list of quadratic constraints from model.

**Synopsis**

```
void Remove(QConstrArray qconstrs)
```

**Arguments**

`qconstrs`: an array of quadratic constraints.

### Model.Remove()

Remove a list of PSD variables from model.

**Synopsis**

```
void Remove(PsdVar[] vars)
```

**Arguments**

`vars`: an array of PSD variables.

### Model.Remove()

Remove a list of PSD variables from model.

**Synopsis**

```
void Remove(PsdVarArray vars)
```

**Arguments**

`vars`: an array of PSD variables.

### Model.Remove()

Remove a list of PSD constraints from model.

**Synopsis**

```
void Remove(PsdConstraint[] constrs)
```

**Arguments**

`constrs`: an array of PSD constraints.

### Model.Remove()

Remove a list of PSD constraints from model.

**Synopsis**

```
void Remove(PsdConstrArray constrs)
```

**Arguments**

`constrs`: an array of PSD constraints.

### Model.Remove()

Remove a list of LMI constraints from model.

**Synopsis**

```
void Remove(LmiConstrArray constrs)
```

**Arguments**

`constrs`: an array of LMI constraints.

### Model.Remove()

Remove a list of LMI constraints from model.

**Synopsis**

```
void Remove(LmiConstraint[] constrs)
```

**Arguments**

`constrs`: an array of LMI constraints.

### Model.Reset()

Reset solution of problem only.

**Synopsis**

```
void Reset()
```

### Model.ResetAll()

Reset solution in problem, and additional information such as MIP start, etc.

**Synopsis**

```
void ResetAll()
```

### Model.ResetObjParamN()

Reset objective parameters of a multi-objective function.

**Synopsis**

```
void ResetObjParamN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

### Model.ResetParam()

Reset parameters to default settings.

**Synopsis**

```
void ResetParam()
```

### Model.ResetParamN()

Reset double and integer parameters of a multi-objective function.

**Synopsis**

```
void ResetParamN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

### Model.Set()

Set values of information associated with variables.

**Synopsis**

```
void Set(
    string name,
    Var[] vars,
    double[] vals)
```

**Arguments**

`name`: name of information.

`vars`: a list of interested variables.

`vals`: values of information.

### Model.Set()

Set values of information associated with variables.

**Synopsis**

```
void Set(
    string name,
    VarArray vars,
    double[] vals)
```

**Arguments**

`name`: name of information.

`vars`: array of interested variables.

`vals`: values of information.

### Model.Set()

Set values of information associated with constraints.

**Synopsis**

```
void Set(
    string name,
    Constraint[] constrs,
    double[] vals)
```

**Arguments**

`name`: name of information.

`constrs`: a list of interested constraints.

`vals`: values of information.

### Model.Set()

Set values of information associated with constraints.

**Synopsis**

```
void Set(
    string name,
    ConstrArray constrs,
    double[] vals)
```

**Arguments**

`name`: name of information.

`constrs`: array of interested constraints.

`vals`: values of information.

### Model.Set()

Set values of information associated with nonlinear constraints.

**Synopsis**

```
void Set(
    string name,
    NlConstraint[] constrs,
    double[] vals)
```

**Arguments**

`name`: name of double information.

`constrs`: array of desired nonlinear constraints.

`vals`: array of values of information.

### Model.Set()

Set values of information associated with nonlinear constraints.

**Synopsis**

```
void Set(
    string name,
    NlConstrArray constrs,
    double[] vals)
```

**Arguments**

`name`: name of double information.

`constrs`: a list of desired nonlinear constraints.

`vals`: array of values of information.

### Model.Set()

Set values of information associated with PSD constraints.

**Synopsis**

```
void Set(
    string name,
    PsdConstraint[] constrs,
    double[] vals)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired PSD constraints.

`vals`: array of values of information.

### Model.Set()

Set values of information associated with PSD constraints.

**Synopsis**

```
void Set(
    string name,
    PsdConstrArray constrs,
    double[] vals)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired PSD constraints.

`vals`: array of values of information.

### Model.SetBasis()

Set column and row basis status to model.

**Synopsis**

```
void SetBasis(int[] colbasis, int[] rowbasis)
```

**Arguments**

`colbasis`: status of column basis.

`rowbasis`: status of row basis.

### Model.SetCallback()

Set user callback to COPT model.

**Synopsis**

```
void SetCallback(CallbackBase cb, int cbctx)
```

**Arguments**

`cb`: user callback instance, inheriting from CallbackBase class.

`cbctx`: COPT callback context.

### Model.SetCoeff()

Set the coefficient of a variable in a linear constraint.

**Synopsis**

```
void SetCoeff(
    Constraint constr,
    Var var,
    double newVal)
```

**Arguments**

`constr`: The requested constraint.

`var`: The requested variable.

`newVal`: New coefficient.

### Model.SetCoeffs()

Set a list of coefficients in the model.

**Synopsis**

```
void SetCoeffs(
    Constraint[] constrs,
    Var[] vars,
    double[] vals)
```

**Arguments**

`constrs`: Array of constraints for coefficients to be set.

`vars`: Array of vars for coefficients to be set.

`vals`: New values for coefficients.

### Model.SetCoeffs()

Set a list of coefficients in the model.

**Synopsis**

```
void SetCoeffs(
    ConstrArray constrs,
    VarArray vars,
    double[] vals)
```

**Arguments**

`constrs`: A list of constraints for coefficients to be set.

`vars`: A list of vars for coefficients to be set.

`vals`: New values for coefficients.

### Model.SetDblParam()

Set value of a COPT double parameter.

**Synopsis**

```
void SetDblParam(string param, double val)
```

**Arguments**

`param`: name of integer parameter.

`val`: double value.

### Model.SetDblParamN()

Set value of a double parameter of a multi-objective function.

**Synopsis**

```
void SetDblParamN(
    int idx,
    string param,
    double val)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of double parameter.

`val`: new value of double parameter.

### Model.SetIntParam()

Set value of a COPT integer parameter.

**Synopsis**

```
void SetIntParam(string param, int val)
```

**Arguments**

`param`: name of integer parameter.

`val`: integer value.

### Model.SetIntParamN()

Set value of an integer parameter of a multi-objective function.

**Synopsis**

```
void SetIntParamN(
    int idx,
    string param,
    int val)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of integer parameter.

`val`: new value of integer parameter.

### Model.SetLmiCoeff()

Set the coefficient matrix of a variable in LMI constraint.

**Synopsis**

```
void SetLmiCoeff(
    LmiConstraint constr,
    Var var,
    SymMatrix mat)
```

**Arguments**

`constr`: The desired LMI constraint.

`var`: The desired variable.

`mat`: new coefficient matrix.

### Model.SetLmiRhs()

Set constant matrix of LMI constraint.

**Synopsis**

```
void SetLmiRhs(LmiConstraint constr, SymMatrix mat)
```

**Arguments**

`constr`: The desired LMI constraint.

`mat`: new constant matrix.

### Model.SetLpSolution()

Set LP solution.

**Synopsis**

```
void SetLpSolution(
    double[] value,
    double[] slack,
    double[] rowDual,
    double[] redCost)
```

**Arguments**

`value`: solution values.

`slack`: slack values.

`rowDual`: dual values.

`redCost`: reduced costs.

### Model.SetMipStart()

Set initial values for variables of given number, starting from the first one.

**Synopsis**

```
void SetMipStart(int count, double[] vals)
```

**Arguments**

`count`: the number of variables to set.

`vals`: values of variables.

### Model.SetMipStart()

Set initial value for the specified variable.

**Synopsis**

```
void SetMipStart(Var var, double val)
```

**Arguments**

`var`: an interested variable.

`val`: initial value of the variable.

### Model.SetMipStart()

Set initial value for the specified variable.

**Synopsis**

```
void SetMipStart(Var[] vars, double[] vals)
```

**Arguments**

`vars`: a list of interested variables.

`vals`: initial values of the variables.

### Model.SetMipStart()

Set initial value for the specified variable.

**Synopsis**

```
void SetMipStart(VarArray vars, double[] vals)
```

**Arguments**

`vars`: a list of interested variables.

`vals`: initial values of the variables.

### Model.SetNames()

Set names for given variables in model.

**Synopsis**

```
void SetNames(Var[] vars, string[] names)
```

**Arguments**

`vars`: array of variables.

`names`: string array of names for variables.

### Model.SetNames()

Set names for given variables in model.

**Synopsis**

```
void SetNames(VarArray vars, string[] names)
```

**Arguments**

`vars`: a list of variables.

`names`: string array of names for variables.

### Model.SetNames()

Set names for given constraints in model.

**Synopsis**

```
void SetNames(Constraint[] cons, string[] names)
```

**Arguments**

`cons`: array of constraints.

`names`: string array of names for constraints.

### Model.SetNames()

Set names for given constraints in model.

**Synopsis**

```
void SetNames(ConstrArray cons, string[] names)
```

**Arguments**

`cons`: a list of constraints.

`names`: string array of names for constraints.

### Model.SetNames()

Set names for given general constraints in model.

**Synopsis**

```
void SetNames(GenConstr[] genConstrs, string[] names)
```

**Arguments**

`genConstrs`: array of general constraints.

`names`: string array of names for general constraints.

### Model.SetNames()

Set names for given general constraints in model.

**Synopsis**

```
void SetNames(GenConstrArray genConstrs, string[] names)
```

**Arguments**

`genConstrs`: a list of general constraints.

`names`: string array of names for general constraints.

### Model.SetNames()

Set names for given nonlinear constraints in model.

**Synopsis**

```
void SetNames(NlConstraint[] cons, string[] names)
```

**Arguments**

`cons`: array of nonlinear constraints.

`names`: string array of names for nonlinear constraints.

### Model.SetNames()

Set names for given nonlinear constraints in model.

**Synopsis**

```
void SetNames(NlConstrArray cons, string[] names)
```

**Arguments**

`cons`: array object of nonlinear constraints.

`names`: string array of names for nonlinear constraints.

### Model.SetNames()

Set names for given quadratic constraints in model.

**Synopsis**

```
void SetNames(QConstraint[] cons, string[] names)
```

**Arguments**

`cons`: array of quadratic constraints.

`names`: string array of names for quadratic constraints.

### Model.SetNames()

Set names for given quadratic constraints in model.

**Synopsis**

```
void SetNames(QConstrArray cons, string[] names)
```

**Arguments**

`cons`: a list of quadratic constraints.

`names`: string array of names for quadratic constraints.

### Model.SetNames()

Set names for given PSD variables in model.

**Synopsis**

```
void SetNames(PsdVar[] vars, string[] names)
```

**Arguments**

`vars`: array of PSD variables.

`names`: string array of names for PSD variables.

### Model.SetNames()

Set names for given PSD variables in model.

**Synopsis**

```
void SetNames(PsdVarArray vars, string[] names)
```

**Arguments**

`vars`: a list of PSD variables.

`names`: string array of names for PSD variables.

### Model.SetNames()

Set names for given PSD constraints in model.

**Synopsis**

```
void SetNames(PsdConstraint[] cons, string[] names)
```

**Arguments**

`cons`: array of PSD constraints.

`names`: string array of names for PSD constraints.

### Model.SetNames()

Set names for given PSD constraints in model.

**Synopsis**

```
void SetNames(PsdConstrArray cons, string[] names)
```

**Arguments**

`cons`: a list of PSD constraints.

`names`: string array of names for PSD constraints.

### Model.SetNames()

Set names for given LMI constraints in model.

**Synopsis**

```
void SetNames(LmiConstraint[] cons, string[] names)
```

**Arguments**

`cons`: array of LMI constraints.

`names`: string array of names for LMI constraints.

### Model.SetNames()

Set names for given LMI constraints in model.

**Synopsis**

```
void SetNames(LmiConstrArray cons, string[] names)
```

**Arguments**

`cons`: a list of LMI constraints.

`names`: string array of names for LMI constraints.

### Model.SetNames()

Set names for given affine cone constraints in model.

**Synopsis**

```
void SetNames(AffineConeArray cones, string[] names)
```

**Arguments**

`cones`: an array of affine cone constraints.

`names`: string array of names for affine cone constraints.

### Model.SetNlObjective()

Set nonlinear objective for model.

**Synopsis**

```
void SetNlObjective(NlExpr expr, int sense)
```

**Arguments**

`expr`: nonlinear expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

### Model.SetNlPrimalStart()

Given count, set initial values for variables of NLP from beginning.

**Synopsis**

```
void SetNlPrimalStart(int count, double[] vals)
```

**Arguments**

`count`: the number of variables to set.

`vals`: initial values of variables.

### Model.SetNlPrimalStart()

Set initial value for the specified variable of NLP.

**Synopsis**

```
void SetNlPrimalStart(Var var, double val)
```

**Arguments**

`var`: an interested variable.

`val`: initial value of the variable.

### Model.SetNlPrimalStart()

Set initial values for an array of variables of NLP.

**Synopsis**

```
void SetNlPrimalStart(Var[] vars, double[] vals)
```

**Arguments**

`vars`: array of interested variables.

`vals`: initial values of variables.

### Model.SetNlPrimalStart()

Set initial values for variable array of NLP.

**Synopsis**

```
void SetNlPrimalStart(VarArray vars, double[] vals)
```

**Arguments**

`vars`: a list of interested variables.

`vals`: initial values of variables.

### Model.SetObjConst()

Set objective constant.

**Synopsis**

```
void SetObjConst(double constant)
```

**Arguments**

`constant`: constant value to set.

### Model.SetObjective()

Set objective for model.

**Synopsis**

```
void SetObjective(MExpression expr, int sense)
```

**Arguments**

`expr`: expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense

### Model.SetObjectiveN()

Set a multi-objective function in model.

**Synopsis**

```
void SetObjectiveN(
```

```
    int idx,
    MExpression expr,
    int sense,
    double priority,
    double weight,
    double abstol,
    double reltol)
```

**Arguments**

`idx`: index of a multi-objective function.

`expr`: linear expression of the multi-objective function.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

`priority`: an optional objective parameter. Default value is 0.0.

`weight`: an optional objective parameter. Default value is 1.0.

`abstol`: absolute tolerance is an optional objective parameter. Default value is 1e-6.

`reltol`: relative tolerance is an optional objective parameter. Default value is 0.0.

### Model.SetObjParamN()

Set value of objective parameter of a multi-objective function.

**Synopsis**

```
void SetObjParamN(
    int idx,
    string param,
    double val)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of objective parameter, including priority, weight, abstol and reltol.

`val`: new value of objective parameter.

### Model.SetObjSense()

Set objective sense for model.

**Synopsis**

```
void SetObjSense(int sense)
```

**Arguments**

`sense`: the objective sense.

### Model.SetPsdCoeff()

Set the coefficient matrix of a PSD variable in a PSD constraint.

**Synopsis**

```
void SetPsdCoeff(
    PsdConstraint constr,
    PsdVar var,
    SymMatrix mat)
```

**Arguments**

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

`mat`: new coefficient matrix.

### Model.SetPsdObjective()

Set PSD objective for model.

**Synopsis**

```
void SetPsdObjective(PsdExpr expr, int sense)
```

**Arguments**

`expr`: PSD expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

### Model.SetQuadObjective()

Set quadratic objective for model.

**Synopsis**

```
void SetQuadObjective(QuadExpr expr, int sense)
```

**Arguments**

`expr`: quadratic expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

### Model.SetSlackBasis()

Set slack basis to model.

**Synopsis**

```
void SetSlackBasis()
```

### Model.SetSolverLogCallback()

Set user log callback to COPT model. The model keeps a reference to the callback until it is replaced or the model is released.

**Synopsis**

```
void SetSolverLogCallback(LogCallback cb)
```

**Arguments**

`cb`: user log callback instance, inheriting from LogCallback class.

### Model.SetSolverLogFile()

Set log file for COPT.

**Synopsis**

```
void SetSolverLogFile(string filename)
```

**Arguments**

`filename`: log file name.

### Model.Solve()

Solve the model as MIP.

**Synopsis**

```
void Solve()
```

### Model.SolveLp()

Solve the model as LP.

**Synopsis**

```
void SolveLp()
```

### Model.Tune()

Tune model.

**Synopsis**

```
void Tune()
```

### Model.Update()

Update current model, including numerical range and removed items.

**Synopsis**

```
void Update()
```

### Model.Write()

Output problem, solution, basis, MIP start or modified COPT parameters to file.

**Synopsis**

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

**Arguments**

`filename`: an output file name.

### Model.WriteBasis()

Output optimal basis to a file of type '.bas'.

**Synopsis**

```
void WriteBasis(string filename)
```

**Arguments**

`filename`: an output file name.

### Model.WriteBin()

Output problem to a file as COPT binary format.

**Synopsis**

```
void WriteBin(string filename)
```

**Arguments**

`filename`: an output file name.

### Model.WriteIIS()

Output IIS to file.

**Synopsis**

```
void WriteIIS(string filename)
```

**Arguments**

`filename`: Output file name.

### Model.WriteJsonSol()

Output solution to a file of type '.json'.

**Synopsis**

```
void WriteJsonSol(string filename)
```

**Arguments**

`filename`: an output file name.

### Model.WriteLp()

Output problem to a file as LP format.

**Synopsis**

```
void WriteLp(string filename)
```

**Arguments**

`filename`: an output file name.

### Model.WriteMps()

Output problem to a file as MPS format.

**Synopsis**

```
void WriteMps(string filename)
```

**Arguments**

`filename`: an output file name.

### Model.WriteMpsStr()

Output MPS problem to problem buffer.

**Synopsis**

```
ProbBuffer WriteMpsStr()
```

**Return**

problem buffer for string of MPS problem.

### Model.WriteMst()

Output MIP start information to a file of type '.mst'.

**Synopsis**

```
void WriteMst(string filename)
```

**Arguments**

`filename`: an output file name.

### Model.WriteNL()

Output problem to a file as NL format.

**Synopsis**

```
void WriteNL(string filename)
```

**Arguments**

`filename`: an output file name.

### Model.WriteOrd()

Output branching order to file.

**Synopsis**

```
void WriteOrd(string filename)
```

**Arguments**

`filename`: Output file name.

### Model.WriteParam()

Output modified COPT parameters to a file of type '.par'.

**Synopsis**

```
void WriteParam(string filename)
```

**Arguments**

`filename`: an output file name.

### Model.WritePoolSol()

Output selected pool solution to a file of type '.sol'.

**Synopsis**

```
void WritePoolSol(int idx, string filename)
```

**Arguments**

`idx`: index of pool solution.

`filename`: an output file name.

### Model.WriteRelax()

Output feasibility relaxation problem to file.

**Synopsis**

```
void WriteRelax(string filename)
```

**Arguments**

`filename`: Output file name.

### Model.WriteSol()

Output solution to a file of type '.sol'.

**Synopsis**

```
void WriteSol(string filename)
```

**Arguments**

`filename`: an output file name.

#### Model.WriteTuneParam()

Output specified tuned parameters to a file of type '.par'.

**Synopsis**

```
void WriteTuneParam(int idx, string filename)
```

**Arguments**

`idx`: Index of tuned parameters.

`filename`: Output file name.

### 26.2.4 Var

COPT variable object. Variables are always associated with a particular model. User creates a variable object by adding a variable to a model, rather than by using constructor of Var class.

#### Var.Get()

Get information value of the variable. Support information of "Value", "RedCost", "PrimalRay", "LB", "UB" and "Obj".

**Synopsis**

```
double Get(string info)
```

**Arguments**

`info`: information name.

**Return**

information value.

#### Var.GetBasis()

Get basis status of the variable.

**Synopsis**

```
int GetBasis()
```

**Return**

Basis status.

#### Var.GetIdx()

Get index of the variable.

**Synopsis**

```
int GetIdx()
```

**Return**

variable index.

### Var.GetLowerIIS()

Get IIS status for lower bound of the variable.

**Synopsis**

```
int GetLowerIIS()
```

**Return**

IIS status.

### Var.GetName()

Get name of the variable.

**Synopsis**

```
string GetName()
```

**Return**

variable name.

### Var.GetType()

Get type of the variable.

**Synopsis**

```
char GetType()
```

**Return**

variable type.

### Var.GetUpperIIS()

Get IIS status for upper bound of the variable.

**Synopsis**

```
int GetUpperIIS()
```

**Return**

IIS status.

### Var.Remove()

Remove variable from model.

**Synopsis**

```
void Remove()
```

#### Var.Set()

Set information value of the variable. Support information of "LB", "UB" and "Obj".

**Synopsis**

```
void Set(string info, double val)
```

**Arguments**

`info`: information name.

`val`: new information value.

#### Var.SetName()

Set name of the variable.

**Synopsis**

```
void SetName(string name)
```

**Arguments**

`name`: variable name.

#### Var.SetType()

Set type of the variable.

**Synopsis**

```
void SetType(char vtype)
```

**Arguments**

`vtype`: variable type.

### 26.2.5 VarArray

COPT variable array object. To store and access a set of C# *Var* objects, Cardinal Optimizer provides C# VarArray class, which defines the following methods.

#### VarArray.VarArray()

Constructor of vararray.

**Synopsis**

```
VarArray()
```

#### VarArray.GetVar()

Get idx-th variable object.

**Synopsis**

```
Var GetVar(int idx)
```

**Arguments**

`idx`: index of the variable.

**Return**

variable object with index idx.

#### VarArray.PushBack()

Add a variable object to variable array.

**Synopsis**

```
void PushBack(Var var)
```

**Arguments**

`var`: a variable object.

#### VarArray.Size()

Get the number of variable objects.

**Synopsis**

```
int Size()
```

**Return**

number of variable objects.

### 26.2.6 Expr

COPT linear expression object. A linear expression consists of a constant term, a list of terms of variables and associated coefficients. Linear expressions are used to build constraints.

#### Expr.Expr()

Constructor of a constant linear expression with default constant value 0.

**Synopsis**

```
Expr(double constant)
```

**Arguments**

`constant`: optional, constant value in expression object.

#### Expr.Expr()

Constructor of a linear expression with one term.

**Synopsis**

```
Expr(Var var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term with default value 1.0.

### Expr.AddConstant()

Add extra constant to the expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: delta value to be added to expression constant.

### Expr.AddExpr()

Add an expression to self.

**Synopsis**

```
void AddExpr(Expr expr, double mult)
```

**Arguments**

`expr`: expression to be added.

`mult`: multiply constant.

### Expr.AddTerm()

Add a term to expression object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: a variable for new term.

`coeff`: coefficient for new term.

### Expr.AddTerms()

Add terms to expression object.

**Synopsis**

```
void AddTerms(Var[] vars, double coeff)
```

**Arguments**

`vars`: variables for added terms.

`coeff`: coefficient array for added terms with default value 1.0.

### Expr.AddTerms()

Add terms to expression object.

**Synopsis**

```
void AddTerms(Var[] vars, double[] coeffs)
```

**Arguments**

`vars`: variables for added terms.

`coeffs`: coefficients array for added terms.

### Expr.AddTerms()

Add terms to expression object.

**Synopsis**

```
void AddTerms(VarArray vars, double coeff)
```

**Arguments**

`vars`: variables for added terms.

`coeff`: coefficient array for added terms with default value 1.0.

### Expr.AddTerms()

Add terms to expression object.

**Synopsis**

```
void AddTerms(VarArray vars, double[] coeffs)
```

**Arguments**

`vars`: variables for added terms.

`coeffs`: coefficients array for added terms.

### Expr.Clone()

Deep copy linear expression object.

**Synopsis**

```
Expr Clone()
```

**Return**

cloned linear expression object.

### Expr.Divide()

Divide itself by double constant.

**Synopsis**

```
void Divide(double c)
```

**Arguments**

`c`: constant operand.

### Expr.Evaluate()

Evaluate linear expression after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of linear expression.

### Expr.GetCoeff()

Get coefficient from the i-th term in expression.

**Synopsis**

```
double GetCoeff(int i)
```

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term in expression object.

### Expr.GetConstant()

Get constant in expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in expression.

### Expr.GetVar()

Get variable from the i-th term in expression.

**Synopsis**

```
Var GetVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

variable of the i-th term in expression object.

### Expr.Multiply()

Multiply itself by a constant.

**Synopsis**

```
void Multiply(double c)
```

**Arguments**

`c`: constant operand.

### Expr.Remove()

Remove idx-th term from expression object.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### Expr.Remove()

Remove the term associated with variable from expression.

**Synopsis**

```
void Remove(Var var)
```

**Arguments**

`var`: a variable whose term should be removed.

### Expr.SetCoeff()

Set coefficient for the i-th term in expression.

**Synopsis**

```
void SetCoeff(int i, double val)
```

**Arguments**

`i`: index of the term.

`val`: coefficient of the term.

### Expr.SetConstant()

Set constant for the expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### Expr.Size()

Get number of terms in expression.

**Synopsis**

```
long Size()
```

**Return**

number of terms.

### 26.2.7 Constraint

COPT constraint object. Constraints are always associated with a particular model. User creates a constraint object by adding a constraint to a model, rather than by using constructor of Constraint class.

#### Constraint.ComputeShadowPriceRange()

Compute the shadow price range for a given constraint.

**Synopsis**

```
void ComputeShadowPriceRange(
    int ifRhs,
    out double lower,
    out double upper)
```

**Arguments**

`ifRhs`: for range constraint, whether to use RHS for computing shadow price.

`lower`: output lower bound of the shadow price.

`upper`: output upper bound of the shadow price.

#### Constraint.Get()

Get information value of the constraint. Support information of "Dual", "Slack", "LB" and "UB".

**Synopsis**

```
double Get(string info)
```

**Arguments**

`info`: name of information being queried.

**Return**

information value.

#### Constraint.GetBasis()

Get basis status of this constraint.

**Synopsis**

```
int GetBasis()
```

**Return**

basis status.

### Constraint.GetIdx()

Get index of the constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

the index of the constraint.

### Constraint.GetLowerIIS()

Get IIS status for lower bound of the constraint.

**Synopsis**

```
int GetLowerIIS()
```

**Return**

IIS status.

### Constraint.GetName()

Get name of the constraint.

**Synopsis**

```
string GetName()
```

**Return**

the name of the constraint.

### Constraint.GetUpperIIS()

Get IIS status for upper bound of the constraint.

**Synopsis**

```
int GetUpperIIS()
```

**Return**

IIS status.

### Constraint.Remove()

Remove this constraint from model.

**Synopsis**

```
void Remove()
```

#### Constraint.Set()

Set information value of the constraint. Support information of "LB" and "UB".

**Synopsis**

```
void Set(string info, double val)
```

**Arguments**

`info`: name of information.

`val`: new information value.

#### Constraint.SetName()

Set name for the constraint.

**Synopsis**

```
void SetName(string name)
```

**Arguments**

`name`: the name to set.

### 26.2.8 ConstrArray

COPT constraint array object. To store and access a set of C# *Constraint* objects, Cardinal Optimizer provides C# ConstrArray class, which defines the following methods.

#### ConstrArray.ConstrArray()

Constructor of constrarray object.

**Synopsis**

```
ConstrArray()
```

#### ConstrArray.GetConstr()

Get idx-th constraint object.

**Synopsis**

```
Constraint GetConstr(int idx)
```

**Arguments**

`idx`: index of the constraint.

**Return**

constraint object with index idx.

#### ConstrArray.PushBack()

Add a constraint object to constraint array.

**Synopsis**

```
void PushBack(Constraint constr)
```

**Arguments**

`constr`: a constraint object.

#### ConstrArray.Size()

Get the number of constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of constraint objects.

### 26.2.9 ConstrBuilder

COPT constraint builder object. To help building a constraint, given a linear expression, constraint sense and right-hand side value, Cardinal Optimizer provides C# ConstrBuilder class, which defines the following methods.

#### ConstrBuilder.ConstrBuilder()

Constructor of constrbuilder object.

**Synopsis**

```
ConstrBuilder()
```

#### ConstrBuilder.GetExpr()

Get expression associated with constraint.

**Synopsis**

```
Expr GetExpr()
```

**Return**

expression object.

#### ConstrBuilder.GetRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of the constraint.

#### ConstrBuilder.GetSense()

Get sense associated with constraint.

**Synopsis**

```
char GetSense()
```

**Return**

constraint sense.

#### ConstrBuilder.Set()

Set detail of a constraint to its builder object.

**Synopsis**

```
void Set(
    Expr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of the constraint.

#### ConstrBuilder.SetRange()

Set a range constraint to its builder.

**Synopsis**

```
void SetRange(Expr expr, double range)
```

**Arguments**

`expr`: expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must be greater than 0.

### 26.2.10 ConstrBuilderArray

COPT constraint builder array object. To store and access a set of C# *ConstrBuilder* objects, Cardinal Optimizer provides C# ConstrBuilderArray class, which defines the following methods.

#### ConstrBuilderArray.ConstrBuilderArray()

Constructor of constrbuilderarray object.

**Synopsis**

```
ConstrBuilderArray()
```

#### ConstrBuilderArray.GetBuilder()

Get idx-th constraint builder object.

**Synopsis**

```
ConstrBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the constraint builder.

**Return**

constraint builder object with index idx.

#### ConstrBuilderArray.PushBack()

Add a constraint builder object to constraint builder array.

**Synopsis**

```
void PushBack(ConstrBuilder builder)
```

**Arguments**

`builder`: a constraint builder object.

#### ConstrBuilderArray.Size()

Get the number of constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of constraint builder objects.

### 26.2.11 Column

COPT column object. A column consists of a list of constraints and associated coefficients. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients.

### Column.Column()

Constructor of column.

**Synopsis**

```
Column()
```

### Column.AddColumn()

Add a column to self.

**Synopsis**

```
void AddColumn(Column col, double mult)
```

**Arguments**

`col`: column object to be added.

`mult`: multiply constant.

### Column.AddTerm()

Add a term to column object.

**Synopsis**

```
void AddTerm(Constraint constr, double coeff)
```

**Arguments**

`constr`: a constraint for new term.

`coeff`: coefficient for new term.

### Column.AddTerms()

Add terms to column object.

**Synopsis**

```
void AddTerms(Constraint[] constrs, double coeff)
```

**Arguments**

`constrs`: constraints for added terms.

`coeff`: coefficient for added terms,default value is 1.

### Column.AddTerms()

Add terms to column object.

**Synopsis**

```
void AddTerms(Constraint[] constrs, double[] coeffs)
```

**Arguments**

`constrs`: constraints for added terms.

`coeffs`: coefficients for added terms.

### Column.AddTerms()

Add terms to column object.

**Synopsis**

```
void AddTerms(ConstrArray constrs, double coeff)
```

**Arguments**

`constrs`: constraints for added terms.

`coeff`: coefficient for added terms,default value is 1.

### Column.AddTerms()

Add terms to column object.

**Synopsis**

```
void AddTerms(ConstrArray constrs, double[] coeffs)
```

**Arguments**

`constrs`: constraints for added terms.

`coeffs`: coefficients for added terms.

### Column.Clear()

Clear all terms.

**Synopsis**

```
void Clear()
```

### Column.Clone()

Deep copy column object.

**Synopsis**

```
Column Clone()
```

**Return**

cloned column object.

### Column.GetCoeff()

Get coefficient from the i-th term in column object.

**Synopsis**

```
double GetCoeff(int i)
```

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term in column object.

#### Column.GetConstr()

Get constraint from the i-th term in column object.

**Synopsis**

```
Constraint GetConstr(int i)
```

**Arguments**

`i`: index of the term.

**Return**

constraint of the i-th term in column object.

#### Column.Remove()

Remove idx-th term from column object.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

#### Column.Remove()

Remove the term associated with constraint from column object.

**Synopsis**

```
void Remove(Constraint constr)
```

**Arguments**

`constr`: a constraint whose term should be removed.

#### Column.Size()

Get number of terms in column object.

**Synopsis**

```
int Size()
```

**Return**

number of terms.

### 26.2.12 ColumnArray

COPT column array object. To store and access a set of C# *Column* objects, Cardinal Optimizer provides C# ColumnArray class, which defines the following methods.

### ColumnArray.ColumnArray()

Constructor of columnarray object.

**Synopsis**

```
ColumnArray()
```

### ColumnArray.Clear()

Clear all column objects.

**Synopsis**

```
void Clear()
```

### ColumnArray.GetColumn()

Get idx-th column object.

**Synopsis**

```
Column GetColumn(int idx)
```

**Arguments**

`idx`: index of the column.

**Return**

column object with index idx.

### ColumnArray.PushBack()

Add a column object to column array.

**Synopsis**

```
void PushBack(Column col)
```

**Arguments**

`col`: a column object.

### ColumnArray.Size()

Get the number of column objects.

**Synopsis**

```
int Size()
```

**Return**

number of column objects.

### 26.2.13 Sos

COPT SOS constraint object. SOS constraints are always associated with a particular model. User creates an SOS constraint object by adding an SOS constraint to a model, rather than by using constructor of Sos class.

An SOS constraint can be type 1 or 2 (`COPT_SOS_TYPE1` or `COPT_SOS_TYPE2`).

#### Sos.GetIdx()

Get the index of SOS constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of SOS constraint.

#### Sos.GetIIS()

Get IIS status of the SOS constraint.

**Synopsis**

```
int GetIIS()
```

**Return**

IIS status.

#### Sos.Remove()

Remove the SOS constraint from model.

**Synopsis**

```
void Remove()
```

### 26.2.14 SosArray

COPT SOS constraint array object. To store and access a set of C# *Sos* objects, Cardinal Optimizer provides C# SosArray class, which defines the following methods.

#### SosArray.SosArray()

Constructor of sosarray object.

**Synopsis**

```
SosArray()
```

#### SosArray.GetSos()

Get idx-th SOS object.

**Synopsis**

```
Sos GetSos(int idx)
```

**Arguments**

`idx`: index of SOS.

**Return**

SOS object with index idx.

#### SosArray.PushBack()

Add a SOS constraint object to SOS constraint array.

**Synopsis**

```
void PushBack(Sos sos)
```

**Arguments**

`sos`: a SOS constraint object.

#### SosArray.Size()

Get the number of SOS constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of SOS constraint objects.

### 26.2.15 SosBuilder

COPT SOS constraint builder object. To help building an SOS constraint, given the SOS type, a set of variables and associated weights, Cardinal Optimizer provides C# SosBuilder class, which defines the following methods.

#### SosBuilder.SosBuilder()

Constructor of sosbuilder object.

**Synopsis**

```
SosBuilder()
```

### SosBuilder.GetSize()

Get number of terms in SOS constraint.

**Synopsis**

```
int GetSize()
```

**Return**

number of terms.

### SosBuilder.GetType()

Get type of SOS constraint.

**Synopsis**

```
int GetType()
```

**Return**

type of SOS constraint.

### SosBuilder.GetVar()

Get variable from the idx-th term in SOS constraint.

**Synopsis**

```
Var GetVar(int idx)
```

**Arguments**

`idx`: index of the term.

**Return**

variable of the idx-th term in SOS constraint.

### SosBuilder.GetVars()

Get all variables in a SOS constraint.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variables in a SOS constraint.

### SosBuilder.GetWeight()

Get weight from the idx-th term in SOS constraint.

**Synopsis**

```
double GetWeight(int idx)
```

**Arguments**

`idx`: index of the term.

**Return**

weight of the idx-th term in SOS constraint.

#### SosBuilder.GetWeights()

Get weights of all terms in SOS constraint.

**Synopsis**

```
double[] GetWeights()
```

**Return**

array of weights.

#### SosBuilder.Set()

Set variables and weights of SOS constraint.

**Synopsis**

```
void Set(
    VarArray vars,
    double[] weights,
    int type)
```

**Arguments**

`vars`: variable array object.

`weights`: pointer to array of weights.

`type`: type of SOS constraint.

### 26.2.16 SosBuilderArray

COPT SOS constraint builder array object. To store and access a set of C# *SosBuilder* objects, Cardinal Optimizer provides C# SosBuilderArray class, which defines the following methods.

#### SosBuilderArray.SosBuilderArray()

Constructor of sosbuilderarray object.

**Synopsis**

```
SosBuilderArray()
```

#### SosBuilderArray.GetBuilder()

Get idx-th SOS constraint builder object.

**Synopsis**

```
SosBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the SOS constraint builder.

**Return**

SOS constraint builder object with index idx.

#### SosBuilderArray.PushBack()

Add a SOS constraint builder object to SOS constraint builder array.

**Synopsis**

```
void PushBack(SosBuilder builder)
```

**Arguments**

`builder`: a SOS constraint builder object.

#### SosBuilderArray.Size()

Get the number of SOS constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of SOS constraint builder objects.

### 26.2.17 GenConstr

COPT general constraint object. General constraints are always associated with a particular model. User creates a general constraint object by adding a general constraint to a model, rather than by using constructor of GenConstr class.

#### GenConstr.GetIdx()

Get the index of the general constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of the general constraint.

#### GenConstr.GetIIS()

Get IIS status of the general constraint.

**Synopsis**

```
int GetIIS()
```

**Return**

IIS status.

#### GenConstr.GetName()

Get name of general constraint.

**Synopsis**

```
string GetName()
```

**Return**

the name of general constraint.

#### GenConstr.Remove()

Remove the general constraint from model.

**Synopsis**

```
void Remove()
```

#### GenConstr.SetName()

Set name for general constraint.

**Synopsis**

```
void SetName(string name)
```

**Arguments**

`name`: the name to set.

### 26.2.18 GenConstrArray

COPT general constraint array object. To store and access a set of C# *GenConstr* objects, Cardinal Optimizer provides C# GenConstrArray class, which defines the following methods.

#### GenConstrArray.GenConstrArray()

Constructor of genconstrarray.

**Synopsis**

```
GenConstrArray()
```

#### GenConstrArray.GetGenConstr()

Get idx-th general constraint object.

**Synopsis**

```
GenConstr GetGenConstr(int idx)
```

**Arguments**

`idx`: index of the general constraint.

**Return**

general constraint object with index idx.

#### GenConstrArray.PushBack()

Add a general constraint object to general constraint array.

**Synopsis**

```
void PushBack(GenConstr genconstr)
```

**Arguments**

`genconstr`: a general constraint object.

#### GenConstrArray.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: capacity number of general constraint object.

#### GenConstrArray.Size()

Get the number of general constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of general constraint objects.

### 26.2.19 GenConstrBuilder

COPT general constraint builder object. To help building a general constraint, given a binary variable and associated value, a linear expression and constraint sense, Cardinal Optimizer provides C# GenConstrBuilder class, which defines the following methods.

#### GenConstrBuilder.GenConstrBuilder()

Constructor of genconstrbuilder.

**Synopsis**

```
GenConstrBuilder()
```

#### GenConstrBuilder.GetBinVal()

Get binary value associated with general constraint.

**Synopsis**

```
int GetBinVal()
```

**Return**

binary value.

### GenConstrBuilder.GetBinVar()

Get binary variable associated with general constraint.

**Synopsis**

```
Var GetBinVar()
```

**Return**

binary vaiable object.

### GenConstrBuilder.GetExpr()

Get expression associated with general constraint.

**Synopsis**

```
Expr GetExpr()
```

**Return**

expression object.

### GenConstrBuilder.GetIndType()

Get type of general constraint.

**Synopsis**

```
int GetIndType()
```

**Return**

type of general constraint.

### GenConstrBuilder.GetSense()

Get sense associated with general constraint.

**Synopsis**

```
char GetSense()
```

**Return**

constraint sense.

### GenConstrBuilder.Set()

Set binary variable, binary value, expression and sense of general constraint.

**Synopsis**

```
void Set(
    Var binvar,
    int binval,
    Expr expr,
    char sense,
    int type)
```

**Arguments**

`binvar`: binary variable.

`binval`: binary value.

`expr`: expression object.

`sense`: general constraint sense.

`type`: type of general constraint.

### 26.2.20 GenConstrBuilderArray

COPT general constraint builder array object. To store and access a set of C# *GenConstrBuilder* objects, Cardinal Optimizer provides C# GenConstrBuilderArray class, which defines the following methods.

#### GenConstrBuilderArray.GenConstrBuilderArray()

Constructor of genconstrbuilderarray.

**Synopsis**

```
GenConstrBuilderArray()
```

#### GenConstrBuilderArray.GetBuilder()

Get idx-th general constraint builder object.

**Synopsis**

```
GenConstrBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the general constraint builder.

**Return**

general constraint builder object with index idx.

#### GenConstrBuilderArray.PushBack()

Add a general constraint builder object to general constraint builder array.

**Synopsis**

```
void PushBack(GenConstrBuilder builder)
```

**Arguments**

`builder`: a general constraint builder object.

#### GenConstrBuilderArray.Size()

Get the number of general constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of general constraint builder objects.

### 26.2.21 Cone

COPT cone constraint object. Cone constraints are always associated with a particular model. User creates a cone constraint object by adding a cone constraint to a model, rather than by using constructor of Cone class.

A cone constraint can be regular or rotated (`COPT_CONE_QUAD` or `COPT_CONE_RQUAD`).

#### Cone.GetIdx()

Get the index of a cone constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of the cone constraint.

#### Cone.Remove()

Remove the cone constraint from model.

**Synopsis**

```
void Remove()
```

### 26.2.22 ConeArray

COPT cone constraint array object. To store and access a set of C# *Cone* objects, Cardinal Optimizer provides C# ConeArray class, which defines the following methods.

#### ConeArray.ConeArray()

Constructor of ConeArray object.

**Synopsis**

```
ConeArray()
```

#### ConeArray.GetCone()

Get idx-th cone object.

**Synopsis**

```
Cone GetCone(int idx)
```

**Arguments**

`idx`: index of cone.

**Return**

cone object with index idx.

#### ConeArray.PushBack()

Add a cone constraint object to cone constraint array.

**Synopsis**

```
void PushBack(Cone cone)
```

**Arguments**

`cone`: a cone constraint object.

#### ConeArray.Size()

Get the number of cone constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of cone constraint objects.

### 26.2.23 ConeBuilder

COPT cone constraint builder object. To help building a cone constraint, given the cone type and a set of variables, Cardinal Optimizer provides C# ConeBuilder class, which defines the following methods.

#### ConeBuilder.ConeBuilder()

Constructor of ConeBuilder object.

**Synopsis**

```
ConeBuilder()
```

#### ConeBuilder.GetSize()

Get number of variables in a cone constraint.

**Synopsis**

```
int GetSize()
```

**Return**

number of variables.

#### ConeBuilder.GetType()

Get type of a cone constraint.

**Synopsis**

```
int GetType()
```

**Return**

type of the cone constraint.

#### ConeBuilder.GetVar()

Get i-th variable in a cone constraint.

**Synopsis**

```
Var GetVar(int idx)
```

**Arguments**

`idx`: index of vars.

**Return**

the i-th variable in a cone constraint.

#### ConeBuilder.GetVars()

Get all variables in a cone constraint.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variables in a cone constraint.

#### ConeBuilder.Set()

Set variables and type of a cone constraint.

**Synopsis**

```
void Set(VarArray vars, int type)
```

**Arguments**

`vars`: variable array object.

`type`: type of a cone constraint.

### 26.2.24 ConeBuilderArray

COPT cone constraint builder array object. To store and access a set of C# *ConeBuilder* objects, Cardinal Optimizer provides C# ConeBuilderArray class, which defines the following methods.

#### ConeBuilderArray.ConeBuilderArray()

Constructor of ConeBuilderArray object.

**Synopsis**

```
ConeBuilderArray()
```

#### ConeBuilderArray.GetBuilder()

Get idx-th cone constraint builder object.

**Synopsis**

```
ConeBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the cone constraint builder.

**Return**

cone constraint builder object with index idx.

#### ConeBuilderArray.PushBack()

Add a cone constraint builder object to cone constraint builder array.

**Synopsis**

```
void PushBack(ConeBuilder builder)
```

**Arguments**

`builder`: a cone constraint builder object.

#### ConeBuilderArray.Size()

Get the number of cone constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of cone constraint builder objects.

### 26.2.25 ExpCone

COPT exponential cone constraint object. Exponential Cone constraints are always associated with a particular model. User creates an exponential cone constraint object by adding an exponential cone constraint to a model, rather than by using constructor of ExpCone class.

#### ExpCone.GetIdx()

Get the index of an exponential cone constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of the exponential cone constraint.

#### ExpCone.Remove()

Remove the exponential cone constraint from model.

**Synopsis**

```
void Remove()
```

### 26.2.26 ExpConeArray

COPT exponential cone constraint array object. To store and access a set of C# *ExpCone* objects, Cardinal Optimizer provides C# ExpConeArray class, which defines the following methods.

#### ExpConeArray.ExpConeArray()

Constructor of ExpConeArray object.

**Synopsis**

```
ExpConeArray()
```

#### ExpConeArray.GetCone()

Get idx-th exponential cone object.

**Synopsis**

```
ExpCone GetCone(int idx)
```

**Arguments**

`idx`: index of exponential cone.

**Return**

exponential cone object with index idx.

#### ExpConeArray.PushBack()

Add an exponential cone constraint object to exponential cone constraint array.

**Synopsis**

```
void PushBack(ExpCone cone)
```

**Arguments**

`cone`: an exponential cone constraint object.

#### ExpConeArray.Size()

Get the number of exponential cone constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of exponential cone constraint objects.

### 26.2.27 ExpConeBuilder

COPT exponential cone constraint builder object. To help building an exponential cone constraint, given the exponential cone type and a set of variables, Cardinal Optimizer provides C# ExpConeBuilder class, which defines the following methods.

#### ExpConeBuilder.ExpConeBuilder()

Constructor of ExpConeBuilder object.

**Synopsis**

```
ExpConeBuilder()
```

#### ExpConeBuilder.GetSize()

Get number of variables in an exponential cone constraint.

**Synopsis**

```
int GetSize()
```

**Return**

number of variables.

#### ExpConeBuilder.GetType()

Get type of an exponential cone constraint.

**Synopsis**

```
int GetType()
```

**Return**

type of the exponential cone constraint.

#### ExpConeBuilder.GetVar()

Get i-th variable in an exponential cone constraint.

**Synopsis**

```
Var GetVar(int idx)
```

**Arguments**

`idx`: index of vars.

**Return**

the i-th variable in an exponential cone constraint.

#### ExpConeBuilder.GetVars()

Get all variables in an exponential cone constraint.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variables in an exponential cone constraint.

#### ExpConeBuilder.Set()

Set variables and type of an exponential cone constraint.

**Synopsis**

```
void Set(VarArray vars, int type)
```

**Arguments**

`vars`: variable array object.

`type`: type of an exponential cone constraint.

### 26.2.28 ExpConeBuilderArray

COPT exponential cone constraint builder array object. To store and access a set of C# *ExpConeBuilder* objects, Cardinal Optimizer provides C# ExpConeBuilderArray class, which defines the following methods.

#### ExpConeBuilderArray.ExpConeBuilderArray()

Constructor of ExpConeBuilderArray object.

**Synopsis**

```
ExpConeBuilderArray()
```

#### ExpConeBuilderArray.GetBuilder()

Get idx-th exponential cone constraint builder object.

**Synopsis**

```
ExpConeBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the exponential cone constraint builder.

**Return**

exponential cone constraint builder object with index idx.

#### ExpConeBuilderArray.PushBack()

Add an exponential cone constraint builder object to exponential cone constraint builder array.

**Synopsis**

```
void PushBack(ExpConeBuilder builder)
```

**Arguments**

`builder`: an exponential cone constraint builder object.

#### ExpConeBuilderArray.Size()

Get the number of exponential cone constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of exponential cone constraint builder objects.

### 26.2.29 AffineCone

The *AffineCone* class in COPT encapsulates operations related to affine cones. The following methods are provided:

#### AffineCone.GetIdx()

Get the index of an affine cone constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of the affine cone constraint.

#### AffineCone.GetName()

Get name of the affine cone.

**Synopsis**

```
string GetName()
```

**Return**

affine cone name.

#### AffineCone.Remove()

Remove the affine cone constraint from model.

**Synopsis**

```
void Remove()
```

#### AffineCone.SetName()

Set name of the affine cone.

**Synopsis**

```
void SetName(string name)
```

**Arguments**

`name`: affinecone name.

### 26.2.30 AffineConeArray

To facilitate operations on a group of C# *AffineCone* objects, the C# interface of COPT introduces the *AffineConeArray* class. The following methods are provided:

#### AffineConeArray.AffineConeArray()

Constructor of AffineConeArray object.

**Synopsis**

```
AffineConeArray()
```

#### AffineConeArray.GetCone()

Get idx-th affine cone object.

**Synopsis**

```
AffineCone GetCone(int idx)
```

**Arguments**

`idx`: index of affine cone.

**Return**

affine cone object with index idx.

#### AffineConeArray.PushBack()

Add an affine cone constraint object to affine cone constraint array.

**Synopsis**

```
void PushBack(AffineCone cone)
```

**Arguments**

`cone`: an affine cone constraint object.

#### AffineConeArray.Size()

Get the number of affine cone constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of affine cone constraint objects.

### 26.2.31 AffineConeBuilder

The *AffineConeBuilder* class in COPT encapsulates the builder for constructing affine cones. The following methods are provided:

#### AffineConeBuilder.AffineConeBuilder()

Constructor of AffineConeBuilder object.

**Synopsis**

```
AffineConeBuilder()
```

#### AffineConeBuilder.GetExpr()

Get i-th linear expression in an affine cone constraint.

**Synopsis**

```
Expr GetExpr(int idx)
```

**Arguments**

`idx`: index of linear expression.

**Return**

the i-th linear expression in an affine cone constraint.

#### AffineConeBuilder.GetExprs()

Get all linear expressions in an affine cone constraint.

**Synopsis**

```
Expr[] GetExprs()
```

**Return**

array of linear expressions.

### AffineConeBuilder.GetPsdExpr()

Get idx-th PSD expression in an affine cone constraint.

**Synopsis**

```
PsdExpr GetPsdExpr(int idx)
```

**Arguments**

`idx`: index of PSD expression.

**Return**

the idx-th PSD expression in an affine cone constraint.

### AffineConeBuilder.GetPsdExprs()

Get all PSD expressions in an affine cone constraint.

**Synopsis**

```
PsdExpr[] GetPsdExprs()
```

**Return**

array of PSD expressions.

### AffineConeBuilder.GetSize()

Get number of variables in an affine cone constraint.

**Synopsis**

```
int GetSize()
```

**Return**

number of variables.

### AffineConeBuilder.GetType()

Get type of an affine cone constraint.

**Synopsis**

```
int GetType()
```

**Return**

type of the affine cone constraint.

### AffineConeBuilder.HasPsdTerm()

Check whether affine cone has PSD terms.

**Synopsis**

```
bool HasPsdTerm()
```

**Return**

flag to indicate whether affine cone has PSD terms.

#### AffineConeBuilder.Set()

Set linear expressions and type of an affine cone constraint.

**Synopsis**

```
void Set(Expr[] exprs, int type)
```

**Arguments**

`exprs`: array of linear expressions.

`type`: type of an affine cone constraint.

#### AffineConeBuilder.Set()

Set PSD expressions and type of an affine cone constraint.

**Synopsis**

```
void Set(PsdExpr[] exprs, int type)
```

**Arguments**

`exprs`: array of PSD expressions.

`type`: type of an affine cone constraint.

### 26.2.32 AffineConeBuilderArray

To facilitate operations on a group of C# *AffineConeBuilder* objects, the C# interface of COPT introduces the *AffineConeBuilderArray* class. The following methods are provided:

#### AffineConeBuilderArray.AffineConeBuilderArray()

Constructor of AffineConeBuilderArray object.

**Synopsis**

```
AffineConeBuilderArray()
```

#### AffineConeBuilderArray.GetBuilder()

Get idx-th affine cone constraint builder object.

**Synopsis**

```
AffineConeBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the affine cone constraint builder.

**Return**

affine cone constraint builder object with index idx.

#### AffineConeBuilderArray.PushBack()

Add an affine cone constraint builder object to affine cone constraint builder array.

**Synopsis**

```
void PushBack(AffineConeBuilder builder)
```

**Arguments**

`builder`: an affine cone constraint builder object.

#### AffineConeBuilderArray.Size()

Get the number of affine cone constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of affine cone constraint builder objects.

### 26.2.33 QuadExpr

COPT quadratic expression object. A quadratic expression consists of a linear expression, a list of variable pairs and associated coefficients of quadratic terms. Quadratic expressions are used to build quadratic constraints and objectives.

#### QuadExpr.QuadExpr()

Constructor of a quadratic expression with default constant value 0.

**Synopsis**

```
QuadExpr(double constant)
```

**Arguments**

`constant`: optional, constant value in quadratic expression object.

#### QuadExpr.QuadExpr()

Constructor of a quadratic expression with one linear term.

**Synopsis**

```
QuadExpr(Var var, double coeff)
```

**Arguments**

`var`: variable of the added linear term.

`coeff`: coefficient for the added linear term with default value 1.0.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with a linear expression.

**Synopsis**

```
QuadExpr(Expr expr)
```

**Arguments**

`expr`: linear expression added to the quadratic expression.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

**Synopsis**

```
QuadExpr(Expr expr, Var var)
```

**Arguments**

`expr`: one linear expression.

`var`: another variable.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

**Synopsis**

```
QuadExpr(Expr left, Expr right)
```

**Arguments**

`left`: one linear expression.

`right`: another linear expression.

### QuadExpr.AddConstant()

Add a constant to the quadratic expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: value to be added.

### QuadExpr.AddLinExpr()

Add a linear expression to self.

**Synopsis**

```
void AddLinExpr(Expr expr)
```

**Arguments**

`expr`: linear expression to be added.

### QuadExpr.AddLinExpr()

Add a linear expression to self.

**Synopsis**

```
void AddLinExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: multiplier constant.

### QuadExpr.AddQuadExpr()

Add a quadratic expression to self.

**Synopsis**

```
void AddQuadExpr(QuadExpr expr)
```

**Arguments**

`expr`: quadratic expression to be added.

### QuadExpr.AddQuadExpr()

Add a quadratic expression to self.

**Synopsis**

```
void AddQuadExpr(QuadExpr expr, double mult)
```

**Arguments**

`expr`: quadratic expression to be added.

`mult`: multiplier constant.

### QuadExpr.AddTerm()

Add a term to quadratic expression object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: a variable of new term.

`coeff`: coefficient of new term.

### QuadExpr.AddTerm()

Add a quadratic term to expression object.

**Synopsis**

```
void AddTerm(
    Var var1,
    Var var2,
    double coeff)
```

**Arguments**

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

### QuadExpr.AddTerms()

Add linear terms to quadratic expression object.

**Synopsis**

```
void AddTerms(Var[] vars, double coeff)
```

**Arguments**

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

### QuadExpr.AddTerms()

Add linear terms to quadratic expression object.

**Synopsis**

```
void AddTerms(Var[] vars, double[] coeffs)
```

**Arguments**

`vars`: variables of added linear terms.

`coeffs`: coefficients of added linear terms.

### QuadExpr.AddTerms()

Add linear terms to quadratic expression object.

**Synopsis**

```
void AddTerms(VarArray vars, double coeff)
```

**Arguments**

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

### QuadExpr.AddTerms()

Add linear terms to quadratic expression object.

**Synopsis**

```
void AddTerms(VarArray vars, double[] coeffs)
```

**Arguments**

`vars`: variables of added terms.

`coeffs`: coefficients of added terms.

### QuadExpr.AddTerms()

Add quadratic terms to expression object.

**Synopsis**

```
void AddTerms(
    VarArray vars1,
    VarArray vars2,
    double[] coeffs)
```

**Arguments**

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`coeffs`: coefficient array for added quadratic terms.

### QuadExpr.AddTerms()

Add quadratic terms to expression object.

**Synopsis**

```
void AddTerms(
    Var[] vars1,
    Var[] vars2,
    double[] coeffs)
```

**Arguments**

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`coeffs`: coefficient array for added quadratic terms.

### QuadExpr.Clone()

Deep copy quadratic expression object.

**Synopsis**

```
QuadExpr Clone()
```

**Return**

cloned quadratic expression object.

### QuadExpr.Divide()

Divide itself by a constant.

**Synopsis**

```
void Divide(double c)
```

**Arguments**

`c`: constant operand.

### QuadExpr.Evaluate()

evaluate quadratic expression after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of quadratic expression.

### QuadExpr.GetCoeff()

Get coefficient from the i-th term in quadratic expression.

**Synopsis**

```
double GetCoeff(int i)
```

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term in quadratic expression object.

### QuadExpr.GetConstant()

Get constant in quadratic expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in quadratic expression.

### QuadExpr.GetLinExpr()

Get linear expression in quadratic expression.

**Synopsis**

```
Expr GetLinExpr()
```

**Return**

linear expression object.

### QuadExpr.GetVar1()

Get first variable from the i-th term in quadratic expression.

**Synopsis**

```
Var GetVar1(int i)
```

**Arguments**

`i`: index of the term.

**Return**

first variable of the i-th term in quadratic expression object.

### QuadExpr.GetVar2()

Get second variable from the i-th term in quadratic expression.

**Synopsis**

```
Var GetVar2(int i)
```

**Arguments**

`i`: index of the term.

**Return**

second variable of the i-th term in quadratic expression object.

### QuadExpr.Multiply()

Multiply itself by a constant.

**Synopsis**

```
void Multiply(double c)
```

**Arguments**

`c`: constant operand.

### QuadExpr.Remove()

Remove idx-th term from quadratic expression object.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### QuadExpr.Remove()

Remove the term associated with variable from quadratic expression.

**Synopsis**

```
void Remove(Var var)
```

**Arguments**

`var`: a variable whose term should be removed.

### QuadExpr.SetCoeff()

Set coefficient of the i-th term in quadratic expression.

**Synopsis**

```
void SetCoeff(int i, double val)
```

**Arguments**

`i`: index of the quadratic term.

`val`: coefficient of the term.

#### QuadExpr.SetConstant()

Set constant for the quadratic expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### QuadExpr.Size()

Get number of terms in quadratic expression.

**Synopsis**

```
long Size()
```

**Return**

number of quadratic terms.

### 26.2.34 QConstraint

COPT quadratic constraint object. Quadratic constraints are always associated with a particular model. User creates a quadratic constraint object by adding a quadratic constraint to a model, rather than by using constructor of QConstraint class.

#### QConstraint.Get()

Get information value of the quadratic constraint.

**Synopsis**

```
double Get(string info)
```

**Arguments**

`info`: name of the information being queried.

**Return**

information value.

#### QConstraint.GetIdx()

Get index of the quadratic constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

the index of the quadratic constraint.

### QConstraint.GetName()

Get name of the quadratic constraint.

**Synopsis**

```
string GetName()
```

**Return**

the name of the quadratic constraint.

### QConstraint.GetRhs()

Get rhs of quadratic constraint.

**Synopsis**

```
double GetRhs()
```

**Return**

rhs of quadratic constraint.

### QConstraint.GetSense()

Get rhs of quadratic constraint.

**Synopsis**

```
char GetSense()
```

**Return**

rhs of quadratic constraint.

### QConstraint.Remove()

Remove this constraint from model.

**Synopsis**

```
void Remove()
```

### QConstraint.Set()

Set information value of the quadratic constraint.

**Synopsis**

```
void Set(string attr, double val)
```

**Arguments**

`attr`: name of the information.

`val`: new information value.

#### QConstraint.SetName()

Set name of quadratic constraint.

**Synopsis**

```
void SetName(string name)
```

**Arguments**

`name`: the name to set.

#### QConstraint.SetRhs()

Set rhs of quadratic constraint.

**Synopsis**

```
void SetRhs(double rhs)
```

**Arguments**

`rhs`: rhs of quadratic constraint.

#### QConstraint.SetSense()

Set sense of quadratic constraint.

**Synopsis**

```
void SetSense(char sense)
```

**Arguments**

`sense`: sense of quadratic constraint.

### 26.2.35 QConstrArray

COPT quadratic constraint array object. To store and access a set of C# *QConstraint* objects, Cardinal Optimizer provides C# QConstrArray class, which defines the following methods.

#### QConstrArray.QConstrArray()

Constructor of qconstrarray object.

**Synopsis**

```
QConstrArray()
```

#### QConstrArray.GetQConstr()

Get idx-th quadratic constraint object.

**Synopsis**

```
QConstraint GetQConstr(int idx)
```

**Arguments**

`idx`: index of the quadratic constraint.

**Return**

constraint object with index 'idx'.

#### QConstrArray.PushBack()

Add a quadratic constraint object to array.

**Synopsis**

```
void PushBack(QConstraint constr)
```

**Arguments**

`constr`: a quadratic constraint object.

#### QConstrArray.Size()

Get the number of quadratic constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of quadratic constraint objects.

### 26.2.36 QConstrBuilder

COPT quadratic constraint builder object. To help building a quadratic constraint, given a quadratic expression, constraint sense and right-hand side value, Cardinal Optimizer provides C# ConeBuilder class, which defines the following methods.

#### QConstrBuilder.QConstrBuilder()

Constructor of QConstrBuilder object.

**Synopsis**

```
QConstrBuilder()
```

#### QConstrBuilder.GetQuadExpr()

Get quadratic expression associated with constraint.

**Synopsis**

```
QuadExpr GetQuadExpr()
```

**Return**

quadratic expression object.

#### QConstrBuilder.GetSense()

Get sense associated with quadratic constraint.

**Synopsis**

```
char GetSense()
```

**Return**

quadratic constraint sense.

#### QConstrBuilder.Set()

Set detail of a quadratic constraint to its builder object.

**Synopsis**

```
void Set(
    QuadExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the quadratic constraint.

`sense`: quadratic constraint sense.

`rhs`: constant of right side of quadratic constraint.

### 26.2.37 QConstrBuilderArray

COPT quadratic constraint builder array object. To store and access a set of C# *QConstrBuilder* objects, Cardinal Optimizer provides C# QConstrBuilderArray class, which defines the following methods.

#### QConstrBuilderArray.QConstrBuilderArray()

Constructor of constrbuilderarray object.

**Synopsis**

```
QConstrBuilderArray()
```

#### QConstrBuilderArray.GetBuilder()

Get idx-th quadratic constraint builder object.

**Synopsis**

```
QConstrBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the quadratic constraint builder.

**Return**

constraint builder object with index 'idx'.

#### QConstrBuilderArray.PushBack()

Add a quadratic constraint builder object to constraint builder array.

**Synopsis**

```
void PushBack(QConstrBuilder builder)
```

**Arguments**

`builder`: a quadratic constraint builder object.

#### QConstrBuilderArray.Size()

Get the number of quadratic constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of quadratic constraint builder objects.

### 26.2.38 PsdVar

COPT PSD variable object. PSD variables are always associated with a particular model. User creates a PSD variable object by adding a PSD variable to model, rather than by constructor of PsdVar class.

#### PsdVar.Diagonal()

Get diagonals of PSD variable.

**Synopsis**

```
MPsdExpr Diagonal(int offset)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

**Return**

one-dimensional MPsdExpr object of diagonals.

#### PsdVar.Get()

Get information values of PSD variable.

**Synopsis**

```
double[] Get(string info)
```

**Arguments**

`info`: name of information.

**Return**

array of information values.

#### PsdVar.GetDim()

Get dimension of PSD variable.

**Synopsis**

```
int GetDim()
```

**Return**

dimension of PSD variable.

### PsdVar.GetIdx()

Get index of PSD variable.

**Synopsis**

```
int GetIdx()
```

**Return**

index of PSD variable.

### PsdVar.GetInfo()

Get information values of PSD variable.

**Synopsis**

```
NdArray<double> GetInfo(string name)
```

**Arguments**

`name`: name of information.

**Return**

2-dimensional NdArray with related information values.

### PsdVar.GetItem()

Get corresponding MPsdExpr of given index from PSD variable.

**Synopsis**

```
PsdExpr GetItem(int i, int j)
```

**Arguments**

`i`: row index.

`j`: col index.

**Return**

new PsdExpr object.

### PsdVar.GetLen()

Get length of PSD variable.

**Synopsis**

```
int GetLen()
```

**Return**

length of PSD variable.

### PsdVar.GetName()

Get name of PSD variable.

**Synopsis**

```
string GetName()
```

**Return**

name of PSD variable.

### PsdVar.GetSize()

Get size or length of PSD variable.

**Synopsis**

```
int GetSize()
```

**Return**

Size of PSD variable.

### PsdVar.Pick()

Given a list of indexes, pick elements of PSD variable in format of MPsdExpr.

**Synopsis**

```
MPsdExpr Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. If two dimensional, each row is position of an element.

**Return**

one-dimensional MPsdExpr object.

### PsdVar.Remove()

Remove PSD variable from model.

**Synopsis**

```
void Remove()
```

### PsdVar.Slice()

Get corresponding MPsdExpr of given view from PSD variable.

**Synopsis**

```
MPsdExpr Slice(View view)
```

**Arguments**

`view`: view of PSD variable.

**Return**

new MPsdExpr object.

#### PsdVar.Sum()

Sum of elements in PSD variable.

**Synopsis**

```
PsdExpr Sum()
```

**Return**

PsdExpr object.

#### PsdVar.ToExpr()

convert PSD variable to 2-dimensional MPsdExpr.

**Synopsis**

```
MPsdExpr ToExpr()
```

**Return**

2-dimensional MPsdExpr object.

### 26.2.39 PsdVarArray

COPT PSD variable array object. To store and access a set of *PsdVar* objects, Cardinal Optimizer provides PsdVarArray class, which defines the following methods.

#### PsdVarArray.PsdVarArray()

Constructor of PsdVarArray.

**Synopsis**

```
PsdVarArray()
```

#### PsdVarArray.GetPsdVar()

Get idx-th PSD variable object.

**Synopsis**

```
PsdVar GetPsdVar(int idx)
```

**Arguments**

`idx`: index of the PSD variable.

**Return**

PSD variable object with index idx.

#### PsdVarArray.PushBack()

Add a PSD variable object to PSD variable array.

**Synopsis**

```
void PushBack(PsdVar var)
```

**Arguments**

`var`: a PSD variable object.

#### PsdVarArray.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD variable object.

#### PsdVarArray.Size()

Get the number of PSD variable objects.

**Synopsis**

```
int Size()
```

**Return**

number of PSD variable objects.

### 26.2.40 PsdExpr

COPT PSD expression object. A PSD expression consists of a linear expression, a list of PSD variables and associated coefficient matrices of PSD terms. PSD expressions are used to build PSD constraints and objectives.

#### PsdExpr.PsdExpr()

Constructor of a PSD expression with default constant value 0.

**Synopsis**

```
PsdExpr(double constant)
```

**Arguments**

`constant`: optional, constant value in PSD expression object.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(Var var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with a linear expression.

**Synopsis**

```
PsdExpr(Expr expr)
```

**Arguments**

`expr`: input linear expression.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(PsdVar var, SymMatrix mat)
```

**Arguments**

`var`: PSD variable for the added term.

`mat`: coefficient matrix for the added term.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(PsdVar var, SymMatExpr expr)
```

**Arguments**

`var`: PSD variable for the added term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

### PsdExpr.AddConstant()

Add constant to the PSD expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: value to be added.

### PsdExpr.AddLinExpr()

Add a linear expression to PSD expression object.

**Synopsis**

```
void AddLinExpr(Expr expr)
```

**Arguments**

`expr`: linear expression to be added.

### PsdExpr.AddLinExpr()

Add a linear expression to PSD expression object.

**Synopsis**

```
void AddLinExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: multiplier constant.

### PsdExpr.AddPsdExpr()

Add a PSD expression to self.

**Synopsis**

```
void AddPsdExpr(PsdExpr expr)
```

**Arguments**

`expr`: PSD expression to be added.

### PsdExpr.AddPsdExpr()

Add a PSD expression to self.

**Synopsis**

```
void AddPsdExpr(PsdExpr expr, double mult)
```

**Arguments**

`expr`: PSD expression to be added.

`mult`: multiplier constant.

### PsdExpr.AddTerm()

Add a linear term to PSD expression object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

### PsdExpr.AddTerm()

Add a PSD term to PSD expression object.

**Synopsis**

```
void AddTerm(PsdVar var, SymMatrix mat)
```

**Arguments**

`var`: PSD variable of new PSD term.

`mat`: coefficient matrix of new PSD term.

### PsdExpr.AddTerm()

Add a PSD term to PSD expression object.

**Synopsis**

```
void AddTerm(PsdVar var, SymMatExpr expr)
```

**Arguments**

`var`: PSD variable of new PSD term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

### PsdExpr.AddTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
void AddTerms(Var[] vars, double coeff)
```

**Arguments**

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

### PsdExpr.AddTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
void AddTerms(Var[] vars, double[] coeffs)
```

**Arguments**

`vars`: variables for added linear terms.

`coeffs`: coefficient array for added linear terms.

### PsdExpr.AddTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
void AddTerms(VarArray vars, double coeff)
```

**Arguments**

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

### PsdExpr.AddTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
void AddTerms(VarArray vars, double[] coeffs)
```

**Arguments**

`vars`: variables of added terms.

`coeffs`: coefficients of added terms.

### PsdExpr.AddTerms()

Add PSD terms to PSD expression object.

**Synopsis**

```
void AddTerms(PsdVarArray vars, SymMatrixArray mats)
```

**Arguments**

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrices for added PSD terms.

### PsdExpr.AddTerms()

Add PSD terms to PSD expression object.

**Synopsis**

```
void AddTerms(PsdVar[] vars, SymMatrix[] mats)
```

**Arguments**

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrices for added PSD terms.

### PsdExpr.Clone()

Deep copy PSD expression object.

**Synopsis**

```
PsdExpr Clone()
```

**Return**

cloned PSD expression object.

### PsdExpr.Evaluate()

Evaluate PSD expression after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of PSD expression.

### PsdExpr.GetCoeff()

Get coefficient from the i-th term in PSD expression.

**Synopsis**

```
SymMatExpr GetCoeff(int i)
```

**Arguments**

`i`: index of the PSD term.

**Return**

coefficient expression of the i-th PSD term.

### PsdExpr.GetConstant()

Get constant in PSD expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in PSD expression.

### PsdExpr.GetLinExpr()

Get linear expression in PSD expression.

**Synopsis**

```
Expr GetLinExpr()
```

**Return**

linear expression object.

### PsdExpr.GetPsdVar()

Get the PSD variable from the i-th term in PSD expression.

**Synopsis**

```
PsdVar GetPsdVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

the first variable of the i-th term in PSD expression object.

### PsdExpr.Multiply()

Multiply itself by a constant.

**Synopsis**

```
void Multiply(double c)
```

**Arguments**

`c`: constant operand.

### PsdExpr.Remove()

Remove i-th term from PSD expression object.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### PsdExpr.Remove()

Remove the term associated with variable from PSD expression.

**Synopsis**

```
void Remove(Var var)
```

**Arguments**

`var`: a variable whose term should be removed.

### PsdExpr.Remove()

Remove the term associated with PSD variable from PSD expression.

**Synopsis**

```
void Remove(PsdVar var)
```

**Arguments**

`var`: a PSD variable whose term should be removed.

#### PsdExpr.SetCoeff()

Set coefficient matrix of the i-th term in PSD expression.

**Synopsis**

```
void SetCoeff(int i, SymMatrix mat)
```

**Arguments**

`i`: index of the PSD term.

`mat`: coefficient matrix of the term.

#### PsdExpr.SetConstant()

Set constant for the PSD expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### PsdExpr.Size()

Get number of PSD terms in expression.

**Synopsis**

```
long Size()
```

**Return**

number of PSD terms.

### 26.2.41 PsdConstraint

COPT PSD constraint object. PSD constraints are always associated with a particular model. User creates a PSD constraint object by adding a PSD constraint to model, rather than by constructor of PsdConstraint class.

#### PsdConstraint.Get()

Get information value of the PSD constraint. Support related PSD information.

**Synopsis**

```
double Get(string info)
```

**Arguments**

`info`: name of queried information.

**Return**

information value.

### PsdConstraint.GetIdx()

Get index of the PSD constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

the index of the PSD constraint.

### PsdConstraint.GetName()

Get name of the PSD constraint.

**Synopsis**

```
string GetName()
```

**Return**

the name of the PSD constraint.

### PsdConstraint.Remove()

Remove this PSD constraint from model.

**Synopsis**

```
void Remove()
```

### PsdConstraint.Set()

Set information value of the PSD constraint. Support related PSD information.

**Synopsis**

```
void Set(string info, double value)
```

**Arguments**

`info`: name of queried information.

`value`: new information value.

### PsdConstraint.SetName()

Set name of a PSD constraint.

**Synopsis**

```
void SetName(string name)
```

**Arguments**

`name`: the name to set.

### 26.2.42 PsdConstrArray

COPT PSD constraint array object. To store and access a set of *PsdConstraint* objects, Cardinal Optimizer provides PsdConstrArray class, which defines the following methods.

#### PsdConstrArray.PsdConstrArray()

Constructor of PsdConstrArray object.

**Synopsis**

```
PsdConstrArray()
```

#### PsdConstrArray.GetPsdConstr()

Get idx-th PSD constraint object.

**Synopsis**

```
PsdConstraint GetPsdConstr(int idx)
```

**Arguments**

`idx`: index of the PSD constraint.

**Return**

PSD constraint object with index idx.

#### PsdConstrArray.PushBack()

Add a PSD constraint object to PSD constraint array.

**Synopsis**

```
void PushBack(PsdConstraint constr)
```

**Arguments**

`constr`: a PSD constraint object.

#### PsdConstrArray.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD constraint objects.

#### PsdConstrArray.Size()

Get the number of PSD constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of PSD constraint objects.

### 26.2.43 PsdConstrBuilder

COPT PSD constraint builder object. To help building a PSD constraint, given a PSD expression, constraint sense and right-hand side value, Cardinal Optimizer provides PsdConstrBuilder class, which defines the following methods.

#### PsdConstrBuilder.PsdConstrBuilder()

Constructor of PsdConstrBuilder object.

**Synopsis**

```
PsdConstrBuilder()
```

#### PsdConstrBuilder.GetPsdExpr()

Get expression associated with PSD constraint.

**Synopsis**

```
PsdExpr GetPsdExpr()
```

**Return**

PSD expression object.

#### PsdConstrBuilder.GetRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of the constraint.

#### PsdConstrBuilder.GetSense()

Get sense associated with PSD constraint.

**Synopsis**

```
char GetSense()
```

**Return**

PSD constraint sense.

#### PsdConstrBuilder.Set()

Set detail of a PSD constraint to its builder object.

**Synopsis**

```
void Set(
    PsdExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the PSD constraint.

`sense`: PSD constraint sense, other than COPT_RANGE.

`rhs`: constant at right side of the PSD constraint.

#### PsdConstrBuilder.SetRange()

Set a range constraint to its builder.

**Synopsis**

```
void SetRange(PsdExpr expr, double range)
```

**Arguments**

`expr`: PSD expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must be greater than 0.

### 26.2.44 PsdConstrBuilderArray

COPT PSD constraint builder array object. To store and access a set of *PsdConstrBuilder* objects, Cardinal Optimizer provides PsdConstrBuilderArray class, which defines the following methods.

#### PsdConstrBuilderArray.PsdConstrBuilderArray()

Constructor of PsdConstrBuilderArray object.

**Synopsis**

```
PsdConstrBuilderArray()
```

#### PsdConstrBuilderArray.GetBuilder()

Get idx-th PSD constraint builder object.

**Synopsis**

```
PsdConstrBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the PSD constraint builder.

**Return**

PSD constraint builder object with index idx.

#### PsdConstrBuilderArray.PushBack()

Add a PSD constraint builder object to PSD constraint builder array.

**Synopsis**

```
void PushBack(PsdConstrBuilder builder)
```

**Arguments**

`builder`: a PSD constraint builder object.

#### PsdConstrBuilderArray.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD constraint builder object.

#### PsdConstrBuilderArray.Size()

Get the number of PSD constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of PSD constraint builder objects.

### 26.2.45 LmiConstraint

COPT LMI constraint object. LMI constraints are always associated with a particular model. User creates a LMI constraint object by adding a LMI constraint to model, rather than by constructor of LmiConstraint class.

#### LmiConstraint.Get()

Get information values of LMI constraint.

**Synopsis**

```
double[] Get(string info)
```

**Arguments**

`info`: name of information.

**Return**

output array of information values.

### LmiConstraint.GetDim()

Get dimension of LMI constraint.

**Synopsis**

```
int GetDim()
```

**Return**

dimension of LMI constraint.

### LmiConstraint.GetIdx()

Get index of LMI constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

index of LMI constraint.

### LmiConstraint.GetLen()

Get length of LMI constraint.

**Synopsis**

```
int GetLen()
```

**Return**

length of LMI constraint.

### LmiConstraint.GetName()

Get name of LMI constraint.

**Synopsis**

```
string GetName()
```

**Return**

name of LMI constraint.

### LmiConstraint.Remove()

Remove this LMI constraint from model.

**Synopsis**

```
void Remove()
```

#### LmiConstraint.SetRhs()

Set constant term of LMI constraint.

**Synopsis**

```
void SetRhs(SymMatrix mat)
```

**Arguments**

`mat`: new symmetric matrix for constant term.

### 26.2.46 LmiConstrArray

COPT LMI constraint array object. To store and access a set of *LmiConstraint* objects, Cardinal Optimizer provides LmiConstrArray class, which defines the following methods.

#### LmiConstrArray.LmiConstrArray()

Constructor of LmiConstrArray.

**Synopsis**

```
LmiConstrArray()
```

#### LmiConstrArray.GetLmiConstr()

Get idx-th LMI constraint object.

**Synopsis**

```
LmiConstraint GetLmiConstr(int idx)
```

**Arguments**

`idx`: index of LMI constraint.

**Return**

LMI constraint object with index idx.

#### LmiConstrArray.PushBack()

Add an LMI constraint to LMI constraint array.

**Synopsis**

```
void PushBack(LmiConstraint constr)
```

**Arguments**

`constr`: LMI constraint object.

#### LmiConstrArray.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: capacity number of LMI constraint object.

#### LmiConstrArray.Size()

Get the number of LMI constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of LMI constraint objects.

### 26.2.47 LmiExpr

COPT LMI expression object. A LMI expression consists of a list of variables, associated coefficient matrices of LMI term, and constant matrices. LMI expressions are used to build LMI constraints.

#### LmiExpr.LmiExpr()

Default constructor of LMI expression

**Synopsis**

```
LmiExpr()
```

#### LmiExpr.LmiExpr()

Constructor of LMI expression with constant term.

**Synopsis**

```
LmiExpr(SymMatrix mat)
```

**Arguments**

`mat`: symmetric matrix object.

#### LmiExpr.LmiExpr()

Constructor of LMI expression with matrix expression.

**Synopsis**

```
LmiExpr(SymMatExpr expr)
```

**Arguments**

`expr`: symmetric matrix expression.

### LmiExpr.LmiExpr()

Constructor of LMI expression with one term.

**Synopsis**

```
LmiExpr(Var var, SymMatrix mat)
```

**Arguments**

`var`: variable of the added term.

`mat`: coefficient matrix of the added term.

### LmiExpr.LmiExpr()

Constructor of LMI expression with one term.

**Synopsis**

```
LmiExpr(Var var, SymMatExpr expr)
```

**Arguments**

`var`: variable of the added term.

`expr`: coefficient expression of symmetric matrices of new LMI term.

### LmiExpr.AddConstant()

Add to constant term of the LMI expression.

**Synopsis**

```
void AddConstant(SymMatExpr expr)
```

**Arguments**

`expr`: matrix expression added to the constant term.

### LmiExpr.AddLmiExpr()

Add an LMI expression to self.

**Synopsis**

```
void AddLmiExpr(LmiExpr expr, double mult)
```

**Arguments**

`expr`: LMI expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### LmiExpr.AddTerm()

Add a term to LMI expression object.

**Synopsis**

```
void AddTerm(Var var, SymMatrix mat)
```

**Arguments**

`var`: variable of new LMI term.

`mat`: coefficient matrix of new LMI term.

### LmiExpr.AddTerm()

Add a term to LMI expression object.

**Synopsis**

```
void AddTerm(Var var, SymMatExpr expr)
```

**Arguments**

`var`: variable of new LMI term.

`expr`: coefficient expression of symmetric matrices of new LMI term.

### LmiExpr.AddTerms()

Add LMI terms to LMI expression object.

**Synopsis**

```
void AddTerms(VarArray vars, SymMatrixArray mats)
```

**Arguments**

`vars`: variables for added LMI terms.

`mats`: coefficient matrices for added LMI terms.

### LmiExpr.AddTerms()

Add LMI terms to LMI expression object.

**Synopsis**

```
void AddTerms(Var[] vars, SymMatrix[] mats)
```

**Arguments**

`vars`: variables for added LMI terms.

`mats`: coefficient matrices for added LMI terms.

### LmiExpr.Clone()

Deep copy LMI expression.

**Synopsis**

```
LmiExpr Clone()
```

**Return**

cloned LMI expression object.

### LmiExpr.GetCoeff()

Get coefficient from the i-th term in LMI expression.

**Synopsis**

```
SymMatExpr GetCoeff(int i)
```

**Arguments**

`i`: index of the LMI term.

**Return**

coefficient expression of the i-th LMI term.

### LmiExpr.GetConstant()

Get constant term in LMI expression.

**Synopsis**

```
SymMatExpr GetConstant()
```

**Return**

symmetric matrix expression object.

### LmiExpr.GetVar()

Get variable from the i-th term in LMI expression.

**Synopsis**

```
Var GetVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

variable of the i-th term in LMI expression object.

### LmiExpr.Multiply()

Multiply itself by a constant.

**Synopsis**

```
void Multiply(double c)
```

**Arguments**

`c`: constant operand.

### LmiExpr.Remove()

Remove i-th term from LMI expression object.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

#### LmiExpr.Remove()

Remove the term associated with variable from LMI expression.

**Synopsis**

```
void Remove(Var var)
```

**Arguments**

`var`: a variable whose term should be removed.

#### LmiExpr.SetCoeff()

Set coefficient matrix of the i-th term in LMI expression.

**Synopsis**

```
void SetCoeff(int i, SymMatrix mat)
```

**Arguments**

`i`: index of the LMI term.

`mat`: coefficient matrix of the term.

#### LmiExpr.SetConstant()

Set constant term of the LMI expression.

**Synopsis**

```
void SetConstant(SymMatrix mat)
```

**Arguments**

`mat`: symmetric matrix of the constant term.

#### LmiExpr.Size()

Get number of LMI terms in expression.

**Synopsis**

```
long Size()
```

**Return**

number of LMI terms.

### 26.2.48 SymMatrix

COPT symmetric matrix object. Symmetric matrices are always associated with a particular model. User creates a symmetric matrix object by adding a symmetric matrix to model, rather than by constructor of SymMatrix class.

Symmetric matrices are used as coefficient matrices of PSD terms in PSD expressions, PSD constraints or PSD objectives.

#### SymMatrix.GetDim()

Get the dimension of a symmetric matrix.

**Synopsis**

```
int GetDim()
```

**Return**

dimension of a symmetric matrix.

#### SymMatrix.GetIdx()

Get the index of a symmetric matrix.

**Synopsis**

```
int GetIdx()
```

**Return**

index of a symmetric matrix.

### 26.2.49 SymMatrixArray

COPT symmetric matrix object. To store and access a set of *SymMatrix* objects, Cardinal Optimizer provides SymMatrixArray class, which defines the following methods.

#### SymMatrixArray.SymMatrixArray()

Constructor of SymMatrixAarray.

**Synopsis**

```
SymMatrixArray()
```

#### SymMatrixArray.GetMatrix()

Get i-th SymMatrix object.

**Synopsis**

```
SymMatrix GetMatrix(int idx)
```

**Arguments**

`idx`: index of the SymMatrix object.

**Return**

SymMatrix object with index idx.

#### SymMatrixArray.PushBack()

Add a SymMatrix object to SymMatrix array.

**Synopsis**

```
void PushBack(SymMatrix mat)
```

**Arguments**

`mat`: a SymMatrix object.

#### SymMatrixArray.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for symmetric matrix object.

#### SymMatrixArray.Size()

Get the number of SymMatrix objects.

**Synopsis**

```
int Size()
```

**Return**

number of SymMatrix objects.

### 26.2.50 SymMatExpr

COPT symmetric matrix expression object. A symmetric matrix expression is a linear combination of symmetric matrices, which is still a symmetric matrix. However, by doing so, we are able to delay computing the final matrix until setting PSD constraints or PSD objective.

#### SymMatExpr.SymMatExpr()

Constructor of a symmetric matrix expression.

**Synopsis**

```
SymMatExpr()
```

#### SymMatExpr.SymMatExpr()

Constructor of a symmetric matrix expression with one term.

**Synopsis**

```
SymMatExpr(SymMatrix mat, double coeff)
```

**Arguments**

`mat`: symmetric matrix of the added term.

`coeff`: optional, coefficient for the added term. Its default value is 1.0.

### SymMatExpr.AddSymMatExpr()

Add a symmetric matrix expression to self.

**Synopsis**

```
void AddSymMatExpr(SymMatExpr expr, double mult)
```

**Arguments**

`expr`: symmetric matrix expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### SymMatExpr.AddTerm()

Add a term to symmetric matrix expression object.

**Synopsis**

```
bool AddTerm(SymMatrix mat, double coeff)
```

**Arguments**

`mat`: symmetric matrix of the new term.

`coeff`: coefficient of the new term.

**Return**

True if the term is added successfully.

### SymMatExpr.AddTerms()

Add multiple terms to expression object.

**Synopsis**

```
int AddTerms(SymMatrix[] mats, double coeff)
```

**Arguments**

`mats`: symmetric matrix array object for added terms.

`coeff`: optional, common coefficient for added terms, default is 1.0.

**Return**

Number of added terms. If negative, fail to add one of terms.

### SymMatExpr.AddTerms()

Add multiple terms to expression object.

**Synopsis**

```
int AddTerms(SymMatrixArray mats, double[] coeffs)
```

**Arguments**

`mats`: symmetric matrix array object for added terms.

`coeffs`: coefficient array for added terms.

**Return**

Number of added terms. If negative, fail to add one of terms.

### SymMatExpr.AddTerms()

Add multiple terms to expression object.

**Synopsis**

```
int AddTerms(SymMatrix[] mats, double[] coeffs)
```

**Arguments**

`mats`: symmetric matrix array object for added terms.

`coeffs`: coefficient array for added terms.

**Return**

Number of added terms. If negative, fail to add one of terms.

### SymMatExpr.Clone()

Deep copy symmetric matrix expression object.

**Synopsis**

```
SymMatExpr Clone()
```

**Return**

cloned expression object.

### SymMatExpr.GetCoeff()

Get coefficient of the i-th term in expression object.

**Synopsis**

```
double GetCoeff(int i)
```

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term.

### SymMatExpr.GetDim()

Get dimension of symmetric matrix in expression.

**Synopsis**

```
int GetDim()
```

**Return**

dimension of symmetric matrix.

### SymMatExpr.GetSymMat()

Get symmetric matrix of the i-th term in expression object.

**Synopsis**

```
SymMatrix GetSymMat(int i)
```

**Arguments**

`i`: index of the term.

**Return**

the symmetric matrix of the i-th term.

### SymMatExpr.Multiply()

Multiply itself by a constant.

**Synopsis**

```
void Multiply(double c)
```

**Arguments**

`c`: constant operand.

### SymMatExpr.Remove()

Remove i-th term from expression object.

**Synopsis**

```
void Remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### SymMatExpr.Remove()

Remove the term associated with the symmetric matrix.

**Synopsis**

```
void Remove(SymMatrix mat)
```

**Arguments**

`mat`: a symmetric matrix whose term should be removed.

### SymMatExpr.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: minimum capacity for expression object.

#### SymMatExpr.SetCoeff()

Set coefficient for the i-th term in expression object.

**Synopsis**

```
void SetCoeff(int i, double val)
```

**Arguments**

`i`: index of the term.

`val`: coefficient of the term.

#### SymMatExpr.Size()

Get number of terms in expression.

**Synopsis**

```
long Size()
```

**Return**

number of terms.

### 26.2.51 MVar

COPT multi-dimensional variable object. It is used to construct multi-dimensional variables and support operations with the built-in multi-dimensional array *NdArray* in COPT. It can be created by calling the method `addMVar` of *Model*. The following methods are provided:

#### MVar.Clone()

Clone MVar object.

**Synopsis**

```
MVar Clone()
```

**Return**

new MVar object.

#### MVar.Diagonal()

Get diagonals of MVar object.

**Synopsis**

```
MVar Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MVar.

`axis2`: 2nd axis of MVar.

**Return**

(N-1)-dimensional diagonals.

### MVar.Expand()

Expand shape of MVar object.

**Synopsis**

```
MVar Expand(int axis)
```

**Arguments**

`axis`: axis of MVar.

**Return**

MVar object of (N+1)-dimensional shape.

### MVar.Flatten()

Flatten a MVar object to a 1-dimensional shape.

**Synopsis**

```
MVar Flatten()
```

**Return**

a MVar object collapsed into one dimension.

### MVar.Get()

Get values of information associated with variables in MVar object.

**Synopsis**

```
NdArray<double> Get(string info)
```

**Arguments**

`info`: name of information.

**Return**

multi-dimensional array of information of variables.

### MVar.GetBasis()

Get basis of variables in MVar object.

**Synopsis**

```
NdArray<int> GetBasis()
```

**Return**

multi-dimensional array of basis of variables.

### MVar.GetDim()

Get i-th dimension of MVar object.

**Synopsis**

```
long GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MVar.GetIdx()

Get indexes of variables in MVar object.

**Synopsis**

```
NdArray<int> GetIdx()
```

**Return**

multi-dimensional array of indexes of variables.

### MVar.GetItem()

Get variable of given index from MVar object.

**Synopsis**

```
Var GetItem(long idx)
```

**Arguments**

`idx`: index of variable.

**Return**

Var object.

### MVar.GetItem()

Get sub-arrays of MVar object, given view object.

**Synopsis**

```
MVar GetItem(View view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MVar object.

### MVar.GetLowerIIS()

Get IIS status of lower bound of variables in MVar object.

**Synopsis**

```
NdArray<int> GetLowerIIS()
```

**Return**

multi-dimensional array of IIS status of lower bounds of variables.

### MVar.GetND()

Get number of dimensions of MVar object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MVar.GetShape()

Get shape of MVar object.

**Synopsis**

```
Shape GetShape()
```

**Return**

shape object.

### MVar.GetSize()

Get size of MVar object.

**Synopsis**

```
long GetSize()
```

**Return**

number of vars.

### MVar.GetType()

Get types of variables in MVar object.

**Synopsis**

```
NdArray<char> GetType()
```

**Return**

multi-dimensional array of types of variables.

### MVar.GetUpperIIS()

Get IIS status of upper bound of variables in MVar object.

**Synopsis**

```
NdArray<int> GetUpperIIS()
```

**Return**

multi-dimensional array of IIS status of upper bounds of variables.

### MVar.HStack()

Stack with other MVar object along horizontal axis.

**Synopsis**

```
MVar HStack(MVar other)
```

**Arguments**

`other`: a MVar object.

**Return**

the result MVar object.

### MVar.Pick()

Given a list of indexes, get variables from MVar object.

**Synopsis**

```
MVar Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. If two dimensional, each row is position of an element.

**Return**

one-dimensional array of desired variables.

### MVar.Repeat()

Repeat each element of MVar along given axis.

**Synopsis**

```
MVar Repeat(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of MVar.

**Return**

new MVar object.

### MVar.RepeatBlock()

Repeat an MVar a number of times along given axis.

**Synopsis**

```
MVar RepeatBlock(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of MVar.

**Return**

new MVar object.

### MVar.Represent()

String representation of MVar object.

**Synopsis**

```
string Represent(int maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

### MVar.Reshape()

Reshape MVar object to new shape.

**Synopsis**

```
MVar Reshape(Shape shp)
```

**Arguments**

`shp`: new shape of M-dimensions.

**Return**

M-dimensional MVar object.

### MVar.Set()

Set values of information associated with variables in MVar object.

**Synopsis**

```
void Set(string info, double val)
```

**Arguments**

`info`: name of information.

`val`: value of information.

### MVar.Set()

Set values of information associated with variables in MVar object.

**Synopsis**

```
void Set(string info, NdArray<double> vals)
```

**Arguments**

`info`: name of information.

`vals`: multi-dimensional array of values of information.

### MVar.SetItem()

Set variable of given index to MVar object.

**Synopsis**

```
void SetItem(long idx, Var var)
```

**Arguments**

`idx`: index of element.

`var`: Var object.

### MVar.Squeeze()

Remove axis of length 1 from shape of MVar object.

**Synopsis**

```
MVar Squeeze(int axis)
```

**Arguments**

`axis`: axis of MVar, where the length is 1.

**Return**

MVar object of (N-1)-dimensional shape.

### MVar.Stack()

Stack with other MVar object along given axis.

**Synopsis**

```
MVar Stack(MVar other, int axis)
```

**Arguments**

`other`: a MVar object.

`axis`: an axis of MVar.

**Return**

the result MVar object.

### MVar.Sum()

Sum of all variables in MVar object.

**Synopsis**

```
MLinExpr Sum()
```

**Return**

sum in zero dimension.

### MVar.Sum()

Sum of variables at given axis of MVar object.

**Synopsis**

```
MLinExpr Sum(int axis)
```

**Arguments**

`axis`: axis of MVar.

**Return**

MLinExpr object in (N-1)-dimension.

### MVar.Transpose()

Perform matrix transpose of MVar object.

**Synopsis**

```
MVar Transpose()
```

**Return**

transposed MVar object.

### MVar.VStack()

Stack with other MVar object along vertical axis.

**Synopsis**

```
MVar VStack(MVar other)
```

**Arguments**

`other`: a MVar object.

**Return**

the result MVar object.

### 26.2.52 MConstr

COPT multi-dimensional linear constraint object. It can be created by calling the method `addMConstr` of *Model*. The following methods are provided:

#### MConstr.Clone()

Clone MConstr object.

**Synopsis**

```
MConstr Clone()
```

**Return**

new MConstr object.

#### MConstr.Diagonal()

Get diagonals of MConstr object.

**Synopsis**

```
MConstr Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MConstr.

`axis2`: 2nd axis of MConstr.

**Return**

(N-1)-dimensional diagonals.

#### MConstr.Expand()

Expand shape of MConstr object.

**Synopsis**

```
MConstr Expand(int axis)
```

**Arguments**

`axis`: axis of MConstr.

**Return**

MConstr object of (N+1)-dimensional shape.

### MConstr.Flatten()

Flatten a MConstr object to a 1-dimensional shape.

**Synopsis**

```
MConstr Flatten()
```

**Return**

a MConstr object collapsed into one dimension.

### MConstr.Get()

Get values of information associated with constraints in MConstr object.

**Synopsis**

```
NdArray<double> Get(string info)
```

**Arguments**

`info`: name of information.

**Return**

multi-dimensional array of information of constraints.

### MConstr.GetDim()

Get i-th dimension of MConstr object.

**Synopsis**

```
long GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MConstr.GetIdx()

Get index of constraints in MConstr object.

**Synopsis**

```
NdArray<int> GetIdx()
```

**Return**

multi-dimensional array of indexes of constraints.

### MConstr.GetItem()

Get constraint of given index from MConstr object.

**Synopsis**

```
Constraint GetItem(long idx)
```

**Arguments**

`idx`: index of constraint.

**Return**

Constraint object.

### MConstr.GetItem()

Get sub-arrays of MConstr object, given view object.

**Synopsis**

```
MConstr GetItem(View view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MConstr object.

### MConstr.GetLowerIIS()

Get IIS status of lower bound of constraints in MConstr object.

**Synopsis**

```
NdArray<int> GetLowerIIS()
```

**Return**

multi-dimensional array of IIS status of lower bounds of constraints.

### MConstr.GetND()

Get number of dimensions of MConstr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MConstr.GetShape()

Get shape of MConstr object.

**Synopsis**

```
Shape GetShape()
```

**Return**

shape object.

### MConstr.GetSize()

Get size of MConstr object.

**Synopsis**

```
long GetSize()
```

**Return**

number of vars.

### MConstr.GetUpperIIS()

Get IIS status of upper bound of constraints in MConstr object.

**Synopsis**

```
NdArray<int> GetUpperIIS()
```

**Return**

multi-dimensional array of IIS status of upper bounds of constraints.

### MConstr.HStack()

Stack with other MConstr object along horizontal axis.

**Synopsis**

```
MConstr HStack(MConstr other)
```

**Arguments**

`other`: a MConstr object.

**Return**

the result MConstr object.

### MConstr.Pick()

Given a list of indexes, get constraints from MConstr object.

**Synopsis**

```
MConstr Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. If two dimensional, each row is position of an element.

**Return**

one-dimensional array of desired constraints.

### MConstr.Represent()

String representation of MConstr object.

**Synopsis**

```
string Represent(int maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

### MConstr.Reshape()

Reshape MConstr object to new shape.

**Synopsis**

```
MConstr Reshape(Shape shp)
```

**Arguments**

`shp`: new shape of M-dimensions.

**Return**

M-dimensional MConstr object.

### MConstr.Set()

Set values of information associated with constraints in MConstr object.

**Synopsis**

```
void Set(string info, NdArray<double> vals)
```

**Arguments**

`info`: name of information.

`vals`: multi-dimensional array of values of information.

### MConstr.Set()

Set values of information associated with constraints in MConstr object.

**Synopsis**

```
void Set(string info, double val)
```

**Arguments**

`info`: name of information.

`val`: value of information.

### MConstr.SetItem()

Set constraint of given index to MConstr object.

**Synopsis**

```
void SetItem(long idx, Constraint constr)
```

**Arguments**

`idx`: index of element.

`constr`: Constraint object.

### MConstr.Squeeze()

Remove axis of length 1 from shape of MConstr object.

**Synopsis**

```
MConstr Squeeze(int axis)
```

**Arguments**

`axis`: axis of MConstr, where the length is 1.

**Return**

MConstr object of (N-1)-dimensional shape.

### MConstr.Stack()

Stack with other MConstr object along given axis.

**Synopsis**

```
MConstr Stack(MConstr other, int axis)
```

**Arguments**

`other`: a MConstr object.

`axis`: an axis of MConstr.

**Return**

the result MConstr object.

### MConstr.Transpose()

Perform matrix transpose of MConstr object.

**Synopsis**

```
MConstr Transpose()
```

**Return**

transposed MConstr object.

#### MConstr.VStack()

Stack with other MConstr object along vertical axis.

**Synopsis**

```
MConstr VStack(MConstr other)
```

**Arguments**

`other`: a MConstr object.

**Return**

the result MConstr object.

### 26.2.53 MConstrBuilder

COPT builder object of multi-dimensional linear constraints. It is used to generate multi-dimensional linear constraints and support operations with the built-in multi-dimensional array *NdArray* in COPT. It is recommended to create MConstrBuilder object by comparing two objects, one of which should be *MVar* object or *MLinExpr* object, by comparison operators. The following methods are provided:

#### MConstrBuilder.MConstrBuilder()

Construct a MConstrBuilder object with the given shape.

**Synopsis**

```
MConstrBuilder(Shape shp)
```

**Arguments**

`shp`: shape of MConstrBuilder.

#### MConstrBuilder.Flatten()

Flatten a MConstrBuilder object to a 1-dimensional shape.

**Synopsis**

```
MConstrBuilder Flatten()
```

**Return**

a MConstrBuilder object collapsed into one dimension.

#### MConstrBuilder.GetExpr()

Get N-dimensional linear expressions associated with N-dimensional constraints.

**Synopsis**

```
MLinExpr GetExpr()
```

**Return**

MLinExpr object.

### MConstrBuilder.GetND()

Get number of dimensions of MConstrBuilder object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MConstrBuilder.GetRange()

Get range from lower bound to upper bound of N-dimensional range constraints.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of range constraints.

### MConstrBuilder.GetSense()

Get sense associated with N-dimensional constraints.

**Synopsis**

```
char GetSense()
```

**Return**

constraint sense.

### MConstrBuilder.Set<T>()

Set N-dimensional constraints to its builder object.

**Synopsis**

```
void Set<T>(
    MLinExpr expr,
    char sense,
    NdArray<T> rhs)
```

**Arguments**

`expr`: MLinExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: N-dimensional constants at right side of constraints.

### MConstrBuilder.Set()

Set N-dimensional constraints to its builder object.

**Synopsis**

```
void Set(
    MLinExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: MLinExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of constraints.

### MConstrBuilder.Set()

Set N-dimensional constraints to its builder object.

**Synopsis**

```
void Set(
    MLinExpr expr,
    char sense,
    MVar rhs)
```

**Arguments**

`expr`: MLinExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MVar object at right side of constraints.

### MConstrBuilder.Set()

Set N-dimensional constraints to its builder object.

**Synopsis**

```
void Set(
    MLinExpr expr,
    char sense,
    MLinExpr rhs)
```

**Arguments**

`expr`: MLinExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MLinExpr object at right side of constraints.

#### MConstrBuilder.SetRange()

Set N-dimensional range constraints to its builder object.

**Synopsis**

```
void SetRange(MLinExpr expr, double range)
```

**Arguments**

`expr`: MLinExpr object.

`range`: length from lower bound to upper bound of the constraint. Must be greater than 0.

### 26.2.54 MExpression

The MExpression class is a generalized version of *Expr*. It represents a linear expression and supports most of methods in Expr class. In addition, it supports linear combination of multi-dimensional objects, such as *MVar* object and *NdArray* object. The following methods are provided:

#### MExpression.MExpression()

Construct a MExpression object with the given constant.

**Synopsis**

```
MExpression(double constant)
```

**Arguments**

`constant`: constant number.

#### MExpression.MExpression()

Construct a MExpression object with the given variable.

**Synopsis**

```
MExpression(Var var)
```

**Arguments**

`var`: variable object.

#### MExpression.MExpression()

Construct a MExpression object with the given linear expression.

**Synopsis**

```
MExpression(Expr expr)
```

**Arguments**

`expr`: a linear expression.

### MExpression.AddConstant()

Add constant for the expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### MExpression.AddExpr()

Add a linear expression to MExpression object.

**Synopsis**

```
void AddExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

### MExpression.AddMExpr()

Add MExpression to MExpression object.

**Synopsis**

```
void AddMExpr(MExpression expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MExpression.AddTerm()

Add a linear term to MExpression object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

### MExpression.Clone()

Clone MExpression object.

**Synopsis**

```
MExpression Clone()
```

**Return**

new MExpression object.

#### MExpression.Evaluate()

evaluate MExpression object after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of MExpression object.

#### MExpression.GetConstant()

Get constant in expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in expression.

#### MExpression.Represent()

String representation of MExpression object.

**Synopsis**

```
string Represent(int maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

#### MExpression.SetConstant()

Set constant for the expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### 26.2.55 MLinExpr

COPT multi-dimensional linear expression object. It is used to construct multi-dimensional linear expressions and perform operations with the built-in multi-dimensional array *NdArray* in COPT. Its elements are *MExpression* objects. It can be created by linear combination of *MVar* objects. The following methods are provided:

### MLinExpr.AddConstant()

Add constant to each expression in MLinExpr object.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### MLinExpr.AddConstant()

Add constants to each expression in MLinExpr object.

**Synopsis**

```
void AddConstant(NdArray<double> constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

### MLinExpr.AddExpr()

Add a linear expression to each expression in MLinExpr object.

**Synopsis**

```
void AddExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

### MLinExpr.AddMExpr()

Add MExpression to each expression in MLinExpr object.

**Synopsis**

```
void AddMExpr(MExpression expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MLinExpr.AddMLinExpr()

Add linear expressions to MLinExpr object.

**Synopsis**

```
void AddMLinExpr(MLinExpr exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MLinExpr object.

`mult`: the same multiplier for added linear expressions, default value is 1.0.

### MLinExpr.AddTerm()

Add a linear term to MLinExpr object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

### MLinExpr.AddTerms()

Add terms to expressions in MLinExpr object.

**Synopsis**

```
void AddTerms(MVar vars, NdArray<double> coeffs)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`coeffs`: N-dimension NdArray object of coefficients for added terms.

### MLinExpr.AddTerms()

Add terms to expressions in MLinExpr object.

**Synopsis**

```
void AddTerms(MVar vars, double mult)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`mult`: the same coefficient for added terms, default value 1.0.

### MLinExpr.Clear()

Clear MLinExpr object.

**Synopsis**

```
void Clear()
```

### MLinExpr.Clone()

Clone MLinExpr object.

**Synopsis**

```
MLinExpr Clone()
```

**Return**

new MLinExpr object.

### MLinExpr.Diagonal()

Get diagonals of MLinExpr object.

**Synopsis**

```
MLinExpr Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MLinExpr.

`axis2`: 2nd axis of MLinExpr.

**Return**

(N-1)-dimensional diagonals.

### MLinExpr.Evaluate()

Evaluate MLinExpr object after solving.

**Synopsis**

```
NdArray<double> Evaluate()
```

**Return**

NdArray object storing value of each linear expression.

### MLinExpr.Expand()

Expand shape of MLinExpr object.

**Synopsis**

```
MLinExpr Expand(int axis)
```

**Arguments**

`axis`: axis of MLinExpr.

**Return**

MLinExpr object of (N+1)-dimensional shape.

### MLinExpr.Flatten()

Flatten a MLinExpr object to a 1-dimensional shape.

**Synopsis**

```
MLinExpr Flatten()
```

**Return**

a MLinExpr object collapsed into one dimension.

### MLinExpr.GetDim()

Get i-th dimension of MLinExpr object.

**Synopsis**

```
long GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MLinExpr.GetItem()

Get expression of given index from MLinExpr object.

**Synopsis**

```
MExpression GetItem(long idx)
```

**Arguments**

`idx`: index of expression.

**Return**

MExpression object.

### MLinExpr.GetItem()

Get sub-arrays of MLinExpr object, given view object.

**Synopsis**

```
MLinExpr GetItem(View view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MLinExpr object.

### MLinExpr.GetND()

Get number of dimensions of MLinExpr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MLinExpr.GetShape()

Get shape of MLinExpr object.

**Synopsis**

```
Shape GetShape()
```

**Return**

shape object.

### MLinExpr.GetSize()

Get size of MLinExpr object.

**Synopsis**

```
long GetSize()
```

**Return**

number of linear expressions.

### MLinExpr.HStack<T>()

Stack with other NdArray object along horizontal axis.

**Synopsis**

```
MLinExpr HStack<T>(NdArray<T> other)
```

**Arguments**

`other`: a NdArray object.

**Return**

the result MLinExpr object.

### MLinExpr.HStack()

Stack with other MLinExpr object along horizontal axis.

**Synopsis**

```
MLinExpr HStack(MLinExpr other)
```

**Arguments**

`other`: a MLinExpr object.

**Return**

the result MLinExpr object.

### MLinExpr.HStack()

Stack with other MVar object along horizontal axis.

**Synopsis**

```
MLinExpr HStack(MVar other)
```

**Arguments**

`other`: a MVar object.

**Return**

the result MLinExpr object.

### MLinExpr.Pick()

Given a list of indexes, get linear expressions from MLinExpr object.

**Synopsis**

```
MLinExpr Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. If two dimensional, each row is position of an element.

**Return**

one-dimensional array of desired linear expressions.

### MLinExpr.Repeat()

Repeat each element of MLinExpr along given axis.

**Synopsis**

```
MLinExpr Repeat(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of MLinExpr.

**Return**

new MLinExpr object.

### MLinExpr.RepeatBlock()

Repeat an MLinExpr a number of times along given axis.

**Synopsis**

```
MLinExpr RepeatBlock(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of MLinExpr.

**Return**

new MLinExpr object.

### MLinExpr.Represent()

String representation of MLinExpr object.

**Synopsis**

```
string Represent(int maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

### MLinExpr.Reshape()

Reshape MLinExpr object to new shape.

**Synopsis**

```
MLinExpr Reshape(Shape shp)
```

**Arguments**

`shp`: new shape of M-dimensions.

**Return**

M-dimensional MLinExpr object.

### MLinExpr.SetItem()

Set expression of given index to MLinExpr object.

**Synopsis**

```
void SetItem(long idx, MExpression expr)
```

**Arguments**

`idx`: index of element.

`expr`: MExpression object.

### MLinExpr.Squeeze()

Remove axis of length 1 from shape of MLinExpr object.

**Synopsis**

```
MLinExpr Squeeze(int axis)
```

**Arguments**

`axis`: axis of MLinExpr, where the length is 1.

**Return**

MLinExpr object of (N-1)-dimensional shape.

### MLinExpr.Stack<T>()

Stack with other NdArray object along given axis.

**Synopsis**

```
MLinExpr Stack<T>(NdArray<T> other, int axis)
```

**Arguments**

`other`: a NdArray object.

`axis`: an axis of MLinExpr.

**Return**

the result MLinExpr object.

### MLinExpr.Stack()

Stack with other MLinExpr object along given axis.

**Synopsis**

```
MLinExpr Stack(MLinExpr other, int axis)
```

**Arguments**

`other`: a MLinExpr object.

`axis`: an axis of MLinExpr.

**Return**

the result MLinExpr object.

### MLinExpr.Stack()

Stack with other MLinExpr object along given axis.

**Synopsis**

```
MLinExpr Stack(MVar other, int axis)
```

**Arguments**

`other`: a MVar object.

`axis`: an axis of MLinExpr.

**Return**

the result MLinExpr object.

### MLinExpr.SubConstant()

Subtract constants from each expression in MLinExpr object.

**Synopsis**

```
void SubConstant(NdArray<double> constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

### MLinExpr.Sum()

Sum of all expressions in MLinExpr object.

**Synopsis**

```
MLinExpr Sum()
```

**Return**

sum in zero dimension.

### MLinExpr.Sum()

Sum of variables at given axis of MLinExpr object.

**Synopsis**

```
MLinExpr Sum(int axis)
```

**Arguments**

`axis`: axis of MLinExpr.

**Return**

MLinExpr object in (N-1)-dimension.

### MLinExpr.Transpose()

Perform matrix transpose of MLinExpr object.

**Synopsis**

```
MLinExpr Transpose()
```

**Return**

transposed MLinExpr object.

### MLinExpr.VStack<T>()

Stack with other NdArray object along vertical axis.

**Synopsis**

```
MLinExpr VStack<T>(NdArray<T> other)
```

**Arguments**

`other`: a NdArray object.

**Return**

the result MLinExpr object.

#### MLinExpr.VStack()

Stack with other MLinExpr object along vertical axis.

**Synopsis**

```
MLinExpr VStack(MLinExpr other)
```

**Arguments**

`other`: a MLinExpr object.

**Return**

the result MLinExpr object.

#### MLinExpr.VStack()

Stack with other MVar object along vertical axis.

**Synopsis**

```
MLinExpr VStack(MVar other)
```

**Arguments**

`other`: a MVar object.

**Return**

the result MLinExpr object.

### 26.2.56 MPsdConstr

The *MPsdConstr* class in COPT represents multi-dimensional semidefinite constraints. It is generated through the methods `addConstrs` or `addConstr` of *Model*.

The following member methods are provided:

#### MPsdConstr.Clone()

Clone MPsdConstr object.

**Synopsis**

```
MPsdConstr Clone()
```

**Return**

new MPsdConstr object.

#### MPsdConstr.Diagonal()

Get diagonals of MPsdConstr object.

**Synopsis**

```
MPsdConstr Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MPsdConstr.

`axis2`: 2nd axis of MPsdConstr.

**Return**

(N-1)-dimensional diagonals.

### MPsdConstr.Expand()

Expand shape of MPsdConstr object.

**Synopsis**

```
MPsdConstr Expand(int axis)
```

**Arguments**

`axis`: axis of MPsdConstr.

**Return**

MPsdConstr object of (N+1)-dimensional shape.

### MPsdConstr.Flatten()

Flatten a MPsdConstr object to a 1-dimensional shape.

**Synopsis**

```
MPsdConstr Flatten()
```

**Return**

a MPsdConstr object collapsed into one dimension.

### MPsdConstr.Get()

Get values of information associated with PSD constraints in MPsdConstr object.

**Synopsis**

```
NdArray<double> Get(string info)
```

**Arguments**

`info`: name of information.

**Return**

multi-dimensional array of information of PSD constraints.

### MPsdConstr.GetDim()

Get i-th dimension of MPsdConstr object.

**Synopsis**

```
long GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MPsdConstr.GetIdx()

Get index of PSD constraints in MPsdConstr object.

**Synopsis**

```
NdArray<int> GetIdx()
```

**Return**

multi-dimensional array of indexes of PSD constraints.

### MPsdConstr.GetItem()

Get PSD constraint of given index from MPsdConstr object.

**Synopsis**

```
PsdConstraint GetItem(long idx)
```

**Arguments**

`idx`: index of var.

**Return**

PsdConstraint object.

### MPsdConstr.GetItem()

Get sub-arrays of MPsdConstr object, given view object.

**Synopsis**

```
MPsdConstr GetItem(View view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MPsdConstr object.

### MPsdConstr.GetND()

Get number of dimensions of MPsdConstr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MPsdConstr.GetShape()

Get shape of MPsdConstr object.

**Synopsis**

```
Shape GetShape()
```

**Return**

shape object.

### MPsdConstr.GetSize()

Get size of MPsdConstr object.

**Synopsis**

```
long GetSize()
```

**Return**

number of vars.

### MPsdConstr.HStack()

Stack with other MPsdConstr object along horizontal axis.

**Synopsis**

```
MPsdConstr HStack(MPsdConstr other)
```

**Arguments**

`other`: a MPsdConstr object.

**Return**

the result MPsdConstr object.

### MPsdConstr.Pick()

Given a list of indexes, get PSD constraints from MPsdConstr object.

**Synopsis**

```
MPsdConstr Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. If two dimensional, each row is position of an element.

**Return**

one-dimensional array of desired PSD constraints.

### MPsdConstr.Represent()

String representation of MPsdConstr object.

**Synopsis**

```
string Represent(int maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

### MPsdConstr.Reshape()

Reshape MPsdConstr object to new shape.

**Synopsis**

```
MPsdConstr Reshape(Shape shp)
```

**Arguments**

`shp`: new shape of M-dimensions.

**Return**

M-dimensional MPsdConstr object.

### MPsdConstr.Set()

Set values of information associated with PSD constraints in MPsdConstr object.

**Synopsis**

```
void Set(string info, double val)
```

**Arguments**

`info`: name of information.

`val`: value of information.

### MPsdConstr.Set()

Set values of information associated with PSD constraints in MPsdConstr object.

**Synopsis**

```
void Set(string info, NdArray<double> vals)
```

**Arguments**

`info`: name of information.

`vals`: multi-dimensional array of values of information.

### MPsdConstr.SetItem()

Set PSD constraint of given index to MPsdConstr object.

**Synopsis**

```
void SetItem(long idx, PsdConstraint constr)
```

**Arguments**

`idx`: index of element.

`constr`: PsdConstraint object.

### MPsdConstr.Squeeze()

Remove axis of length 1 from shape of MPsdConstr object.

**Synopsis**

```
MPsdConstr Squeeze(int axis)
```

**Arguments**

`axis`: axis of MPsdConstr, where the length is 1.

**Return**

MPsdConstr object of (N-1)-dimensional shape.

### MPsdConstr.Stack()

Stack with other MPsdConstr object along given axis.

**Synopsis**

```
MPsdConstr Stack(MPsdConstr other, int axis)
```

**Arguments**

`other`: a MPsdConstr object.

`axis`: an axis of MPsdConstr.

**Return**

the result MPsdConstr object.

### MPsdConstr.Transpose()

Perform matrix transpose of MPsdConstr object.

**Synopsis**

```
MPsdConstr Transpose()
```

**Return**

transposed MPsdConstr object.

#### MPsdConstr.VStack()

Stack with other MPsdConstr object along vertical axis.

**Synopsis**

```
MPsdConstr VStack(MPsdConstr other)
```

**Arguments**

`other`: a MPsdConstr object.

**Return**

the result MPsdConstr object.

### 26.2.57 MPsdConstrBuilder

The *MPsdConstrBuilder* class in COPT serves as a builder for multi-dimensional semidefinite constraints. It is used to generate multi-dimensional semidefinite constraints and supports operations with the built-in multi-dimensional array *NdArray*. An *MPsdConstrBuilder* object can be created through comparison operations between two objects, one of which can be an *MPsdExpr* object. The following member methods are provided:

#### MPsdConstrBuilder.MPsdConstrBuilder()

Construct a MPsdConstrBuilder object with the given shape.

**Synopsis**

```
MPsdConstrBuilder(Shape shp)
```

**Arguments**

`shp`: shape of MPsdConstrBuilder.

#### MPsdConstrBuilder.Flatten()

Flatten a MPsdConstrBuilder object to a 1-dimensional shape.

**Synopsis**

```
MPsdConstrBuilder Flatten()
```

**Return**

a MPsdConstrBuilder object collapsed into one dimension.

#### MPsdConstrBuilder.GetND()

Get number of dimensions of MPsdConstrBuilder object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MPsdConstrBuilder.GetPsdExpr()

Get N-dimensional PSD expressions associated with N-dimensional constraints.

**Synopsis**

```
MPsdExpr GetPsdExpr()
```

**Return**

MPsdExpr object.

### MPsdConstrBuilder.GetRange()

Get range from lower bound to upper bound of N-dimensional range constraints.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of range constraints.

### MPsdConstrBuilder.GetSense()

Get sense associated with N-dimensional PSD constraints.

**Synopsis**

```
char GetSense()
```

**Return**

PSD constraint sense.

### MPsdConstrBuilder.Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
void Set(
    MPsdExpr expr,
    char sense,
    NdArray<double> rhs)
```

**Arguments**

`expr`: MPsdExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: N-dimensional constants at right side of constraints.

### MPsdConstrBuilder.Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
void Set(
    MPsdExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: MPsdExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of constraints.

### MPsdConstrBuilder.Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
void Set(
    MPsdExpr expr,
    char sense,
    MVar rhs)
```

**Arguments**

`expr`: MPsdExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MVar object at right side of constraints.

### MPsdConstrBuilder.Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
void Set(
    MPsdExpr expr,
    char sense,
    MLinExpr rhs)
```

**Arguments**

`expr`: MPsdExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MLinExpr object at right side of constraints.

#### MPsdConstrBuilder.Set()

Set N-dimensional PSD constraints to its builder object.

**Synopsis**

```
void Set(
    MPsdExpr expr,
    char sense,
    MPsdExpr rhs)
```

**Arguments**

`expr`: MPsdExpr object

`sense`: PSD constraint sense other than COPT_RANGE.

`rhs`: MPsdExpr object at right side of PSD constraints.

#### MPsdConstrBuilder.SetRange()

Set N-dimensional range PSD constraints to its builder object.

**Synopsis**

```
void SetRange(MPsdExpr expr, double range)
```

**Arguments**

`expr`: MPsdExpr object.

`range`: length from lower bound to upper bound of PSD constraint. Must be greater than 0.

### 26.2.58 MPsdExpr

The *MPsdExpr* class in COPT represents multi-dimensional semidefinite expressions. It is used to construct multi-dimensional semidefinite expressions and perform operations with the built-in multi-dimensional array *NdArray* in COPT. The elements of *MPsdExpr* are either *PsdExpr* objects or their multi-dimensional linear combinations. The following member methods are provided:

#### MPsdExpr.AddConstant()

Add constant to each quadratic expression in MPsdExpr object.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### MPsdExpr.AddConstant()

Add constants to each PSD expression in MPsdExpr object.

**Synopsis**

```
void AddConstant(NdArray<double> constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

### MPsdExpr.AddLinExpr()

Add a linear expression to each PsdExpr in MPsdExpr object.

**Synopsis**

```
void AddLinExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

### MPsdExpr.AddMExpr()

Add MExpression to each PSD expression in MPsdExpr object.

**Synopsis**

```
void AddMExpr(MExpression expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MPsdExpr.AddMLinExpr()

Add linear expressions to MPsdExpr object.

**Synopsis**

```
void AddMLinExpr(MLinExpr exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MLinExpr object.

`mult`: the same multiplier for added linear expressions, default value is 1.0.

### MPsdExpr.AddMPsdExpr()

Add PSD expressions to MPsdExpr object.

**Synopsis**

```
void AddMPsdExpr(MPsdExpr exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MPsdExpr object.

`mult`: the same multiplier for added PSD expressions, default value is 1.0.

### MPsdExpr.AddPsdExpr()

Add a PSD expression to each PSD expression in MPsdExpr object.

**Synopsis**

```
void AddPsdExpr(PsdExpr expr, double mult)
```

**Arguments**

`expr`: PSD expression object.

`mult`: the multiplier of PSD expression, default value is 1.0.

### MPsdExpr.AddTerm()

Add a PSD term to MPsdExpr object.

**Synopsis**

```
void AddTerm(PsdVar var, SymMatExpr expr)
```

**Arguments**

`var`: PSD variable of new PSD term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

### MPsdExpr.AddTerm()

Add a linear term to MPsdExpr object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

### MPsdExpr.AddTerm()

Add a PSD term to MPsdExpr object.

**Synopsis**

```
void AddTerm(PsdVar var, SymMatrix mat)
```

**Arguments**

`var`: PSD variable of new PSD term.

`mat`: coefficient matrix of new PSD term.

### MPsdExpr.AddTerms()

Add terms to PSD expressions in MPsdExpr object.

**Synopsis**

```
void AddTerms(MVar vars, double mult)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`mult`: the same coefficient for added terms, default value 1.0.

### MPsdExpr.AddTerms()

Add terms to PSD expressions in MPsdExpr object.

**Synopsis**

```
void AddTerms(MVar vars, NdArray<double> coeffs)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`coeffs`: N-dimension NdArray object of coefficients for added terms.

### MPsdExpr.Clear()

Clear MPsdExpr object.

**Synopsis**

```
void Clear()
```

### MPsdExpr.Clone()

Clone MPsdExpr object.

**Synopsis**

```
MPsdExpr Clone()
```

**Return**

new MPsdExpr object.

### MPsdExpr.Diagonal()

Get diagonals of MPsdExpr object.

**Synopsis**

```
MPsdExpr Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MPsdExpr.

`axis2`: 2nd axis of MPsdExpr.

**Return**

(N-1)-dimensional diagonals.

### MPsdExpr.Evaluate()

Evaluate MPsdExpr object after solving.

**Synopsis**

```
NdArray<double> Evaluate()
```

**Return**

NdArray object storing value of each PSD expression.

### MPsdExpr.Expand()

Expand shape of MPsdExpr object.

**Synopsis**

```
MPsdExpr Expand(int axis)
```

**Arguments**

`axis`: axis of MPsdExpr.

**Return**

MPsdExpr object of (N+1)-dimensional shape.

### MPsdExpr.Flatten()

Flatten a MPsdExpr object to a 1-dimensional shape.

**Synopsis**

```
MPsdExpr Flatten()
```

**Return**

a MPsdExpr object collapsed into one dimension.

### MPsdExpr.GetDim()

Get i-th dimension of MPsdExpr object.

**Synopsis**

```
long GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MPsdExpr.GetItem()

Get PSD expression of given index from MPsdExpr object.

**Synopsis**

```
PsdExpr GetItem(long idx)
```

**Arguments**

`idx`: index of PSD expression.

**Return**

PSD expression object.

### MPsdExpr.GetItem()

Get sub-arrays of MPsdExpr object, given view object.

**Synopsis**

```
MPsdExpr GetItem(View view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MPsdExpr object.

### MPsdExpr.GetND()

Get number of dimensions of MPsdExpr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MPsdExpr.GetShape()

Get shape of MPsdExpr object.

**Synopsis**

```
Shape GetShape()
```

**Return**

shape object.

### MPsdExpr.GetSize()

Get size of MPsdExpr object.

**Synopsis**

```
long GetSize()
```

**Return**

number of linear expressions.

### MPsdExpr.HStack<T>()

Stack with other NdArray object along horizontal axis.

**Synopsis**

```
MPsdExpr HStack<T>(NdArray<T> other)
```

**Arguments**

`other`: a NdArray object.

**Return**

the result MPsdExpr object.

### MPsdExpr.HStack()

Stack with other MPsdExpr object along horizontal axis.

**Synopsis**

```
MPsdExpr HStack(MPsdExpr other)
```

**Arguments**

`other`: a MPsdExpr object.

**Return**

the result MPsdExpr object.

### MPsdExpr.HStack()

Stack with other MLinExpr object along horizontal axis.

**Synopsis**

```
MPsdExpr HStack(MLinExpr other)
```

**Arguments**

`other`: a MLinExpr object.

**Return**

the result MPsdExpr object.

### MPsdExpr.HStack()

Stack with other MVar object along horizontal axis.

**Synopsis**

```
MPsdExpr HStack(MVar other)
```

**Arguments**

`other`: a MVar object.

**Return**

the result MPsdExpr object.

### MPsdExpr.Pick()

Given a list of indexes, get PSD expressions from MPsdExpr object.

**Synopsis**

```
MPsdExpr Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. If two dimensional, each row is position of an element.

**Return**

one-dimensional array of desired PSD expressions.

### MPsdExpr.Repeat()

Repeat each element of MPsdExpr along given axis.

**Synopsis**

```
MPsdExpr Repeat(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of MPsdExpr.

**Return**

new MPsdExpr object.

### MPsdExpr.RepeatBlock()

Repeat an MPsdExpr a number of times along given axis.

**Synopsis**

```
MPsdExpr RepeatBlock(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of MPsdExpr.

**Return**

new MPsdExpr object.

### MPsdExpr.Represent()

String representation of MPsdExpr object.

**Synopsis**

```
string Represent(int maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

### MPsdExpr.Reshape()

Reshape MPsdExpr object to new shape.

**Synopsis**

```
MPsdExpr Reshape(Shape shp)
```

**Arguments**

`shp`: new shape of M-dimensions.

**Return**

M-dimensional MPsdExpr object.

### MPsdExpr.SetItem()

Set expression of given index to MPsdExpr object.

**Synopsis**

```
void SetItem(long idx, MExpression expr)
```

**Arguments**

`idx`: index of element.

`expr`: MExpression object.

### MPsdExpr.SetItem()

Set PSD expression of given index to MPsdExpr object.

**Synopsis**

```
void SetItem(long idx, PsdExpr expr)
```

**Arguments**

`idx`: index of element.

`expr`: PSD expression object.

### MPsdExpr.Squeeze()

Remove axis of length 1 from shape of MPsdExpr object.

**Synopsis**

```
MPsdExpr Squeeze(int axis)
```

**Arguments**

`axis`: axis of MPsdExpr, where the length is 1.

**Return**

MPsdExpr object of (N-1)-dimensional shape.

### MPsdExpr.Stack<T>()

Stack with other NdArray object along given axis.

**Synopsis**

```
MPsdExpr Stack<T>(NdArray<T> other, int axis)
```

**Arguments**

`other`: a NdArray object.

`axis`: an axis of MPsdExpr.

**Return**

the result MPsdExpr object.

### MPsdExpr.Stack()

Stack with other MPsdExpr object along given axis.

**Synopsis**

```
MPsdExpr Stack(MPsdExpr other, int axis)
```

**Arguments**

`other`: a MPsdExpr object.

`axis`: an axis of MPsdExpr.

**Return**

the result MPsdExpr object.

### MPsdExpr.Stack()

Stack with other MLinExpr object along given axis.

**Synopsis**

```
MPsdExpr Stack(MLinExpr other, int axis)
```

**Arguments**

`other`: a MLinExpr object.

`axis`: an axis of MPsdExpr.

**Return**

the result MPsdExpr object.

### MPsdExpr.Stack()

Stack with other MPsdExpr object along given axis.

**Synopsis**

```
MPsdExpr Stack(MVar other, int axis)
```

**Arguments**

`other`: a MVar object.

`axis`: an axis of MPsdExpr.

**Return**

the result MPsdExpr object.

### MPsdExpr.SubConstant()

Subtract constants from each PSD expression in MPsdExpr object.

**Synopsis**

```
void SubConstant(NdArray<double> constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

### MPsdExpr.Sum()

Sum of all expressions in MPsdExpr object.

**Synopsis**

```
MPsdExpr Sum()
```

**Return**

sum in zero dimension.

### MPsdExpr.Sum()

Sum of variables at given axis of MPsdExpr object.

**Synopsis**

```
MPsdExpr Sum(int axis)
```

**Arguments**

`axis`: axis of MPsdExpr.

**Return**

MPsdExpr object in (N-1)-dimension.

### MPsdExpr.Transpose()

Perform matrix transpose of MPsdExpr object.

**Synopsis**

```
MPsdExpr Transpose()
```

**Return**

transposed MPsdExpr object.

### MPsdExpr.VStack<T>()

Stack with other NdArray object along vertical axis.

**Synopsis**

```
MPsdExpr VStack<T>(NdArray<T> other)
```

**Arguments**

`other`: a NdArray object.

**Return**

the result MPsdExpr object.

### MPsdExpr.VStack()

Stack with other MPsdExpr object along vertical axis.

**Synopsis**

```
MPsdExpr VStack(MPsdExpr other)
```

**Arguments**

`other`: a MPsdExpr object.

**Return**

the result MPsdExpr object.

### MPsdExpr.VStack()

Stack with other MLinExpr object along vertical axis.

**Synopsis**

```
MPsdExpr VStack(MLinExpr other)
```

**Arguments**

`other`: a MLinExpr object.

**Return**

the result MPsdExpr object.

#### MPsdExpr.VStack()

Stack with other MVar object along vertical axis.

**Synopsis**

```
MPsdExpr VStack(MVar other)
```

**Arguments**

`other`: a MVar object.

**Return**

the result MPsdExpr object.

### 26.2.59 MQConstr

The MQConstr class is a COPT multi-dimensional quadratic constraint object. It can be created by calling the method `addMQConstr` of *Model*. The following methods are provided:

#### MQConstr.Clone()

Clone MQConstr object.

**Synopsis**

```
MQConstr Clone()
```

**Return**

new MQConstr object.

#### MQConstr.Diagonal()

Get diagonals of MQConstr object.

**Synopsis**

```
MQConstr Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MQConstr.

`axis2`: 2nd axis of MQConstr.

**Return**

(N-1)-dimensional diagonals.

### MQConstr.Expand()

Expand shape of MQConstr object.

**Synopsis**

```
MQConstr Expand(int axis)
```

**Arguments**

`axis`: axis of MQConstr.

**Return**

MQConstr object of (N+1)-dimensional shape.

### MQConstr.Flatten()

Flatten a MQConstr object to a 1-dimensional shape.

**Synopsis**

```
MQConstr Flatten()
```

**Return**

a MQConstr object collapsed into one dimension.

### MQConstr.Get()

Get values of information associated with quadratic constraints in MQConstr object.

**Synopsis**

```
NdArray<double> Get(string info)
```

**Arguments**

`info`: name of information.

**Return**

multi-dimensional array of information of quadratic constraints.

### MQConstr.GetDim()

Get i-th dimension of MQConstr object.

**Synopsis**

```
long GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MQConstr.GetIdx()

Get index of quadratic constraints in MQConstr object.

**Synopsis**

```
NdArray<int> GetIdx()
```

**Return**

multi-dimensional array of indexes of quadratic constraints.

### MQConstr.GetItem()

Get quadratic constraint of given index from MQConstr object.

**Synopsis**

```
QConstraint GetItem(long idx)
```

**Arguments**

`idx`: index of quadratic constraint.

**Return**

QConstraint object.

### MQConstr.GetItem()

Get sub-arrays of MQConstr object, given view object.

**Synopsis**

```
MQConstr GetItem(View view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MQConstr object.

### MQConstr.GetND()

Get number of dimensions of MQConstr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MQConstr.GetRhs()

Get RHS of quadratic constraints in MQConstr object.

**Synopsis**

```
NdArray<double> GetRhs()
```

**Return**

multi-dimensional array of RHS of quadratic constraints.

### MQConstr.GetSense()

Get senses of quadratic constraints in MQConstr object.

**Synopsis**

```
NdArray<char> GetSense()
```

**Return**

multi-dimensional array of senses of quadratic constraints.

### MQConstr.GetShape()

Get shape of MQConstr object.

**Synopsis**

```
Shape GetShape()
```

**Return**

shape object.

### MQConstr.GetSize()

Get size of MQConstr object.

**Synopsis**

```
long GetSize()
```

**Return**

number of vars.

### MQConstr.HStack()

Stack with other MQConstr object along horizontal axis.

**Synopsis**

```
MQConstr HStack(MQConstr other)
```

**Arguments**

`other`: a MQConstr object.

**Return**

the result MQConstr object.

### MQConstr.Pick()

Given a list of indexes, get quadratic constraints from MQConstr object.

**Synopsis**

```
MQConstr Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. If two dimensional, each row is position of an element.

**Return**

one-dimensional array of desired quadratic constraints.

### MQConstr.Represent()

String representation of MQConstr object.

**Synopsis**

```
string Represent(uint maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

### MQConstr.Reshape()

Reshape MQConstr object to new shape.

**Synopsis**

```
MQConstr Reshape(Shape shp)
```

**Arguments**

`shp`: new shape of M-dimensions.

**Return**

M-dimensional MQConstr object.

### MQConstr.Set()

Set values of information associated with quadratic constraints in MQConstr object.

**Synopsis**

```
void Set(string info, NdArray<double> vals)
```

**Arguments**

`info`: name of information.

`vals`: multi-dimensional array of values of information.

### MQConstr.Set()

Set values of information associated with quadratic constraints in MQConstr object.

**Synopsis**

```
void Set(string info, double val)
```

**Arguments**

`info`: name of information.

`val`: value of information.

### MQConstr.SetItem()

Set quadratic constraint of given index to MQConstr object.

**Synopsis**

```
void SetItem(long idx, QConstraint constr)
```

**Arguments**

`idx`: index of element.

`constr`: quadratic constraint object.

### MQConstr.Squeeze()

Remove axis of length 1 from shape of MQConstr object.

**Synopsis**

```
MQConstr Squeeze(int axis)
```

**Arguments**

`axis`: axis of MQConstr, where the length is 1.

**Return**

MQConstr object of (N-1)-dimensional shape.

### MQConstr.Stack()

Stack with other MQConstr object along given axis.

**Synopsis**

```
MQConstr Stack(MQConstr other, int axis)
```

**Arguments**

`other`: a MQConstr object.

`axis`: an axis of MQConstr.

**Return**

the result MQConstr object.

#### MQConstr.Transpose()

Perform matrix transpose of MQConstr object.

**Synopsis**

```
MQConstr Transpose()
```

**Return**

transposed MQConstr object.

#### MQConstr.VStack()

Stack with other MQConstr object along vertical axis.

**Synopsis**

```
MQConstr VStack(MQConstr other)
```

**Arguments**

`other`: a MQConstr object.

**Return**

the result MQConstr object.

### 26.2.60 MQConstrBuilder

The MQConstrBuilder class is a COPT builder object of multi-dimensional quadratic constraints. It is used to generate multi-dimensional quadratic constraints and supports operations with the built-in multi-dimensional array *NdArray* in COPT. It can be created by comparing two objects, one of which should be *MQuadExpr* object, by comparison operators. The following methods are provided:

#### MQConstrBuilder.MQConstrBuilder()

Construct a MQConstrBuilder object with the given shape.

**Synopsis**

```
MQConstrBuilder(Shape shp)
```

**Arguments**

`shp`: shape of MQConstrBuilder.

#### MQConstrBuilder.Flatten()

Flatten a MQConstrBuilder object to a 1-dimensional shape.

**Synopsis**

```
MQConstrBuilder Flatten()
```

**Return**

a MQConstrBuilder object collapsed into one dimension.

### MQConstrBuilder.GetND()

Get number of dimensions of MQConstrBuilder object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MQConstrBuilder.GetQuadExpr()

Get N-dimensional quadratic expressions associated with N-dimensional quadratic constraints.

**Synopsis**

```
MQuadExpr GetQuadExpr()
```

**Return**

MQuadExpr object.

### MQConstrBuilder.GetSense()

Get sense associated with N-dimensional constraints.

**Synopsis**

```
char GetSense()
```

**Return**

constraint sense.

### MQConstrBuilder.Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
void Set(
    MQuadExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of constraints.

### MQConstrBuilder.Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
void Set(
    MQuadExpr expr,
    char sense,
    MVar rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MVar object at right side of quadratic constraints.

### MQConstrBuilder.Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
void Set(
    MQuadExpr expr,
    char sense,
    MLinExpr rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MLinExpr object at right side of quadratic constraints.

### MQConstrBuilder.Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
void Set(
    MQuadExpr expr,
    char sense,
    MQuadExpr rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: MQuadExpr object at right side of quadratic constraints.

#### MQConstrBuilder.Set()

Set N-dimensional quadratic constraints to its builder object.

**Synopsis**

```
void Set(
    MQuadExpr expr,
    char sense,
    NdArray<double> rhs)
```

**Arguments**

`expr`: MQuadExpr object

`sense`: constraint sense other than COPT_RANGE.

`rhs`: N-dimensional constants at right side of quadratic constraints.

### 26.2.61 MQExpression

The MQExpression class is a generalized version of *QuadExpr*. It represents a quadratic expression and supports most of methods in QuadExpr class. In addition, it supports quadratic combination of multi-dimensional objects, such as *MVar* object and *MLinExpr* object. The following methods are provided:

#### MQExpression.MQExpression()

Construct a MQExpression object with the given constant.

**Synopsis**

```
MQExpression(double constant)
```

**Arguments**

`constant`: constant number.

#### MQExpression.MQExpression()

Construct a MQExpression object with the given quadratic expression.

**Synopsis**

```
MQExpression(QuadExpr expr)
```

**Arguments**

`expr`: a quadratic expression.

#### MQExpression.AddConstant()

Add constant for the expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### MQExpression.AddExpr()

Add a linear expression to MQExpression object.

**Synopsis**

```
void AddExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

### MQExpression.AddMExpr()

Add MExpression to MQExpression object.

**Synopsis**

```
void AddMExpr(MExpression expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MQExpression.AddMQExpr()

Add MQExpression to MQExpression object.

**Synopsis**

```
void AddMQExpr(MQExpression expr, double mult)
```

**Arguments**

`expr`: MQExpression object.

`mult`: the multiplier of MQExpression, default value is 1.0.

### MQExpression.AddQuadExpr()

Add a quadratic expression to MQExpression object.

**Synopsis**

```
void AddQuadExpr(MExpression left, Expr right)
```

**Arguments**

`left`: MExpression object.

`right`: Expr object.

### MQExpression.AddQuadExpr()

Add a quadratic expression to MQExpression object.

**Synopsis**

```
void AddQuadExpr(MExpression left, MExpression right)
```

**Arguments**

`left`: left MExpression object.

`right`: right MExpression object.

### MQExpression.AddQuadExpr()

Add a quadratic expression to MQExpression object.

**Synopsis**

```
void AddQuadExpr(QuadExpr expr, double mult)
```

**Arguments**

`expr`: quadratic expression object.

`mult`: the multiplier of quadratic expression, default value is 1.0.

### MQExpression.AddQuadExpr()

Add a quadratic expression to MQExpression object.

**Synopsis**

```
void AddQuadExpr(MExpression expr, Var var)
```

**Arguments**

`expr`: MExpression object.

`var`: Var object.

### MQExpression.AddTerm()

Add a linear term to MQExpression object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

### MQExpression.AddTerm()

Add a quadratic term to MQExpression object.

**Synopsis**

```
void AddTerm(
    Var var1,
    Var var2,
    double coeff)
```

**Arguments**

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

### MQExpression.Clone()

Clone MQExpression object.

**Synopsis**

```
MQExpression Clone()
```

**Return**

new MQExpression object.

### MQExpression.Evaluate()

evaluate MQExpression object after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of MQExpression object.

### MQExpression.GetConstant()

Get constant in expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in expression.

#### MQExpression.Represent()

String representation of MQExpression object.

**Synopsis**

```
string Represent(uint maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

### 26.2.62 MQuadExpr

COPT multi-dimensional quadratic expression object.It is used to construct multi-dimensional quadratic expressions and perform operations with the multi-dimensional array built in COPT *NdArray* . Its elements are *MQExpression* objects. It can be created by quadratic combination of *MVar* objects. The following methods are provided:

#### MQuadExpr.AddConstant()

Add constant to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### MQuadExpr.AddConstant()

Add constants to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddConstant(NdArray<double> constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

#### MQuadExpr.AddExpr()

Add a linear expression to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression object.

`mult`: the multiplier of linear expression, default value is 1.0.

### MQuadExpr.AddMExpr()

Add MExpression to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddMExpr(MExpression expr, double mult)
```

**Arguments**

`expr`: MExpression object.

`mult`: the multiplier of MExpression, default value is 1.0.

### MQuadExpr.AddMLinExpr()

Add linear expressions to MQuadExpr object.

**Synopsis**

```
void AddMLinExpr(MLinExpr exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MLinExpr object.

`mult`: the same multiplier for added linear expressions, default value is 1.0.

### MQuadExpr.AddMQExpr()

Add MQExpression to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddMQExpr(MQExpression expr, double mult)
```

**Arguments**

`expr`: MQExpression object.

`mult`: the multiplier of MQExpression, default value is 1.0.

### MQuadExpr.AddMQuadExpr()

Add quadratic expressions to MQuadExpr object.

**Synopsis**

```
void AddMQuadExpr(MQuadExpr exprs, double mult)
```

**Arguments**

`exprs`: N-dimension MQuadExpr object.

`mult`: the same multiplier for added quadratic expressions, default value is 1.0.

### MQuadExpr.AddQuadExpr()

Add a quadratic expression to each quadratic expression in MQuadExpr object.

**Synopsis**

```
void AddQuadExpr(QuadExpr expr, double mult)
```

**Arguments**

`expr`: quadratic expression object.

`mult`: the multiplier of quadratic expression, default value is 1.0.

### MQuadExpr.AddTerm()

Add a linear term to MQuadExpr object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new term.

`coeff`: coefficient of new term.

### MQuadExpr.AddTerm()

Add a quadratic term to MQuadExpr object.

**Synopsis**

```
void AddTerm(
    Var var1,
    Var var2,
    double coeff)
```

**Arguments**

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

### MQuadExpr.AddTerms()

Add terms to quadratic expressions in MQuadExpr object.

**Synopsis**

```
void AddTerms(MVar vars, double mult)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`mult`: the same coefficient for added terms, default value 1.0.

### MQuadExpr.AddTerms()

Add terms to quadratic expressions in MQuadExpr object.

**Synopsis**

```
void AddTerms(MVar vars, NdArray<double> coeffs)
```

**Arguments**

`vars`: N-dimension MVar object for added terms.

`coeffs`: N-dimension NdArray object of coefficients for added terms.

### MQuadExpr.Clear()

Clear MQuadExpr object.

**Synopsis**

```
void Clear()
```

### MQuadExpr.Clone()

Clone MQuadExpr object.

**Synopsis**

```
MQuadExpr Clone()
```

**Return**

new MQuadExpr object.

### MQuadExpr.Diagonal()

Get diagonals of MQuadExpr object.

**Synopsis**

```
MQuadExpr Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of MQuadExpr.

`axis2`: 2nd axis of MQuadExpr.

**Return**

(N-1)-dimensional diagonals.

### MQuadExpr.Evaluate()

Evaluate MQuadExpr object after solving.

**Synopsis**

```
NdArray<double> Evaluate()
```

**Return**

NdArray object storing value of each quadratic expression.

### MQuadExpr.Expand()

Expand shape of MQuadExpr object.

**Synopsis**

```
MQuadExpr Expand(int axis)
```

**Arguments**

`axis`: axis of MQuadExpr.

**Return**

MQuadExpr object of (N+1)-dimensional shape.

### MQuadExpr.Flatten()

Flatten a MQuadExpr object to a 1-dimensional shape.

**Synopsis**

```
MQuadExpr Flatten()
```

**Return**

a MQuadExpr object collapsed into one dimension.

### MQuadExpr.GetDim()

Get i-th dimension of MQuadExpr object.

**Synopsis**

```
uint GetDim(int i)
```

**Arguments**

`i`: index of dimension

**Return**

i-th dimension.

### MQuadExpr.GetItem()

Get quadratic expression of given index from MQuadExpr object.

**Synopsis**

```
MQExpression GetItem(long idx)
```

**Arguments**

`idx`: index of quadratic expression.

**Return**

MQExpression object.

### MQuadExpr.GetItem()

Get sub-arrays of MQuadExpr object, given view object.

**Synopsis**

```
MQuadExpr GetItem(View view)
```

**Arguments**

`view`: view of multi-dimensional array.

**Return**

sub-arrays of MQuadExpr object.

### MQuadExpr.GetND()

Get number of dimensions of MQuadExpr object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### MQuadExpr.GetShape()

Get shape of MQuadExpr object.

**Synopsis**

```
Shape GetShape()
```

**Return**

shape object.

### MQuadExpr.GetSize()

Get size of MQuadExpr object.

**Synopsis**

```
uint GetSize()
```

**Return**

number of linear expressions.

### MQuadExpr.HStack<T>()

Stack with other NdArray object along horizontal axis.

**Synopsis**

```
MQuadExpr HStack<T>(NdArray<T> other)
```

**Arguments**

`other`: a NdArray object.

**Return**

the result MQuadExpr object.

### MQuadExpr.HStack()

Stack with other MQuadExpr object along horizontal axis.

**Synopsis**

```
MQuadExpr HStack(MQuadExpr other)
```

**Arguments**

`other`: a MQuadExpr object.

**Return**

the result MQuadExpr object.

### MQuadExpr.HStack()

Stack with other MLinExpr object along horizontal axis.

**Synopsis**

```
MQuadExpr HStack(MLinExpr other)
```

**Arguments**

`other`: a MLinExpr object.

**Return**

the result MQuadExpr object.

### MQuadExpr.HStack()

Stack with other MVar object along horizontal axis.

**Synopsis**

```
MQuadExpr HStack(MVar other)
```

**Arguments**

`other`: a MVar object.

**Return**

the result MQuadExpr object.

### MQuadExpr.Pick()

Given a list of indexes, get quadratic expressions from MQuadExpr object.

**Synopsis**

```
MQuadExpr Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. If two dimensional, each row is position of an element.

**Return**

one-dimensional array of desired quadratic expressions.

### MQuadExpr.Repeat()

Repeat each element of MQuadExpr along given axis.

**Synopsis**

```
MQuadExpr Repeat(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of MQuadExpr.

**Return**

new MQuadExpr object.

### MQuadExpr.RepeatBlock()

Repeat an MQuadExpr a number of times along given axis.

**Synopsis**

```
MQuadExpr RepeatBlock(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of MQuadExpr.

**Return**

new MQuadExpr object.

### MQuadExpr.Represent()

String representation of MQuadExpr object.

**Synopsis**

```
string Represent(int maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

string object.

### MQuadExpr.Reshape()

Reshape MQuadExpr object to new shape.

**Synopsis**

```
MQuadExpr Reshape(Shape shp)
```

**Arguments**

`shp`: new shape of M-dimensions.

**Return**

M-dimensional MQuadExpr object.

### MQuadExpr.SetItem()

Set quadratic expression of given index to MQuadExpr object.

**Synopsis**

```
void SetItem(long idx, MQExpression expr)
```

**Arguments**

`idx`: index of element.

`expr`: MQExpression object.

### MQuadExpr.Squeeze()

Remove axis of length 1 from shape of MQuadExpr object.

**Synopsis**

```
MQuadExpr Squeeze(int axis)
```

**Arguments**

`axis`: axis of MQuadExpr, where the length is 1.

**Return**

MQuadExpr object of (N-1)-dimensional shape.

### MQuadExpr.Stack<T>()

Stack with other NdArray object along given axis.

**Synopsis**

```
MQuadExpr Stack<T>(NdArray<T> other, int axis)
```

**Arguments**

`other`: a NdArray object.

`axis`: an axis of MQuadExpr.

**Return**

the result MQuadExpr object.

### MQuadExpr.Stack()

Stack with other MQuadExpr object along given axis.

**Synopsis**

```
MQuadExpr Stack(MQuadExpr other, int axis)
```

**Arguments**

`other`: a MQuadExpr object.

`axis`: an axis of MQuadExpr.

**Return**

the result MQuadExpr object.

### MQuadExpr.Stack()

Stack with other MLinExpr object along given axis.

**Synopsis**

```
MQuadExpr Stack(MLinExpr other, int axis)
```

**Arguments**

`other`: a MLinExpr object.

`axis`: an axis of MQuadExpr.

**Return**

the result MQuadExpr object.

### MQuadExpr.Stack()

Stack with other MQuadExpr object along given axis.

**Synopsis**

```
MQuadExpr Stack(MVar other, int axis)
```

**Arguments**

`other`: a MVar object.

`axis`: an axis of MQuadExpr.

**Return**

the result MQuadExpr object.

### MQuadExpr.SubConstant()

Subtract constants from each quadratic expression in MQuadExpr object.

**Synopsis**

```
void SubConstant(NdArray<double> constants)
```

**Arguments**

`constants`: N-dimension NdArray object.

### MQuadExpr.Sum()

Sum of all expressions in MQuadExpr object.

**Synopsis**

```
MQuadExpr Sum()
```

**Return**

sum in zero dimension.

### MQuadExpr.Sum()

Sum of variables at given axis of MQuadExpr object.

**Synopsis**

```
MQuadExpr Sum(int axis)
```

**Arguments**

`axis`: axis of MQuadExpr.

**Return**

MQuadExpr object in (N-1)-dimension.

### MQuadExpr.Transpose()

Perform matrix transpose of MQuadExpr object.

**Synopsis**

```
MQuadExpr Transpose()
```

**Return**

transposed MQuadExpr object.

#### MQuadExpr.VStack<T>()

Stack with other NdArray object along vertical axis.

**Synopsis**

```
MQuadExpr VStack<T>(NdArray<T> other)
```

**Arguments**

`other`: a NdArray object.

**Return**

the result MQuadExpr object.

#### MQuadExpr.VStack()

Stack with other MQuadExpr object along vertical axis.

**Synopsis**

```
MQuadExpr VStack(MQuadExpr other)
```

**Arguments**

`other`: a MQuadExpr object.

**Return**

the result MQuadExpr object.

#### MQuadExpr.VStack()

Stack with other MLinExpr object along vertical axis.

**Synopsis**

```
MQuadExpr VStack(MLinExpr other)
```

**Arguments**

`other`: a MLinExpr object.

**Return**

the result MQuadExpr object.

#### MQuadExpr.VStack()

Stack with other MVar object along vertical axis.

**Synopsis**

```
MQuadExpr VStack(MVar other)
```

**Arguments**

`other`: a MVar object.

**Return**

the result MQuadExpr object.

### 26.2.63 NdArray

The NdArray class is a built-in multi-dimensional array in COPT. It represents a table of elements of the same type, indexed by a tuple of integers. The following methods are provided:

#### NdArray<T>.Clone()

Clone NdArray object.

**Synopsis**

```
NdArray<T> Clone()
```

**Return**

new NdArray object with its own copy of the data.

#### NdArray<T>.Diagonal()

Get data type of NdArray object. Get diagonals of NdArray object.

**Synopsis**

```
NdArray<T> Diagonal(
    int offset,
    int axis1,
    int axis2)
```

**Arguments**

`offset`: offset of the diagonal from the main diagonal. Can be positive or negative.

`axis1`: 1st axis of NdArray.

`axis2`: 2nd axis of NdArray.

**Return**

data type of elements. (N-1)-dimensional diagonals.

#### NdArray<T>.Dot()

Dot product with another NdArray object of type double.

**Synopsis**

```
double Dot(NdArray<double> other)
```

**Arguments**

`other`: another NdArray object of type double.

**Return**

dot product value.

### NdArray<T>.Dot()

Dot product with another NdArray object of type int.

**Synopsis**

```
T Dot(NdArray<int> other)
```

**Arguments**

`other`: another NdArray object of type int.

**Return**

dot product value.

### NdArray<T>.Expand()

Expand shape of NdArray object.

**Synopsis**

```
NdArray<T> Expand(int axis)
```

**Arguments**

`axis`: axis of NdArray.

**Return**

NdArray object in (N+1)-dimensions.

### NdArray<T>.Fill()

Fill NdArray object with given value.

**Synopsis**

```
void Fill(T val)
```

**Arguments**

`val`: new value.

### NdArray<T>.Flatten()

Flatten an NdArray object to a 1-dimensional shape.

**Synopsis**

```
NdArray<T> Flatten()
```

**Return**

An NdArray object collapsed into one dimension.

### NdArray<T>.GetDim()

Get i-th dimension in NdArray object.

**Synopsis**

```
long GetDim(int i)
```

**Arguments**

`i`: index of dimensions.

**Return**

the i-th dimension.

### NdArray<T>.GetItem()

Get element of given index from NdArray object.

**Synopsis**

```
T GetItem(long idx)
```

**Arguments**

`idx`: index of element.

**Return**

value of element.

### NdArray<T>.GetItem()

Get sub-array of NdArray object, given View object.

**Synopsis**

```
NdArray<T> GetItem(View view)
```

**Arguments**

`view`: View object.

**Return**

sub NdArray without copying underlying data.

### NdArray<T>.GetND()

Get number of dimensions in NdArray object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### NdArray<T>.GetShape()

Get shape of NdArray object.

**Synopsis**

```
Shape GetShape()
```

**Return**

shape object.

### NdArray<T>.GetSize()

Get size of NdArray object.

**Synopsis**

```
long GetSize()
```

**Return**

size of NdArray.

### NdArray<T>.NdArray()

Constructor of NdArray object.

**Synopsis**

```
NdArray(Shape shp, T val)
```

**Arguments**

`shp`: shape of NdArray object.

`val`: initial value of elements, including int, long, float and double.

### NdArray<T>.Pick()

Given a list of indexes, get elements from NdArray object.

**Synopsis**

```
NdArray<T> Pick(NdArray<int> indexes)
```

**Arguments**

`indexes`: one or two dimensional indexes of elements. if two dimensional, each row is position of an element.

**Return**

one-dimensional array of desired elements.

### NdArray<T>.Prod()

Product of all elements in NdArray object.

**Synopsis**

```
T Prod()
```

**Return**

product value.

### NdArray<T>.Prod()

Product of elements at given axis of NdArray object.

**Synopsis**

```
NdArray<T, N - 1> Prod(int axis)
```

**Arguments**

`axis`: axis of NdArray.

**Return**

(N-1)-dimensional NdArray object.

### NdArray<T>.Repeat()

Repeat each element of an array along given axis.

**Synopsis**

```
NdArray<T> Repeat(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions for each element.

`axis`: axis of NdArray.

**Return**

new NdArray object.

### NdArray<T>.RepeatBlock()

Repeat an array a number of times along given axis.

**Synopsis**

```
NdArray<T> RepeatBlock(long repeats, int axis)
```

**Arguments**

`repeats`: number of repetitions.

`axis`: axis of NdArray.

**Return**

new NdArray object.

### NdArray<T>.Represent()

String representation of NdArray object.

**Synopsis**

```
string Represent(int maxlen)
```

**Arguments**

`maxlen`: maximum buffer length for representations string.

**Return**

representation string object.

### NdArray<T>.Reshape()

Reshape NdArray object to new shape.

**Synopsis**

```
NdArray<T> Reshape(Shape shp)
```

**Arguments**

`shp`: new shape of M-dimensions.

**Return**

M-dimensional NdArray object.

### NdArray<T>.SetItem()

Set element of given index to NdArray object.

**Synopsis**

```
void SetItem(long idx, T val)
```

**Arguments**

`idx`: index of element.

`val`: value of element.

### NdArray<T>.Squeeze()

Remove axis of length 1 from shape of NdArray object.

**Synopsis**

```
NdArray<T> Squeeze(int axis)
```

**Arguments**

`axis`: axis of NdArray, where the length is 1.

**Return**

(N-1)-dimensional NdArray object.

#### NdArray<T>.Sum()

Sum of all elements in NdArray object.

**Synopsis**

```
T Sum()
```

**Return**

sum value.

#### NdArray<T>.Sum()

Sum of elements at given axis of NdArray object.

**Synopsis**

```
NdArray<T, N - 1> Sum(int axis)
```

**Arguments**

`axis`: axis of NdArray.

**Return**

(N-1)-dimensional NdArray object.

#### NdArray<T>.Transpose()

Perform matrix transpose of NdArray object.

**Synopsis**

```
NdArray<T> Transpose()
```

**Return**

transposed NdArray object.

### 26.2.64 Shape

The Shape class encapsulates a tuple of integers, indicating the size of array along each dimension. It refers to dimensions of built-in *NdArray* in COPT. The following methods are provided:

#### Shape.Shape()

Constructor of Shape object.

**Synopsis**

```
Shape()
```

### Shape.Expand()

Expand shape of Shape object.

**Synopsis**

```
Shape Expand(int axis)
```

**Arguments**

`axis`: given axis.

**Return**

Shape object in (N+1)-dimensions.

### Shape.GetDim()

Get i-th dimension in Shape object.

**Synopsis**

```
long GetDim(int i)
```

**Arguments**

`i`: index of dimensions.

**Return**

the i-th dimension.

### Shape.GetND()

Get number of dimensions in Shape object.

**Synopsis**

```
int GetND()
```

**Return**

number of dimensions.

### Shape.GetSize()

Get size of Shape object.

**Synopsis**

```
long GetSize()
```

**Return**

size of shape.

### Shape.GetStart()

Get the i-th start position in Shape object.

**Synopsis**

```
int GetStart(int i)
```

**Arguments**

`i`: index of dimensions.

**Return**

start position in i-th dimension.

### Shape.GetStride()

Get i-th stride in Shape object.

**Synopsis**

```
int GetStride(int i)
```

**Arguments**

`i`: index of dimensions.

**Return**

stride in i-th dimension.

### Shape.Rebuild()

Rebuild Shape object, that is, keep dimensions while reset strides and starts.

**Synopsis**

```
Shape Rebuild()
```

**Return**

new Shape object.

### Shape.Represent()

String representation of Shape object.

**Synopsis**

```
string Represent(int type)
```

**Arguments**

`type`: 0: dimensions; 1: strides; 2: starts.

**Return**

string object.

#### Shape.Squeeze()

Remove axis of length 1 from Shape object.

**Synopsis**

```
Shape Squeeze(int axis)
```

**Arguments**

`axis`: given axis, where the length is 1.

**Return**

Shape object in (N-1)-dimensions.

### 26.2.65 View

The View class is used to perform slicing operations on multi-dimensional arrays. The following methods are provided:

#### View.View()

Constructor of View object.

**Synopsis**

```
View()
```

#### View.AddFull()

Create full view object at current dimension.

**Synopsis**

```
View AddFull()
```

**Return**

View object.

#### View.AddScalar()

Create View object of given index at current dimension.

**Synopsis**

```
View AddScalar(long n)
```

**Arguments**

`n`: given index.

**Return**

View object.

### View.AddSlice()

Create view object of slice at current dimension.

**Synopsis**

```
View AddSlice(long start)
```

**Arguments**

`start`: start index, inclusive.

**Return**

View object.

### View.AddSlice()

Create view object of slice at current dimension.

**Synopsis**

```
View AddSlice(long start, long stop)
```

**Arguments**

`start`: start index, inclusive.

`stop`: stop index, exclusive.

**Return**

View object.

### View.AddSlice()

Create view object of slice at current dimension.

**Synopsis**

```
View AddSlice(
    long start,
    long stop,
    long step,
    int flag)
```

**Arguments**

`start`: start index, inclusive.

`stop`: stop index, exclusive.

`step`: step size between start and stop index. It can be negative.

`flag`: optional, flag for slicing type. Default is 0.

**Return**

View object.

### 26.2.66 NlExpr Class

COPT nonlinear expression object. The `NlExpr` class represents nonlinear expressions in COPT. The nonlinear expressions are used to build nonlinear constraints. The following methods are provided:

#### NlExpr.NlExpr()

Constructor of a nonlinear expression with a constant.

**Synopsis**

```
NlExpr(double constant)
```

**Arguments**

`constant`: constant value in nonlinear expression object.

#### NlExpr.NlExpr()

Constructor of a nonlinear expression with one linear term.

**Synopsis**

```
NlExpr(Var var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: optional, coefficient for the added term.

#### NlExpr.NlExpr()

Constructor of a nonlinear expression with a linear expression.

**Synopsis**

```
NlExpr(Expr expr)
```

**Arguments**

`expr`: linear expression.

#### NlExpr.NlExpr()

Constructor of a nonlinear expression with a quadratic expression.

**Synopsis**

```
NlExpr(QuadExpr expr)
```

**Arguments**

`expr`: quadratic expression.

### NlExpr.AddConstant()

Add constant to the nonlinear expression.

**Synopsis**

```
void AddConstant(double constant)
```

**Arguments**

`constant`: value to be added.

### NlExpr.AddLinExpr()

Add a linear expression to self.

**Synopsis**

```
void AddLinExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### NlExpr.AddNlExpr()

Add a nonlinear expression to self.

**Synopsis**

```
void AddNlExpr(NlExpr expr, double mult)
```

**Arguments**

`expr`: nonlinear expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### NlExpr.AddQuadExpr()

Add a quadratic expression to self.

**Synopsis**

```
void AddQuadExpr(QuadExpr expr, double mult)
```

**Arguments**

`expr`: quadratic expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

### NlExpr.AddTerm()

Add a linear term to nonlinear expression object.

**Synopsis**

```
void AddTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new linear term.

`coeff`: optional, coefficient of new linear term.

### NlExpr.AddTerms()

Add linear terms to nonlinear expression object.

**Synopsis**

```
void AddTerms(VarArray vars, double[] coeffs)
```

**Arguments**

`vars`: variables for added linear terms.

`coeffs`: coefficient array for added linear terms.

### NlExpr.AddTerms()

Add linear terms to nonlinear expression object.

**Synopsis**

```
void AddTerms(Var[] vars, double[] coeffs)
```

**Arguments**

`vars`: variable array for added linear terms.

`coeffs`: coefficient array for added linear terms.

### NlExpr.Clear()

Clear nonlinear expression object.

**Synopsis**

```
void Clear()
```

### NlExpr.Clone()

Deep copy nonlinear expression object.

**Synopsis**

```
NlExpr Clone()
```

**Return**

cloned nonlinear expression object.

### NlExpr.Divide()

Divide itself by an expression.

**Synopsis**

```
void Divide(NlExpr expr)
```

**Arguments**

`expr`: expression operand, including NlExpr, QuadExpr, Expr, Var and constant.

### NlExpr.Evaluate()

Evaluate nonlinear expression after solving.

**Synopsis**

```
double Evaluate()
```

**Return**

value of nonlinear expression.

### NlExpr.GetConstant()

Get constant in nonlinear expression.

**Synopsis**

```
double GetConstant()
```

**Return**

constant in nonlinear expression.

### NlExpr.GetLinExpr()

Get linear expression of nonlinear expression.

**Synopsis**

```
Expr GetLinExpr()
```

**Return**

linear expression object.

### NlExpr.Multiply()

Multiply itself by an expression.

**Synopsis**

```
void Multiply(NlExpr expr)
```

**Arguments**

`expr`: expression operand, including NlExpr, QuadExpr, Expr, Var and constant.

### NlExpr.Negate()

Negate itself.

**Synopsis**

```
void Negate()
```

#### NlExpr.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: capacity of nonlinear constraint objects.

#### NlExpr.SetConstant()

Set constant for the nonlinear expression.

**Synopsis**

```
void SetConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### NlExpr.Size()

Get size of tokens in nonlinear expression.

**Synopsis**

```
long Size()
```

**Return**

size of none-linear tokens.

### 26.2.67 NlConstraint Class

COPT nonlinear constraint object. The `NlConstraint` object is always associated with a particular model. User creates a `NlConstraint` object by adding a nonlinear constraint to model, rather than by constructor of `NlConstraint` class.

#### NlConstraint.Get()

Get information value of the nonlinear constraint. Support information of "LB", "UB" and "Slack".

**Synopsis**

```
double Get(string info)
```

**Arguments**

`info`: name of the information being queried.

**Return**

value of information.

### NlConstraint.GetIdx()

Get index of nonlinear constraint.

**Synopsis**

```
int GetIdx()
```

**Return**

the index of nonlinear constraint.

### NlConstraint.GetName()

Get name of nonlinear constraint.

**Synopsis**

```
string GetName()
```

**Return**

the name of nonlinear constraint.

### NlConstraint.Remove()

Remove this nonlinear constraint from model.

**Synopsis**

```
void Remove()
```

### NlConstraint.Set()

Set information value of nonlinear constraint. Support information of "LB" and "UB".

**Synopsis**

```
void Set(string info, double val)
```

**Arguments**

`info`: name of the information.

`val`: new information value.

### NlConstraint.SetName()

Set name for nonlinear constraint.

**Synopsis**

```
void SetName(string name)
```

**Arguments**

`name`: the name to set.

### 26.2.68 NlConstrArray Class

COPT nonlinear constraint array object. To store and access a set of *NlConstraint Class* objects, Cardinal Optimizer provides NlConstrArray class, which defines the following methods.

#### NlConstrArray.NlConstrArray()

Constructor of NlConstrArray object.

**Synopsis**

```
NlConstrArray()
```

#### NlConstrArray.GetNlConstr()

Get idx-th nonlinear constraint object.

**Synopsis**

```
NlConstraint GetNlConstr(int idx)
```

**Arguments**

`idx`: index of the nonlinear constraint.

**Return**

nonlinear constraint object with index value.

#### NlConstrArray.PushBack()

Add a nonlinear constraint to nonlinear constraint array.

**Synopsis**

```
void PushBack(NlConstraint constr)
```

**Arguments**

`constr`: nonlinear constraint object.

#### NlConstrArray.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: capacity of nonlinear constraint objects.

#### NlConstrArray.Size()

Get the number of nonlinear constraint objects.

**Synopsis**

```
int Size()
```

**Return**

number of nonlinear constraint objects.

### 26.2.69 NlConstrBuilder Class

COPT nonlinear constraint builder object. To help building a nonlinear constraint, given a nonlinear expression, constraint sense and right-hand side value, Cardinal Optimizer provides NlConstrBuilder class, which defines the following methods.

#### NlConstrBuilder.NlConstrBuilder()

Constructor of NlConstrBuilder object.

**Synopsis**

```
NlConstrBuilder()
```

#### NlConstrBuilder.GetNlExpr()

Get nonlinear expression associated with constraint.

**Synopsis**

```
NlExpr GetNlExpr()
```

**Return**

nonlinear expression object.

#### NlConstrBuilder.GetRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double GetRange()
```

**Return**

length from lower bound to upper bound of nonlinear constraint.

#### NlConstrBuilder.GetSense()

Get sense associated with nonlinear constraint.

**Synopsis**

```
char GetSense()
```

**Return**

nonlinear constraint sense.

#### NlConstrBuilder.Set()

Set detail of a nonlinear constraint to its builder object.

**Synopsis**

```
void Set(
    NlExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: nonlinear expression object at one side of nonlinear constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of nonlinear constraint.

#### NlConstrBuilder.SetRange()

Set a range constraint to nonlinear constraint builder.

**Synopsis**

```
void SetRange(NlExpr expr, double range)
```

**Arguments**

`expr`: nonlinear expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of nonlinear constraint. Must be greater than 0.

### 26.2.70 NlConstrBuilderArray Class

COPT nonlinear constraint builder array object. To store and access a set of *NlConstrBuilder Class* objects, Cardinal Optimizer provides `NlConstrBuilderArray` class, which defines the following methods.

#### NlConstrBuilderArray.NlConstrBuilderArray()

Constructor of NlConstrBuilderArray object.

**Synopsis**

```
NlConstrBuilderArray()
```

#### NlConstrBuilderArray.GetBuilder()

Get idx-th nonlinear constraint builder object.

**Synopsis**

```
NlConstrBuilder GetBuilder(int idx)
```

**Arguments**

`idx`: index of the nonlinear constraint builder.

**Return**

nonlinear constraint builder object with index idx.

#### NlConstrBuilderArray.PushBack()

Add a nonlinear constraint builder object to nonlinear constraint builder array.

**Synopsis**

```
void PushBack(NlConstrBuilder builder)
```

**Arguments**

`builder`: a nonlinear constraint builder object.

#### NlConstrBuilderArray.Reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void Reserve(int n)
```

**Arguments**

`n`: capacity of nonlinear constraint objects.

#### NlConstrBuilderArray.Size()

Get the number of nonlinear constraint builder objects.

**Synopsis**

```
int Size()
```

**Return**

number of nonlinear constraint builder objects.

### 26.2.71 NL Namespace

Common nonlinear functions in the `NL` namespace are provided for constructing nonlinear expressions. The following methods are provided:

#### NL.Abs()

Calculate absolute value of a nonlinear expression.

**Synopsis**

```
static NlExpr Abs(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.ACos()

Calculate arccosine of a nonlinear expression.

**Synopsis**

```
static NlExpr ACos(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.ACosH()

Calculate inverse hyperbolic cosine of a nonlinear expression.

**Synopsis**

```
static NlExpr ACosH(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.ASin()

Calculate arcsine of a nonlinear expression.

**Synopsis**

```
static NlExpr ASin(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.ASinH()

Calculate inverse hyperbolic sine of a nonlinear expression.

**Synopsis**

```
static NlExpr ASinH(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.ATan()

Calculate arctangent of a nonlinear expression.

**Synopsis**

```
static NlExpr ATan(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.ATan2()

Calculate two-argument arctangent of a nonlinear expression.

**Synopsis**

```
static NlExpr ATan2(NlExpr y, NlExpr x)
```

**Arguments**

`y`: y coordinate as a nonlinear expression.

`x`: x coordinate as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.ATanH()

Calculate inverse hyperbolic tangent of a nonlinear expression.

**Synopsis**

```
static NlExpr ATanH(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Ceil()

Calculate ceiling value of a nonlinear expression.

**Synopsis**

```
static NlExpr Ceil(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Cos()

Calculate cosine of a nonlinear expression.

**Synopsis**

```
static NlExpr Cos(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.CosH()

Calculate hyperbolic cosine of a nonlinear expression.

**Synopsis**

```
static NlExpr CosH(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Exp()

Calculate exponential function of a nonlinear expression.

**Synopsis**

```
static NlExpr Exp(NlExpr expo)
```

**Arguments**

`expo`: exponent as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Floor()

Calculate floor value of a nonlinear expression.

**Synopsis**

```
static NlExpr Floor(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Log10()

Calculate logarithmic function of a nonlinear expression with base 10.

**Synopsis**

```
static NlExpr Log10(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Log()

Calculate nature logarithmic function of a nonlinear expression.

**Synopsis**

```
static NlExpr Log(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Neg()

Calculate negative value of a nonlinear expression.

**Synopsis**

```
static NlExpr Neg(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Pow()

Calculate power function of a nonlinear expression.

**Synopsis**

```
static NlExpr Pow(NlExpr bas, NlExpr expo)
```

**Arguments**

`bas`: base as a nonlinear expression.

`expo`: exponent as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Sin()

Calculate sine of a nonlinear expression.

**Synopsis**

```
static NlExpr Sin(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.SinH()

Calculate hyperbolic sine of a nonlinear expression.

**Synopsis**

```
static NlExpr SinH(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Sqrt()

Calculate square root of a nonlinear expression.

**Synopsis**

```
static NlExpr Sqrt(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Sum()

Sum of nonlinear expressions.

**Synopsis**

```
static NlExpr Sum(
    NlExpr op1,
    NlExpr op2,
    NlExpr op3,
    NlExpr op4)
```

**Arguments**

`op1`: first nonlinear expression.

`op2`: second nonlinear expression.

`op3`: third nonlinear expression.

`op4`: fourth nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Sum()

Sum of nonlinear expressions.

**Synopsis**

```
static NlExpr Sum(
    NlExpr op1,
    NlExpr op2,
    NlExpr op3)
```

**Arguments**

`op1`: first nonlinear expression.

`op2`: second nonlinear expression.

`op3`: third nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.Sum()

Sum of nonlinear expressions.

**Synopsis**

```
static NlExpr Sum(NlExpr[] exprs)
```

**Arguments**

`exprs`: array of nonlinear expressions.

**Return**

result as a nonlinear expression.

### NL.Tan()

Calculate tangent of a nonlinear expression.

**Synopsis**

```
static NlExpr Tan(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

#### NL.TanH()

Calculate hyperbolic tangent of a nonlinear expression.

**Synopsis**

```
static NlExpr TanH(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### 26.2.72 CallbackBase

COPT Callback abstract base object. Users must implement its virtual method `virtual void CallbackBase::callback()` to instantiate an instance, which pass to `Model::SetCallback(CallbackBase cb, int cbctx)` as the first parameter. Subclass of CallbackBase inherits the following member methods:

#### CallbackBase.CallbackBase()

Constructor of CallbackBase, implementing ICallback interface.

**Synopsis**

```
CallbackBase()
```

#### CallbackBase.AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(
    Expr lhs,
    char sense,
    double rhs)
```

**Arguments**

`lhs`: expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

#### CallbackBase.AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(
    Expr lhs,
    char sense,
    Expr rhs)
```

**Arguments**

`lhs`: left hand side expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy constraint.

### CallbackBase.AddLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void AddLazyConstr(ConstrBuilder builder)
```

**Arguments**

`builder`: builder for lazy constraint.

### CallbackBase.AddLazyConstrs()

Add lazy constraints to model.

**Synopsis**

```
void AddLazyConstrs(ConstrBuilderArray builders)
```

**Arguments**

`builders`: array of builders for lazy constraints.

### CallbackBase.AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(
    Expr lhs,
    char sense,
    double rhs)
```

**Arguments**

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

### CallbackBase.AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(
    Expr lhs,
    char sense,
    Expr rhs)
```

**Arguments**

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

### CallbackBase.AddUserCut()

Add a user cut to model.

**Synopsis**

```
void AddUserCut(ConstrBuilder builder)
```

**Arguments**

`builder`: builder for user cut.

### CallbackBase.AddUserCuts()

Add user cuts to model.

**Synopsis**

```
void AddUserCuts(ConstrBuilderArray builders)
```

**Arguments**

`builders`: array of builders for user cuts.

### CallbackBase.callback()

Pure virtual function defined in ICallback interface. User must implement it.

**Synopsis**

```
void callback()
```

### CallbackBase.GetDblInfo()

Get double value of given information name in callback.

**Synopsis**

```
double GetDblInfo(string cbinfo)
```

**Arguments**

`cbinfo`: name of callback info.

**Return**

value of desired information.

### CallbackBase.GetIncumbent()

Get best feasible solution of given variable in callback.

**Synopsis**

```
double GetIncumbent(Var var)
```

**Arguments**

`var`: given variable.

**Return**

best feasible solution of given variable.

### CallbackBase.GetIncumbent()

Get best feasible solution of variables in callback.

**Synopsis**

```
double[] GetIncumbent(VarArray vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

best feasible solution of desired variables.

### CallbackBase.GetIncumbent()

Get best feasible solution of variables in callback.

**Synopsis**

```
double[] GetIncumbent(Var[] vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

best feasible solution of desired variables.

### CallbackBase.GetIncumbent()

Get best feasible solution of all variables in callback.

**Synopsis**

```
double[] GetIncumbent()
```

**Return**

best feasible solution of all variables.

### CallbackBase.GetIntInfo()

Get integer value of given information name in callback.

**Synopsis**

```
int GetIntInfo(string cbinfo)
```

**Arguments**

`cbinfo`: name of callback info.

**Return**

value of desired information.

### CallbackBase.GetRelaxSol()

Get LP-relaxation solution of given variable in callback.

**Synopsis**

```
double GetRelaxSol(Var var)
```

**Arguments**

`var`: given variable.

**Return**

LP-relaxation solution of given variable.

### CallbackBase.GetRelaxSol()

Get LP-relaxation solution of variables in callback.

**Synopsis**

```
double[] GetRelaxSol(VarArray vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

LP-relaxation solution of variables.

### CallbackBase.GetRelaxSol()

Get LP-relaxation solution of variables in callback.

**Synopsis**

```
double[] GetRelaxSol(Var[] vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

LP-relaxation solution of variables.

### CallbackBase.GetRelaxSol()

Get LP-relaxation solution of all variables in callback.

**Synopsis**

```
double[] GetRelaxSol()
```

**Return**

LP-relaxation solution of all variables.

### CallbackBase.GetSolution()

Get solution of given variable in callback.

**Synopsis**

```
double GetSolution(Var var)
```

**Arguments**

`var`: given variable.

**Return**

solution of given variable.

### CallbackBase.GetSolution()

Get solution of variables in callback.

**Synopsis**

```
double[] GetSolution(VarArray vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

solution of variables.

### CallbackBase.GetSolution()

Get solution of variables in callback.

**Synopsis**

```
double[] GetSolution(Var[] vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

solution of variables.

### CallbackBase.GetSolution()

Get solution of all variables in callback.

**Synopsis**

```
double[] GetSolution()
```

**Return**

solution of all variables.

### CallbackBase.Interrupt()

Interrupt solving problems in callback

**Synopsis**

```
void Interrupt()
```

### CallbackBase.LoadSolution()

Load customized solution to model.

**Synopsis**

```
double LoadSolution()
```

**Return**

objective value of given solution.

### CallbackBase.SetSolution()

Set solution of a given variable in callback.

**Synopsis**

```
void SetSolution(Var var, double val)
```

**Arguments**

`var`: a variable object.

`val`: double value.

### CallbackBase.SetSolution()

Set solution of variables in callback.

**Synopsis**

```
void SetSolution(VarArray vars, double[] vals)
```

**Arguments**

`vars`: an array of variable objects.

`vals`: an array of double values.

#### CallbackBase.SetSolution()

Set solution of variables in callback.

**Synopsis**

```
void SetSolution(Var[] vars, double[] vals)
```

**Arguments**

`vars`: an array of variable objects.

`vals`: an array of double values.

#### CallbackBase.Where()

Get context in callback.

**Synopsis**

```
int Where()
```

**Return**

integer value of context.

### 26.2.73 ProbBuffer

Buffer object for COPT problem. ProbBuffer object holds the (MPS) problem in string format.

#### ProbBuffer.ProbBuffer()

Constructor of ProbBuffer object.

**Synopsis**

```
ProbBuffer(int sz)
```

**Arguments**

`sz`: initial size of the problem buffer.

#### ProbBuffer.GetData()

Get string of problem in problem buffer.

**Synopsis**

```
string GetData()
```

**Return**

string of problem in problem buffer.

#### ProbBuffer.Resize()

Resize buffer to given size, and zero-ended

**Synopsis**

```
void Resize(int sz)
```

**Arguments**

`sz`: new buffer size.

#### ProbBuffer.Size()

Get the size of problem buffer.

**Synopsis**

```
int Size()
```

**Return**

size of problem buffer.

### 26.2.74 CoptException

Copt exception object.

#### CoptException.CoptException()

Constructor of COPT Exception class.

**Synopsis**

```
CoptException(int code, string msg)
```

**Arguments**

`code`: error code for exception.

`msg`: error message for exception.

#### CoptException.GetCode()

Get the error code associated with the exception.

**Synopsis**

```
int GetCode()
```

**Return**

the error code.

# Chapter 27

# Java API Reference

The **Cardinal Optimizer** provides a Java API library. This chapter documents all COPT Java constants and API functions for Java applications.

## 27.1 Constants

There are four types of constants defined in **Cardinal Optimizer**. They are general constants, information constants, attributes constants and parameters constants.

### 27.1.1 General Constants

For the contents of Java general constants, see *General Constants*.

General constants are defined in `Consts` class. User may refer to general constants with namespace, that is, `copt.Consts.XXXX`.

### 27.1.2 Attributes

For the contents of Java attribute constants, see *Attributes*.

All COPT Java attributes are defined in `DblAttr` and `IntAttr` classes. User may refer to double attributes by `copt.DblAttr.XXXX`, and integer attributes by `copt.IntAttr.XXXX`.

In the Java API, user can get the attribute value by specifying the attribute name. The two functions of obtaining attribute values are as follows, please refer to *Java API: Model Class* for details.

- `Model.getIntAttr()`: Get value of a COPT integer attribute.
- `Model.getDblAttr()`: Get value of a COPT double attribute.

### 27.1.3 Information

For the content of Java information constants, see *Information*.

In the Java API, information constants are defined in the `DblInfo` class. Users can access information constants through the prefix `copt` in the namespace (usually can be omitted) `copt.DblInfo.`

For instance, `copt.DblInfo.Obj` is the coefficients of variables in the objective function.

### 27.1.4 Callback Information

For the content of Java API callback information class constants, see *Callback Information*.

In the Java API, callback-related information constants are defined in the `CbInfo` class. Users can access information constants through the prefix `copt` in the namespace (usually can be omitted) `copt.CbInfo.`

For instance, `copt.CbInfo.BestObj` is the current best objective.

### 27.1.5 Parameters

For the contents of Java parameters constants, see *Parameters*.

All COPT Java parameters are defined in DblParam and IntParam classes. User may refer to double parameters by `copt.DblParam.XXXX`, and integer parameters by `copt.IntParam.XXXX`.

In the Java API, user can get and set the parameter value by specifying the parameter name. The provided functions are as follows, please refer to *Java API: Model Class* for details.

- Get detailed information of the specified parameter (current value/max/min): `Model.getParamInfo()`
- Get the current value of the specified integer/double parameter: `Model.getIntParam()` / `Model.getDblParam()`
- Set the specified integer/double parameter value: `Model.setIntParam()` / `Model.setDblParam()`

## 27.2 Java Modeling Classes

This chapter documents COPT Java interface. Users may refer to Java classes described below for details of how to construct and solve Java models.

### 27.2.1 Envr

Essentially, any Java application using Cardinal Optimizer should start with a COPT environment. COPT models are always associated with a COPT environment. User must create an environment object before populating models. User generally only need a single environment object in program.

#### Envr.Envr()

Constructor of COPT Envr object.

**Synopsis**

```
Envr()
```

#### Envr.Envr()

Constructor of COPT Envr object, given a license folder.

**Synopsis**

```
Envr(String licDir)
```

**Arguments**

`licDir`: directory having local license or client config file.

### Envr.Envr()

Constructor of COPT Envr object, given an Envr config object.

**Synopsis**

```
Envr(EnvrConfig config)
```

**Arguments**

`config`: Envr config object holding settings for remote connection.

### Envr.bindNumaCpu()

Bind the CPUs for the current process to a NUMA node.

**Synopsis**

```
void bindNumaCpu(int numaNode)
```

**Arguments**

`numaNode`: ID of a NUMA node.

### Envr.bindNumaMem()

Bind memory for the current process to a NUMA node (Linux only).

**Synopsis**

```
void bindNumaMem(int numaNode)
```

**Arguments**

`numaNode`: the ID of a NUMA node.

### Envr.close()

Close remote connection and token becomes invalid for all problems in current environment. To release the environment, call Dispose() for that.

**Synopsis**

```
void close()
```

### Envr.createModel()

Create a model object.

**Synopsis**

```
Model createModel(String name)
```

**Arguments**

`name`: customized model name.

**Return**

a model object.

#### Envr.dispose()

Release this environment reference on the calling thread. Call it (e.g. in a finally block) explicitly on the same thread, instead of relying on the GC finalizer thread. Note the underlying native environment is reference-counted and destroyed when the last referring to it is released.

**Synopsis**

```
void dispose()
```

#### Envr.getCpuAffinity()

Get CPU affinity for the current process, which is saved in an integer array.

**Synopsis**

```
int[] getCpuAffinity()
```

**Return**

an integer array of CPU IDs.

#### Envr.getNumaNodeCount()

Get count of NUMA nodes.

**Synopsis**

```
int getNumaNodeCount()
```

**Return**

count of NUMA nodes.

#### Envr.setCpuAffinity()

Set CPU affinity with given mask string.

**Synopsis**

```
void setCpuAffinity(String hexMask)
```

**Arguments**

`hexMask`: CPU mask string of hexadecimal characters.

### 27.2.2 EnvrConfig

If user connects to COPT remote services, such as floating token server or compute cluster, it is necessary to add config settings with EnvrConfig object.

#### EnvrConfig.EnvrConfig()

Constructor of envr config object.

**Synopsis**

```
EnvrConfig()
```

#### EnvrConfig.set()

Set config settings in terms of name-value pair.

**Synopsis**

```
void set(String name, String value)
```

**Arguments**

`name`: keyword of a config setting.

`value`: value of a config setting.

### 27.2.3 Model

In general, a COPT model consists of a set of variables, a (linear) objective function on these variables, a set of constraints on these variables, etc. COPT model class encapsulates all required methods for constructing a COPT model.

#### Model.Model()

Constructor of model.

**Synopsis**

```
Model(Envr env, String name)
```

**Arguments**

`env`: associated environment object.

`name`: string of model name.

#### Model.addAffineCone()

Add an affine cone constraint to model.

**Synopsis**

```
AffineCone addAffineCone(AffineConeBuilder builder, String name)
```

**Arguments**

`builder`: builder for new affine cone constraint.

`name`: name of new affine cone.

**Return**

new affine cone constraint object.

### Model.addAffineCone()

Add an affine cone constraint to model.

**Synopsis**

```
AffineCone addAffineCone(
    Expr[] exprs,
    int type,
    String name)
```

**Arguments**

`exprs`: linear expressions that participate in the affine cone constraint.

`type`: type of an affine cone constraint.

`name`: name of new affine cone.

**Return**

new affine cone constraint object.

### Model.addAffineCone()

Add an affine cone constraint to model.

**Synopsis**

```
AffineCone addAffineCone(
    PsdExpr[] exprs,
    int type,
    String name)
```

**Arguments**

`exprs`: PSD expressions that participate in the affine cone constraint.

`type`: type of an affine cone constraint.

`name`: name of new affine cone.

**Return**

new affine cone constraint object.

### Model.addCone()

Add a cone constraint to model.

**Synopsis**

```
Cone addCone(
    int dim,
    int type,
    char[] pvtype,
    String prefix)
```

**Arguments**

`dim`: dimension of the cone constraint.

`type`: type of a cone constraint.

`pvtype`: types of variables in the cone.

`prefix`: name prefix of variables in the cone.

**Return**

new cone constraint object.

### Model.addCone()

Add a cone constraint to model.

**Synopsis**

```
Cone addCone(ConeBuilder builder)
```

**Arguments**

`builder`: builder for new cone constraint.

**Return**

new cone constraint object.

### Model.addCone()

Add a cone constraint to model.

**Synopsis**

```
Cone addCone(Var[] vars, int type)
```

**Arguments**

`vars`: variables that participate in the cone constraint.

`type`: type of a cone constraint.

**Return**

new cone constraint object.

### Model.addCone()

Add a cone constraint to model.

**Synopsis**

```
Cone addCone(VarArray vars, int type)
```

**Arguments**

`vars`: variables that participate in the cone constraint.

`type`: type of a cone constraint.

**Return**

new cone constraint object.

### Model.addConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint addConstr(
    Expr expr,
    char sense,
    double rhs,
    String name)
```

**Arguments**

`expr`: expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`name`: name of new constraint.

**Return**

new constraint object.

### Model.addConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint addConstr(
    Expr expr,
    char sense,
    Var var,
    String name)
```

**Arguments**

`expr`: expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`var`: variable for the new constraint.

`name`: name of new constraint.

**Return**

new constraint object.

### Model.addConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint addConstr(
    Expr lhs,
    char sense,
    Expr rhs,
    String name)
```

**Arguments**

`lhs`: left hand side expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side expression for the new constraint.

`name`: name of new constraint.

**Return**

new constraint object.

### Model.addConstr()

Add a linear constraint to model.

**Synopsis**

```
Constraint addConstr(
    Expr expr,
    double lb,
    double ub,
    String name)
```

**Arguments**

`expr`: expression for the new constraint.

`lb`: lower bound for the new constraint.

`ub`: upper bound for the new constraint

`name`: name of new constraint.

**Return**

new constraint object.

### Model.addConstr()

Add a linear constraint to a model.

**Synopsis**

```
Constraint addConstr(ConstrBuilder builder, String name)
```

**Arguments**

`builder`: builder for the new constraint.

`name`: name of new constraint.

**Return**

new constraint object.

### Model.addConstrs()

Add linear constraints to model.

**Synopsis**

```
ConstrArray addConstrs(
    int count,
    char[] senses,
    double[] rhss,
    String prefix)
```

**Arguments**

`count`: number of constraints added to model.

`senses`: sense array for new linear constraints, other than range sense.

`rhss`: right hand side values for new variables.

`prefix`: name prefix for new constraints.

**Return**

array of new constraint objects.

### Model.addConstrs()

Add linear constraints to a model.

**Synopsis**

```
ConstrArray addConstrs(
    int count,
    double[] lbs,
    double[] ubs,
    String prefix)
```

**Arguments**

`count`: number of constraints added to the model.

`lbs`: lower bounds of new constraints.

`ubs`: upper bounds of new constraints.

`prefix`: name prefix for new constraints.

**Return**

array of new constraint objects.

### Model.addConstrs()

Add linear constraints to a model.

**Synopsis**

```
ConstrArray addConstrs(ConstrBuilderArray builders, String prefix)
```

**Arguments**

`builders`: builders for new constraints.

`prefix`: name prefix for new constraints.

**Return**

array of new constraint objects.

### Model.addDenseMat()

Add a dense symmetric matrix to a model.

**Synopsis**

```
SymMatrix addDenseMat(int dim, double[] vals)
```

**Arguments**

`dim`: dimension of the dense symmetric matrix.

`vals`: array of non-zero elements, filled in column-wise up to len or max length of symmetric matrix.

**Return**

new symmetric matrix object.

### Model.addDenseMat()

Add a dense symmetric matrix to a model.

**Synopsis**

```
SymMatrix addDenseMat(int dim, double val)
```

**Arguments**

`dim`: dimension of dense symmetric matrix.

`val`: value to fill dense symmetric matrix.

**Return**

new symmetric matrix object.

### Model.addDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix addDiagMat(int dim, double val)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

**Return**

new diagonal matrix object.

### Model.addDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix addDiagMat(int dim, double[] vals)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`vals`: array of values of diagonal elements.

**Return**

new diagonal matrix object.

### Model.addDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix addDiagMat(
    int dim,
    double val,
    int offset)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

`offset`: shift distance against diagonal line.

**Return**

new diagonal matrix object.

### Model.addDiagMat()

Add a diagonal matrix to a model.

**Synopsis**

```
SymMatrix addDiagMat(
    int dim,
    double[] vals,
    int offset)
```

**Arguments**

`dim`: dimension of diagonal matrix.

`vals`: array of values of diagonal elements.

`offset`: shift distance against diagonal line.

**Return**

new diagonal matrix object.

### Model.addExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone addExpCone(
    int type,
    char[] pvtype,
    String prefix)
```

**Arguments**

`type`: type of an exponential cone constraint.

`pvtype`: types of variables in the exponential cone.

`prefix`: name prefix of variables in the exponential cone.

**Return**

new exponential cone constraint object.

### Model.addExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone addExpCone(ExpConeBuilder builder)
```

**Arguments**

`builder`: builder for new exponential cone constraint.

**Return**

new exponential cone constraint object.

### Model.addExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone addExpCone(Var[] vars, int type)
```

**Arguments**

`vars`: variables that participate in the exponential cone constraint.

`type`: type of an exponential cone constraint.

**Return**

new exponential cone constraint object.

### Model.addExpCone()

Add an exponential cone constraint to model.

**Synopsis**

```
ExpCone addExpCone(VarArray vars, int type)
```

**Arguments**

`vars`: variables that participate in the exponential cone constraint.

`type`: type of an exponential cone constraint.

**Return**

new exponential cone constraint object.

### Model.addEyeMat()

Add an identity matrix to a model.

**Synopsis**

```
SymMatrix addEyeMat(int dim)
```

**Arguments**

`dim`: dimension of identity matrix.

**Return**

new identity matrix object.

### Model.addGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr addGenConstrIndicator(GenConstrBuilder builder, String
name)
```

**Arguments**

`builder`: builder for the general constraint.

`name`: name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model.addGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr addGenConstrIndicator(
    Var binvar,
    int binval,
    ConstrBuilder builder,
    int type,
    String name)
```

**Arguments**

`binvar`: binary indicator variable.

`binval`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`builder`: builder for linear constraint.

`type`: type of general constraint.

`name`: name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model.addGenConstrIndicator()

Add a general constraint of type indicator to model.

**Synopsis**

```
GenConstr addGenConstrIndicator(
    Var binvar,
    int binval,
    Expr expr,
    char sense,
    double rhs,
    int type,
    String name)
```

**Arguments**

`binvar`: binary indicator variable.

`binval`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`expr`: expression for new linear constraint.

`sense`: sense for new linear constraint.

`rhs`: right hand side value for new linear constraint.

`type`: type of general constraint.

`name`: name of new general constraint.

**Return**

new general constraint object of type indicator.

### Model.addGenConstrIndicators()

Add general constraints to a model.

**Synopsis**

```
GenConstrArray addGenConstrIndicators(GenConstrBuilderArray
builders, String prefix)
```

**Arguments**

`builders`: builders for new general constraints.

`prefix`: name prefix for new general constraints.

**Return**

array of new general constraint objects.

### Model.addLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void addLazyConstr(
    Expr lhs,
    char sense,
    double rhs,
    String name)
```

**Arguments**

`lhs`: expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

`name`: name of lazy constraint.

### Model.addLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void addLazyConstr(
    Expr lhs,
    char sense,
    Expr rhs,
    String name)
```

**Arguments**

`lhs`: left hand side expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy constraint.

`name`: name of lazy constraint.

### Model.addLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void addLazyConstr(ConstrBuilder builder, String name)
```

**Arguments**

`builder`: builder for lazy constraint.

`name`: name of lazy constraint.

### Model.addLazyConstrs()

Add lazy constraints to model.

**Synopsis**

```
void addLazyConstrs(ConstrBuilderArray builders, String prefix)
```

**Arguments**

`builders`: array of builders for lazy constraints.

`prefix`: name prefix of new lazy constraints.

### Model.addLmiConstr()

Add an LMI constraint to model.

**Synopsis**

```
LmiConstraint addLmiConstr(LmiExpr expr, String name)
```

**Arguments**

`expr`: LMI expression for new LMI constraint.

`name`: name of new LMI constraint.

**Return**

new LMI constraint object.

### Model.addNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint addNlConstr(
    NlExpr expr,
    char sense,
    double rhs,
```

```
    String name)
```

**Arguments**

`expr`: nonlinear expression for the new constraint.

`sense`: sense for new nonlinear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`name`: name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model.addNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint addNlConstr(
    NlExpr lhs,
    char sense,
    NlExpr rhs,
    String name)
```

**Arguments**

`lhs`: left hand side nonlinear expression for the new constraint.

`sense`: sense for new nonlinear constraint, other than range sense.

`rhs`: right hand side nonlinear expression for the new constraint.

`name`: name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model.addNlConstr()

Add a nonlinear constraint to model.

**Synopsis**

```
NlConstraint addNlConstr(
    NlExpr expr,
    double lb,
    double ub,
    String name)
```

**Arguments**

`expr`: nonlinear expression for the new constraint.

`lb`: lower bound for the new nonlinear constraint.

`ub`: upper bound for the new nonlinear constraint

`name`: name of new constraint.

**Return**

new nonlinear constraint object.

### Model.addNlConstr()

Add a nonlinear constraint to a model.

**Synopsis**

```
NlConstraint addNlConstr(NlConstrBuilder builder, String name)
```

**Arguments**

`builder`: builder for the new nonlinear constraint.

`name`: name of new nonlinear constraint.

**Return**

new nonlinear constraint object.

### Model.addNlConstrs()

Add nonlinear constraints to a model.

**Synopsis**

```
NlConstrArray addNlConstrs(NlConstrBuilderArray builders, String
prefix)
```

**Arguments**

`builders`: builders for new nonlinear constraints.

`prefix`: name prefix for new constraints.

**Return**

array of new nonlinear constraint objects.

### Model.addOnesMat()

Add a dense symmetric matrix of value one to a model.

**Synopsis**

```
SymMatrix addOnesMat(int dim)
```

**Arguments**

`dim`: dimension of dense symmetric matrix.

**Return**

new symmetric matrix object.

### Model.addPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint addPsdConstr(
    PsdExpr expr,
    char sense,
    double rhs,
    String name)
```

**Arguments**

`expr`: PSD expression for new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: double value at right side of the new PSD constraint.

`name`: name of new PSD constraint.

**Return**

new PSD constraint object.

### Model.addPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint addPsdConstr(
    PsdExpr expr,
    double lb,
    double ub,
    String name)
```

**Arguments**

`expr`: expression for new PSD constraint.

`lb`: lower bound for ew PSD constraint.

`ub`: upper bound for new PSD constraint

`name`: name of new PSD constraint.

**Return**

new PSD constraint object.

### Model.addPsdConstr()

Add a PSD constraint to model.

**Synopsis**

```
PsdConstraint addPsdConstr(
    PsdExpr lhs,
    char sense,
    PsdExpr rhs,
    String name)
```

**Arguments**

`lhs`: PSD expression at left side of new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: PSD expression at right side of new PSD constraint.

`name`: name of new PSD constraint.

**Return**

new PSD constraint object.

### Model.addPsdConstr()

Add a PSD constraint to a model.

**Synopsis**

```
PsdConstraint addPsdConstr(PsdConstrBuilder builder, String name)
```

**Arguments**

`builder`: builder for new PSD constraint.

`name`: name of new PSD constraint.

**Return**

new PSD constraint object.

### Model.addPsdVar()

Add a new PSD variable to model.

**Synopsis**

```
PsdVar addPsdVar(int dim, String name)
```

**Arguments**

`dim`: dimension of new PSD variable.

`name`: name of new PSD variable.

**Return**

PSD variable object.

### Model.addPsdVars()

Add new PSD variables to model.

**Synopsis**

```
PsdVarArray addPsdVars(
    int count,
    int[] dims,
    String prefix)
```

**Arguments**

`count`: number of new PSD variables.

`dims`: array of dimensions of new PSD variables.

`prefix`: name prefix of new PSD variables.

**Return**

array of PSD variable objects.

### Model.addQConstr()

Add a quadratic constraint to model.

**Synopsis**

```
QConstraint addQConstr(
    QuadExpr expr,
    char sense,
    double rhs,
    String name)
```

**Arguments**

`expr`: quadratic expression for the new constraint.

`sense`: sense for new quadratic constraint.

`rhs`: double value at right side of the new quadratic constraint.

`name`: name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model.addQConstr()

Add a quadratic constraint to model.

**Synopsis**

```
QConstraint addQConstr(
    QuadExpr lhs,
    char sense,
    QuadExpr rhs,
    String name)
```

**Arguments**

`lhs`: quadratic expression at left side of new quadratic constraint.

`sense`: sense for new quadratic constraint.

`rhs`: quadratic expression at right side of new quadratic constraint.

`name`: name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model.addQConstr()

Add a quadratic constraint to a model.

**Synopsis**

```
QConstraint addQConstr(QConstrBuilder builder, String name)
```

**Arguments**

`builder`: builder for the new quadratic constraint.

`name`: name of new quadratic constraint.

**Return**

new quadratic constraint object.

### Model.addSos()

Add a SOS constraint to model.

**Synopsis**

```
Sos addSos(SosBuilder builder)
```

**Arguments**

`builder`: builder for new SOS constraint.

**Return**

new SOS constraint object.

### Model.addSos()

Add a SOS constraint to model.

**Synopsis**

```
Sos addSos(
    Var[] vars,
    double[] weights,
    int type)
```

**Arguments**

`vars`: variables that participate in the SOS constraint.

`weights`: weights for variables in the SOS constraint.

`type`: type of SOS constraint.

**Return**

new SOS constraint object.

### Model.addSos()

Add a SOS constraint to model.

**Synopsis**

```
Sos addSos(
    VarArray vars,
    double[] weights,
    int type)
```

**Arguments**

`vars`: variables that participate in the SOS constraint.

`weights`: weights for variables in the SOS constraint.

`type`: type of SOS constraint.

**Return**

new SOS constraint object.

### Model.addSparseMat()

Add a sparse symmetric matrix to a model.

**Synopsis**

```
SymMatrix addSparseMat(
    int dim,
    int nElems,
    int[] rows,
    int[] cols,
    double[] vals)
```

**Arguments**

`dim`: dimension of the sparse symmetric matrix.

`nElems`: number of non-zero elements in the sparse symmetric matrix.

`rows`: array of row indexes of non-zero elements.

`cols`: array of col indexes of non-zero elements.

`vals`: array of values of non-zero elements.

**Return**

new symmetric matrix object.

### Model.addSymMat()

Given a symmetric matrix expression, add results matrix to model.

**Synopsis**

```
SymMatrix addSymMat(SymMatExpr expr)
```

**Arguments**

`expr`: symmetric matrix expression object.

**Return**

results symmetric matrix object.

### Model.addUserCut()

Add a user cut to model.

**Synopsis**

```
void addUserCut(
    Expr lhs,
    char sense,
    double rhs,
    String name)
```

**Arguments**

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

`name`: name of user cut.

### Model.addUserCut()

Add a user cut to model.

**Synopsis**

```
void addUserCut(
    Expr lhs,
    char sense,
    Expr rhs,
    String name)
```

**Arguments**

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

`name`: name of user cut.

### Model.addUserCut()

Add a user cut to model.

**Synopsis**

```
void addUserCut(ConstrBuilder builder, String name)
```

**Arguments**

`builder`: builder for user cut.

`name`: name of user cut.

### Model.addUserCuts()

Add user cuts to model.

**Synopsis**

```
void addUserCuts(ConstrBuilderArray builders, String prefix)
```

**Arguments**

`builders`: array of builders for user cuts.

`prefix`: name prefix of new user cuts.

### Model.addVar()

Add a variable and the associated non-zero coefficients as column.

**Synopsis**

```
Var addVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    String name)
```

**Arguments**

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`name`: name for new variable.

**Return**

new variable object.

### Model.addVar()

Add a variable and the associated non-zero coefficients as column.

**Synopsis**

```
Var addVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    Column col,
    String name)
```

**Arguments**

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`col`: column object for specifying a set of constraints to which the variable belongs.

`name`: name for new variable.

**Return**

new variable object.

### Model.addVars()

Add new variables to model.

**Synopsis**

```
VarArray addVars(
    int count,
    char vtype,
    String prefix)
```

**Arguments**

`count`: the number of variables to add.

`vtype`: variable types for new variables.

`prefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model.addVars()

Add new variables to model.

**Synopsis**

```
VarArray addVars(
    int count,
    double lb,
    double ub,
    double obj,
    char vtype,
    String prefix)
```

**Arguments**

`count`: the number of variables to add.

`lb`: lower bound for new variables.

`ub`: upper bound for new variables.

`obj`: objective coefficient for new variables.

`vtype`: variable type for new variables.

`prefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model.addVars()

Add new variables to model.

**Synopsis**

```
VarArray addVars(
    int count,
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    String prefix)
```

**Arguments**

`count`: the number of variables to add.

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`prefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model.addVars()

Add new variables to model.

**Synopsis**

```
VarArray addVars(
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    Column[] cols,
    String prefix)
```

**Arguments**

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`cols`: column objects for specifying a set of constraints to which each new variable belongs.

`prefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model.addVars()

Add new variables to model.

**Synopsis**

```
VarArray addVars(
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    ColumnArray cols,
    String prefix)
```

**Arguments**

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`cols`: columnarray for specifying a set of constraints to which each new variable belongs.

`prefix`: prefix part for names of new variables.

**Return**

array of new variable objects.

### Model.bsolve()

Given a sparse vector, perform a backward solve. The result is stored in place in the same vector.

**Synopsis**

```
int bsolve(
    double[] vals,
    int[] idxes,
    int len)
```

**Arguments**

`vals`: input values of the sparse vector; output values of the result, up to row size.

`idxes`: input indices of the sparse vector; output indices of the result, up to row size.

`len`: length of the sparse input vector.

**Return**

length of the result sparse vector.

### Model.clear()

Clear all settings including problem itself.

**Synopsis**

```
void clear()
```

### Model.clone()

Deep copy COPT model.

**Synopsis**

```
Model clone()
```

**Return**

cloned model object.

### Model.computeIIS()

Compute IIS for infeasible model.

**Synopsis**

```
void computeIIS()
```

### Model.computeShadowPriceRange()

Compute the shadow price range for a given constraint.

**Synopsis**

```
double[] computeShadowPriceRange(int iRow, int ifRhs)
```

**Arguments**

`iRow`: constraint (row) index.

`ifRhs`: for range constraint, whether to use RHS for computing shadow price.

**Return**

lower and upper bound of the shadow price.

### Model.delNlObj()

delete nonlinear part of objective in model.

**Synopsis**

```
void delNlObj()
```

### Model.delObjN()

Delete linear part of specific multi-objective function in model.

**Synopsis**

```
void delObjN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

### Model.delPsdObj()

delete PSD part of objective in model.

**Synopsis**

```
void delPsdObj()
```

### Model.delQuadObj()

delete quadratic part of objective in model.

**Synopsis**

```
void delQuadObj()
```

### Model.dispose()

Release this model reference on the calling thread. Call it (e.g. in a finally block) explicitly on the same thread, instead of relying on the GC finalizer thread. Note the underlying native problem is reference-counted and destroyed when the last referring to it is released.

**Synopsis**

```
void dispose()
```

### Model.feasRelax()

Compute feasibility relaxation for infeasible model.

**Synopsis**

```
void feasRelax(
    VarArray vars,
    double[] colLowPen,
    double[] colUppPen,
    ConstrArray cons,
    double[] rowBndPen,
    double[] rowUppPen)
```

**Arguments**

`vars`: an array of variables.

`colLowPen`: penalties for lower bounds of variables.

`colUppPen`: penalties for upper bounds of variables.

`cons`: an array of constraints.

`rowBndPen`: penalties for right hand sides of constraints.

`rowUppPen`: penalties for upper bounds of range constraints.

### Model.feasRelax()

Compute feasibility relaxation for infeasible model.

**Synopsis**

```
void feasRelax(int ifRelaxVars, int ifRelaxCons)
```

**Arguments**

`ifRelaxVars`: whether to relax variables.

`ifRelaxCons`: whether to relax constraints.

### Model.fsolve()

Given a sparse vector, perform a forward solve. The result is stored in place in the same vector.

**Synopsis**

```
int fsolve(
    double[] vals,
    int[] idxes,
    int len)
```

**Arguments**

`vals`: input values of the sparse vector; output values of the result, up to row size.

`idxes`: input indices of the sparse vector; output indices of the result, up to row size.

`len`: length of the sparse input vector.

**Return**

length of the result sparse vector.

### Model.get()

Query values of information associated with variables.

**Synopsis**

```
double[] get(String name, Var[] vars)
```

**Arguments**

`name`: name of information.

`vars`: a list of interested variables.

**Return**

values of information.

### Model.get()

Query values of information associated with variables.

**Synopsis**

```
double[] get(String name, VarArray vars)
```

**Arguments**

`name`: name of information.

`vars`: array of interested variables.

**Return**

values of information.

### Model.get()

Query values of information associated with constraints.

**Synopsis**

```
double[] get(String name, Constraint[] constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of interested constraints.

**Return**

values of information.

### Model.get()

Query values of information associated with constraints.

**Synopsis**

```
double[] get(String name, ConstrArray constrs)
```

**Arguments**

`name`: name of information.

`constrs`: array of interested constraints.

**Return**

values of information.

### Model.get()

Query values of information associated with nonlinear constraints.

**Synopsis**

```
double[] get(String name, NlConstraint[] constrs)
```

**Arguments**

`name`: name of information.

`constrs`: array of desired nonlinear constraints.

**Return**

output array of information values.

### Model.get()

Query values of information associated with nonlinear constraints.

**Synopsis**

```
double[] get(String name, NlConstrArray constrs)
```

**Arguments**

`name`: name of information.

`constrs`: an array object of desired nonlinear constraints.

**Return**

output array of information values.

### Model.get()

Query values of information associated with quadratic constraints.

**Synopsis**

```
double[] get(String name, QConstraint[] constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of interested quadratic constraints.

**Return**

values of information.

### Model.get()

Query values of information associated with quadratic constraints.

**Synopsis**

```
double[] get(String name, QConstrArray constrs)
```

**Arguments**

`name`: name of information.

`constrs`: array of interested quadratic constraints.

**Return**

values of information.

### Model.get()

Query values of information associated with PSD constraints.

**Synopsis**

```
double[] get(String name, PsdConstraint[] constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired PSD constraints.

**Return**

output array of information values.

### Model.get()

Query values of information associated with PSD constraints.

**Synopsis**

```
double[] get(String name, PsdConstrArray constrs)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired PSD constraints.

**Return**

output array of information values.

### Model.getAffineCone()

Get an affine cone constraint of given index in model.

**Synopsis**

```
AffineCone getAffineCone(int idx)
```

**Arguments**

`idx`: index of the desired affine cone constraint.

**Return**

the desired affine cone constraint object.

### Model.GetAffineConeBuilder()

Get builder of given affine cone constraint in model.

**Synopsis**

```
AffineConeBuilder GetAffineConeBuilder(AffineCone cone)
```

**Arguments**

`cone`: affine cone constraint.

**Return**

desired affine cone constraint builder.

### Model.GetAffineConeBuilders()

Get builders of all affine cone constraints in model.

**Synopsis**

```
AffineConeBuilderArray GetAffineConeBuilders()
```

**Return**

array object of affine cone constraint builders.

### Model.getAffineConeBuilders()

Get builders of given affine cone constraints in model.

**Synopsis**

```
AffineConeBuilderArray getAffineConeBuilders(AffineCone[] cones)
```

**Arguments**

`cones`: array of affine cone constraints.

**Return**

array object of desired affine cone constraint builders.

### Model.getAffineConeBuilders()

Get builders of given affine cone constraints in model.

**Synopsis**

```
AffineConeBuilderArray getAffineConeBuilders(AffineConeArray cones)
```

**Arguments**

`cones`: array of affine cone constraints.

**Return**

array object of desired affine cone constraint builders.

### Model.getAffineConeByName()

Get an affine cone constraint of given name in model.

**Synopsis**

```
AffineCone getAffineConeByName(String name)
```

**Arguments**

`name`: name of the desired affine cone constraint.

**Return**

the desired affine cone constraint object.

### Model.getAffineCones()

Get all affine cone constraints in model.

**Synopsis**

```
AffineConeArray getAffineCones()
```

**Return**

array object of affine cone constraints.

### Model.getBasisHead()

Get indices of the variables that make up the current basis.

**Synopsis**

```
int[] getBasisHead()
```

**Return**

array of basis head indices.

### Model.getCoeff()

Get the coefficient of variable in linear constraint.

**Synopsis**

```
double getCoeff(Constraint constr, Var var)
```

**Arguments**

`constr`: The requested constraint.

`var`: The requested variable.

**Return**

The requested coefficient.

### Model.getCol()

Get a column object that have a list of constraints in which the variable participates.

**Synopsis**

```
Column getCol(Var var)
```

**Arguments**

`var`: a variable object.

**Return**

a column object associated with a variable.

### Model.getColBasis()

Get status of column basis.

**Synopsis**

```
int[] getColBasis()
```

**Return**

basis status.

### Model.getCone()

Get a cone constraint of given index in model.

**Synopsis**

```
Cone getCone(int idx)
```

**Arguments**

`idx`: index of the desired cone constraint.

**Return**

the desired cone constraint object.

### Model.getConeBuilders()

Get builders of all cone constraints in model.

**Synopsis**

```
ConeBuilderArray getConeBuilders()
```

**Return**

array object of cone constraint builders.

### Model.getConeBuilders()

Get builders of given cone constraints in model.

**Synopsis**

```
ConeBuilderArray getConeBuilders(Cone[] cones)
```

**Arguments**

`cones`: array of cone constraints.

**Return**

array object of desired cone constraint builders.

### Model.getConeBuilders()

Get builders of given cone constraints in model.

**Synopsis**

```
ConeBuilderArray getConeBuilders(ConeArray cones)
```

**Arguments**

`cones`: array of cone constraints.

**Return**

array object of desired cone constraint builders.

### Model.getCones()

Get all cone constraints in model.

**Synopsis**

```
ConeArray getCones()
```

**Return**

array object of cone constraints.

### Model.getConstr()

Get a constraint of given index in model.

**Synopsis**

```
Constraint getConstr(int idx)
```

**Arguments**

`idx`: index of the desired constraint.

**Return**

the desired constraint object.

### Model.getConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

**Synopsis**

```
ConstrBuilder getConstrBuilder(Constraint constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

constraint builder object.

### Model.getConstrBuilders()

Get builders of all constraints in model.

**Synopsis**

```
ConstrBuilderArray getConstrBuilders()
```

**Return**

array object of constraint builders.

### Model.getConstrByName()

Get a constraint of given name in model.

**Synopsis**

```
Constraint getConstrByName(String name)
```

**Arguments**

`name`: name of the desired constraint.

**Return**

the desired constraint object.

### Model.getConstrLowerIIS()

Get IIS status of lower bounds of constraints.

**Synopsis**

```
int[] getConstrLowerIIS(ConstrArray constrs)
```

**Arguments**

`constrs`: Array of constraints.

**Return**

IIS status of lower bounds of constraints.

### Model.getConstrLowerIIS()

Get IIS status of lower bounds of constraints.

**Synopsis**

```
int[] getConstrLowerIIS(Constraint[] constrs)
```

**Arguments**

`constrs`: Array of constraints.

**Return**

IIS status of lower bounds of constraints.

### Model.getConstrs()

Get all constraints in model.

**Synopsis**

```
ConstrArray getConstrs()
```

**Return**

array object of constraints.

### Model.getConstrUpperIIS()

Get IIS status of upper bounds of constraints.

**Synopsis**

```
int[] getConstrUpperIIS(ConstrArray constrs)
```

**Arguments**

`constrs`: Array of constraints.

**Return**

IIS status of upper bounds of constraints.

### Model.getConstrUpperIIS()

Get IIS status of upper bounds of constraints.

**Synopsis**

```
int[] getConstrUpperIIS(Constraint[] constrs)
```

**Arguments**

`constrs`: Array of constraints.

**Return**

IIS status of upper bounds of constraints.

### Model.getDblAttr()

Get value of a COPT double attribute.

**Synopsis**

```
double getDblAttr(String attr)
```

**Arguments**

`attr`: name of double attribute.

**Return**

value of double attribute.

### Model.getDblAttrN()

Get value of a double attribute of a multi-objective function.

**Synopsis**

```
double getDblAttrN(int idx, String attr)
```

**Arguments**

`idx`: index of a multi-objective function.

`attr`: name of double attribute.

**Return**

value of double attribute.

### Model.getDblParam()

Get value of a COPT double parameter.

**Synopsis**

```
double getDblParam(String param)
```

**Arguments**

`param`: name of double parameter.

**Return**

value of double parameter.

### Model.getDblParamInfo()

Get current, default, minimum, maximum of COPT double parameter.

**Synopsis**

```
double[] getDblParamInfo(String name)
```

**Arguments**

`name`: name of integer parameter.

**Return**

current, default, minimum, maximum of COPT double parameter.

### Model.getDblParamN()

Get value of a double parameter of a multi-objective function.

**Synopsis**

```
double getDblParamN(int idx, String param)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of double parameter.

**Return**

value of double parameter.

### Model.getExpCone()

Get an exponential cone constraint of given index in model.

**Synopsis**

```
ExpCone getExpCone(int idx)
```

**Arguments**

`idx`: index of the desired exponential cone constraint.

**Return**

the desired exponential cone constraint object.

### Model.getExpConeBuilders()

Get builders of all exponential cone constraints in model.

**Synopsis**

```
ExpConeBuilderArray getExpConeBuilders()
```

**Return**

array object of exponential cone constraint builders.

### Model.getExpConeBuilders()

Get builders of given exponential cone constraints in model.

**Synopsis**

```
ExpConeBuilderArray getExpConeBuilders(ExpCone[] cones)
```

**Arguments**

`cones`: array of exponential cone constraints.

**Return**

array object of desired exponential cone constraint builders.

### Model.getExpConeBuilders()

Get builders of given exponential cone constraints in model.

**Synopsis**

```
ExpConeBuilderArray getExpConeBuilders(ExpConeArray cones)
```

**Arguments**

`cones`: array of exponential cone constraints.

**Return**

array object of desired exponential cone constraint builders.

### Model.getExpCones()

Get all exponential cone constraints in model.

**Synopsis**

```
ExpConeArray getExpCones()
```

**Return**

array object of exponential cone constraints.

### Model.getGenConstr()

Get a general constraint of given index in model.

**Synopsis**

```
GenConstr getGenConstr(int idx)
```

**Arguments**

`idx`: index of the desired general constraint.

**Return**

the desired general constraint object.

### Model.getGenConstrByName()

Get a general constraint of given name in model.

**Synopsis**

```
GenConstr getGenConstrByName(String name)
```

**Arguments**

`name`: name of the desired general constraint.

**Return**

the desired general constraint object.

### Model.getGenConstrIndicator()

Get builder of given general constraint of type indicator.

**Synopsis**

```
GenConstrBuilder getGenConstrIndicator(GenConstr indicator)
```

**Arguments**

`indicator`: a general constraint of type indicator.

**Return**

builder object of general constraint of type indicator.

### Model.getGenConstrIndicators()

Get builders of all general constraints in model.

**Synopsis**

```
GenConstrBuilderArray getGenConstrIndicators()
```

**Return**

array object of general constraint builders.

### Model.getGenConstrs()

Get all general constraints in model.

**Synopsis**

```
GenConstrArray getGenConstrs()
```

**Return**

array object of general constraints.

### Model.getIndicatorIIS()

Get IIS status of indicator constraints.

**Synopsis**

```
int[] getIndicatorIIS(GenConstrArray genconstrs)
```

**Arguments**

`genconstrs`: Array of indicator constraints.

**Return**

IIS status of indicator constraints.

### Model.getIndicatorIIS()

Get IIS status of indicator constraints.

**Synopsis**

```
int[] getIndicatorIIS(GenConstr[] genconstrs)
```

**Arguments**

`genconstrs`: Array of indicator constraints.

**Return**

IIS status of indicator constraints.

### Model.getIntAttr()

Get value of a COPT integer attribute

**Synopsis**

```
int getIntAttr(String attr)
```

**Arguments**

`attr`: name of integer attribute.

**Return**

value of integer attribute.

### Model.getIntAttrN()

Get value of an integer attribute of a multi-objective function.

**Synopsis**

```
int getIntAttrN(int idx, String attr)
```

**Arguments**

`idx`: index of a multi-objective function.

`attr`: name of integer attribute.

**Return**

value of integer attribute.

### Model.getIntParam()

Get value of a COPT integer parameter.

**Synopsis**

```
int getIntParam(String param)
```

**Arguments**

`param`: name of integer parameter.

**Return**

value of integer parameter.

### Model.getIntParamInfo()

Get current, default, minimum, maximum of COPT integer parameter.

**Synopsis**

```
int[] getIntParamInfo(String name)
```

**Arguments**

`name`: name of integer parameter.

**Return**

current, default, minimum, maximum of COPT integer parameter.

### Model.getIntParamN()

Get value of an integer parameter of a multi-objective function.

**Synopsis**

```
int getIntParamN(int idx, String param)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of integer parameter.

**Return**

value of integer parameter.

### Model.getLmiCoeff()

Get the symmetric matrix of variable in LMI constraint.

**Synopsis**

```
SymMatrix getLmiCoeff(LmiConstraint constr, Var var)
```

**Arguments**

`constr`: The desired LMI constraint.

`var`: The desired variable.

**Return**

The associated coefficient matrix.

### Model.getLmiConstr()

Get LMI constraint of given index in model.

**Synopsis**

```
LmiConstraint getLmiConstr(int idx)
```

**Arguments**

`idx`: index of desired LMI constraint.

**Return**

LMI constraint object.

### Model.getLmiConstrByName()

Get LMI constraint of given name in model.

**Synopsis**

```
LmiConstraint getLmiConstrByName(String name)
```

**Arguments**

`name`: name of desired LMI constraint.

**Return**

LMI constraint object.

### Model.getLmiConstrs()

Get all LMI constraints in model.

**Synopsis**

```
LmiConstrArray getLmiConstrs()
```

**Return**

array object of LMI constraints.

### Model.getLmiRhs()

Get the symmetric matrix of constant of LMI constraint.

**Synopsis**

```
SymMatrix getLmiRhs(LmiConstraint constr)
```

**Arguments**

`constr`: The desired LMI constraint.

**Return**

matrix of constant term.

### Model.getLmiRow()

Get variables and associated symmetric matrices that participate in a LMI constraint.

**Synopsis**

```
LmiExpr getLmiRow(LmiConstraint constr)
```

**Arguments**

`constr`: given LMI constraint object.

**Return**

LMI expression object of the LMI constraint.

### Model.getLmiSolution()

Get LMI solution.

**Synopsis**

```
Object[] getLmiSolution()
```

**Return**

slack and dual values.

### Model.getLpSolution()

Get LP solution.

**Synopsis**

```
Object[] getLpSolution()
```

**Return**

solution, slack, dual and reduced values.

### Model.getNlConstr()

Get a nonlinear constraint of given index in model.

**Synopsis**

```
NlConstraint getNlConstr(int idx)
```

**Arguments**

`idx`: index of the desired nonlinear constraint.

**Return**

the desired nonlinear constraint object.

### Model.getNlConstrBuilder()

Get builder of a nonlinear constraint in model, including nonlinear expression, sense and RHS.

**Synopsis**

```
NlConstrBuilder getNlConstrBuilder(NlConstraint constr)
```

**Arguments**

`constr`: a nonlinear constraint object.

**Return**

nonlinear constraint builder object.

### Model.getNlConstrBuilders()

Get builders of all nonlinear constraints in model.

**Synopsis**

```
NlConstrBuilderArray getNlConstrBuilders()
```

**Return**

array object of nonlinear constraint builders.

### Model.getNlConstrByName()

Get a nonlinear constraint of given name in model.

**Synopsis**

```
NlConstraint getNlConstrByName(String name)
```

**Arguments**

`name`: name of the desired constraint.

**Return**

the desired nonlinear constraint object.

### Model.getNlConstrs()

Get all nonlinear constraints in model.

**Synopsis**

```
NlConstrArray getNlConstrs()
```

**Return**

array object of nonlinear constraints.

### Model.getNlObjective()

Get nonlinear objective of model.

**Synopsis**

```
NlExpr getNlObjective()
```

**Return**

a nonlinear expression object.

### Model.getNlRow()

Get nonlinear expression of a nonlinear constraint.

**Synopsis**

```
NlExpr getNlRow(NlConstraint constr)
```

**Arguments**

`constr`: a nonlinear constraint object.

**Return**

output object of nonlinear expression.

### Model.getObjective()

Get linear expression of objective for model.

**Synopsis**

```
Expr getObjective()
```

**Return**

an linear expression object.

### Model.getObjectiveN()

Get linear expression of a multi-objective function in model.

**Synopsis**

```
Expr getObjectiveN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

**Return**

a linear expression object.

### Model.getObjParamN()

Get value of objective parameter of a multi-objective function.

**Synopsis**

```
double getObjParamN(int idx, String param)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of objective parameter, including priority, weight, abstol and reltol.

**Return**

value of objective parameter.

### Model.getPoolObjVal()

Get the idx-th objective value in solution pool.

**Synopsis**

```
double getPoolObjVal(int idx)
```

**Arguments**

`idx`: Index of solution.

**Return**

The requested objective value.

### Model.getPoolObjValN()

Get the objective value of required multi-objective function in solution pool.

**Synopsis**

```
double getPoolObjValN(int idx, int iSol)
```

**Arguments**

`idx`: index of a multi-objective function.

`iSol`: index of solution.

**Return**

value of required multi-objective function.

### Model.getPoolSolution()

Get the idx-th solution in solution pool.

**Synopsis**

```
double[] getPoolSolution(int idx, VarArray vars)
```

**Arguments**

`idx`: Index of solution.

`vars`: The requested variables.

**Return**

The requested solution.

### Model.getPoolSolution()

Get the idx-th solution in solution pool.

**Synopsis**

```
double[] getPoolSolution(int idx, Var[] vars)
```

**Arguments**

`idx`: Index of solution.

`vars`: The requested variables.

**Return**

The requested solution.

### Model.getPsdCoeff()

Get the symmetric matrix of PSD variable in a PSD constraint.

**Synopsis**

```
SymMatrix getPsdCoeff(PsdConstraint constr, PsdVar var)
```

**Arguments**

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

**Return**

The associated coefficient matrix.

### Model.getPsdConstr()

Get PSD constraint of given index in model.

**Synopsis**

```
PsdConstraint getPsdConstr(int idx)
```

**Arguments**

`idx`: index of desired PSD constraint.

**Return**

PSD constraint object.

### Model.getPsdConstrBuilder()

Get builder of a PSD constraint in model, including PSD variables, sense and associated symmetric matrix.

**Synopsis**

```
PsdConstrBuilder getPsdConstrBuilder(PsdConstraint constr)
```

**Arguments**

`constr`: a PSD constraint object.

**Return**

PSD constraint builder object.

### Model.getPsdConstrBuilders()

Get builders of all PSD constraints in model.

**Synopsis**

```
PsdConstrBuilderArray getPsdConstrBuilders()
```

**Return**

array object of PSD constraint builders.

### Model.getPsdConstrByName()

Get PSD constraint of given name in model.

**Synopsis**

```
PsdConstraint getPsdConstrByName(String name)
```

**Arguments**

`name`: name of desired PSD constraint.

**Return**

PSD constraint object.

### Model.getPsdConstrs()

Get all PSD constraints in model.

**Synopsis**

```
PsdConstrArray getPsdConstrs()
```

**Return**

array object of PSD constraints.

### Model.getPsdObjective()

Get PSD objective of model.

**Synopsis**

```
PsdExpr getPsdObjective()
```

**Return**

a PSD expression object.

### Model.getPsdRow()

Get PSD variables and associated symmetric matrices that participate in a PSD constraint.

**Synopsis**

```
PsdExpr getPsdRow(PsdConstraint constr)
```

**Arguments**

`constr`: a PSD constraint object.

**Return**

PSD expression object of the PSD constraint.

### Model.getPsdSolution()

Get PSD solution.

**Synopsis**

```
Object[] getPsdSolution()
```

**Return**

solution, slack, dual and reduced values.

### Model.getPsdVar()

Get a PSD variable of given index in model.

**Synopsis**

```
PsdVar getPsdVar(int idx)
```

**Arguments**

`idx`: index of the desired PSD variable.

**Return**

the desired PSD variable object.

### Model.getPsdVarByName()

Get a PSD variable of given name in model.

**Synopsis**

```
PsdVar getPsdVarByName(String name)
```

**Arguments**

`name`: name of the desired PSD variable.

**Return**

the desired PSD variable object.

### Model.getPsdVars()

Get all PSD variables in model.

**Synopsis**

```
PsdVarArray getPsdVars()
```

**Return**

array object of PSD variables.

### Model.getQConstr()

Get a quadratic constraint of given index in model.

**Synopsis**

```
QConstraint getQConstr(int idx)
```

**Arguments**

`idx`: index of the desired quadratic constraint.

**Return**

the desired quadratic constraint object.

### Model.getQConstrBuilder()

Get builder of a quadratic constraint in model, including variables and associated coefficients, sense and RHS.

**Synopsis**

```
QConstrBuilder getQConstrBuilder(QConstraint constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

constraint builder object.

### Model.getQConstrBuilders()

Get builders of all constraints in model.

**Synopsis**

```
QConstrBuilderArray getQConstrBuilders()
```

**Return**

array object of constraint builders.

### Model.getQConstrByName()

Get a quadratic constraint of given name in model.

**Synopsis**

```
QConstraint getQConstrByName(String name)
```

**Arguments**

`name`: name of the desired constraint.

**Return**

the desired quadratic constraint object.

### Model.getQConstrs()

Get all quadratic constraints in model.

**Synopsis**

```
QConstrArray getQConstrs()
```

**Return**

array object of quadratic constraints.

### Model.getQuadObjective()

Get quadratic objective of model.

**Synopsis**

```
QuadExpr getQuadObjective()
```

**Return**

a quadratic expression object.

### Model.getQuadRow()

Get quadratic expression that participate in quadratic constraint.

**Synopsis**

```
QuadExpr getQuadRow(QConstraint constr)
```

**Arguments**

`constr`: a quadratic constraint object.

**Return**

quadratic expression object of the constraint.

### Model.getRow()

Get variables that participate in a constraint, and the associated coefficients.

**Synopsis**

```
Expr getRow(Constraint constr)
```

**Arguments**

`constr`: a constraint object.

**Return**

expression object of the constraint.

### Model.getRowBasis()

Get status of row basis.

**Synopsis**

```
int[] getRowBasis()
```

**Return**

basis status.

### Model.getSolution()

Get MIP solution.

**Synopsis**

```
double[] getSolution()
```

**Return**

solution values.

### Model.getSos()

Get a SOS constraint of given index in model.

**Synopsis**

```
Sos getSos(int idx)
```

**Arguments**

`idx`: index of the desired SOS constraint.

**Return**

the desired SOS constraint object.

### Model.getSosBuilders()

Get builders of all SOS constraints in model.

**Synopsis**

```
SosBuilderArray getSosBuilders()
```

**Return**

array object of SOS constraint builders.

### Model.getSosBuilders()

Get builders of given SOS constraints in model.

**Synopsis**

```
SosBuilderArray getSosBuilders(Sos[] soss)
```

**Arguments**

`soss`: array of SOS constraints.

**Return**

array object of desired SOS constraint builders.

### Model.getSosBuilders()

Get builders of given SOS constraints in model.

**Synopsis**

```
SosBuilderArray getSosBuilders(SosArray soss)
```

**Arguments**

`soss`: array of SOS constraints.

**Return**

array object of desired SOS constraint builders.

### Model.getSOSIIS()

Get IIS status of SOS constraints.

**Synopsis**

```
int[] getSOSIIS(SosArray soss)
```

**Arguments**

`soss`: Array of SOS constraints.

**Return**

IIS status of SOS constraints.

### Model.getSOSIIS()

Get IIS status of SOS constraints.

**Synopsis**

```
int[] getSOSIIS(Sos[] soss)
```

**Arguments**

`soss`: Array of SOS constraints.

**Return**

IIS status of SOS constraints.

### Model.getSoss()

Get all SOS constraints in model.

**Synopsis**

```
SosArray getSoss()
```

**Return**

array object of SOS constraints.

### Model.getSymMat()

Get a symmetric matrix of given index in model.

**Synopsis**

```
SymMatrix getSymMat(int idx)
```

**Arguments**

`idx`: index of the desired symmetric matrix.

**Return**

the desired symmetric matrix object.

### Model.getVar()

Get a variable of given index in model.

**Synopsis**

```
Var getVar(int idx)
```

**Arguments**

`idx`: index of the desired variable.

**Return**

the desired variable object.

### Model.getVarByName()

Get a variable of given name in model.

**Synopsis**

```
Var getVarByName(String name)
```

**Arguments**

`name`: name of the desired variable.

**Return**

the desired variable object.

### Model.getVarLowerIIS()

Get IIS status of lower bounds of variables.

**Synopsis**

```
int[] getVarLowerIIS(VarArray vars)
```

**Arguments**

`vars`: Array of variables.

**Return**

IIS status of lower bounds of variables.

### Model.getVarLowerIIS()

Get IIS status of lower bounds of variables.

**Synopsis**

```
int[] getVarLowerIIS(Var[] vars)
```

**Arguments**

`vars`: Array of variables.

**Return**

IIS status of lower bounds of variables.

### Model.getVars()

Get all variables in model.

**Synopsis**

```
VarArray getVars()
```

**Return**

variable array object.

### Model.getVarUpperIIS()

Get IIS status of upper bounds of variables.

**Synopsis**

```
int[] getVarUpperIIS(VarArray vars)
```

**Arguments**

`vars`: Array of variables.

**Return**

IIS status of upper bounds of variables.

### Model.getVarUpperIIS()

Get IIS status of upper bounds of variables.

**Synopsis**

```
int[] getVarUpperIIS(Var[] vars)
```

**Arguments**

`vars`: Array of variables.

**Return**

IIS status of upper bounds of variables.

### Model.interrupt()

Interrupt optimization of current problem.

**Synopsis**

```
void interrupt()
```

### Model.loadMipStart()

Load final initial values of variables to the problem.

**Synopsis**

```
void loadMipStart()
```

### Model.loadTuneParam()

Load specified tuned parameters into model.

**Synopsis**

```
void loadTuneParam(int idx)
```

**Arguments**

`idx`: Index of tuned parameters.

### Model.read()

Read problem, solution, basis, MIP start or COPT parameters from file.

**Synopsis**

```
void read(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readBasis()

Read basis from file.

**Synopsis**

```
void readBasis(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readBin()

Read problem in COPT binary format from file.

**Synopsis**

```
void readBin(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readCbf()

Read problem in CBF format from file.

**Synopsis**

```
void readCbf(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readJsonSol()

Read solution in format of JSON from file.

**Synopsis**

```
void readJsonSol(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readLp()

Read problem in LP format from file.

**Synopsis**

```
void readLp(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readMps()

Read problem in MPS format from file.

**Synopsis**

```
void readMps(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readMst()

Read MIP start information from file.

**Synopsis**

```
void readMst(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readOrd()

Read branching order from file.

**Synopsis**

```
void readOrd(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readParam()

Read COPT parameters from file.

**Synopsis**

```
void readParam(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readSdpa()

Read problem in SDPA format from file.

**Synopsis**

```
void readSdpa(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readSol()

Read solution from file.

**Synopsis**

```
void readSol(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.readTune()

Read tuning parameters from file.

**Synopsis**

```
void readTune(String filename)
```

**Arguments**

`filename`: an input file name.

### Model.remove()

Remove an array of variables from model.

**Synopsis**

```
void remove(Var[] vars)
```

**Arguments**

`vars`: a list of variables.

### Model.remove()

Remove array of variables from model.

**Synopsis**

```
void remove(VarArray vars)
```

**Arguments**

`vars`: an array of variables.

### Model.remove()

Remove a list of constraints from model.

**Synopsis**

```
void remove(Constraint[] constrs)
```

**Arguments**

`constrs`: a list of constraints.

### Model.remove()

Remove a list of constraints from model.

**Synopsis**

```
void remove(ConstrArray constrs)
```

**Arguments**

`constrs`: an array of constraints.

### Model.remove()

Remove an array of nonlinear constraints from model.

**Synopsis**

```
void remove(NlConstraint[] constrs)
```

**Arguments**

`constrs`: array of nonlinear constraints.

### Model.remove()

Remove a list of nonlinear constraints from model.

**Synopsis**

```
void remove(NlConstrArray constrs)
```

**Arguments**

`constrs`: array object of nonlinear constraints.

### Model.remove()

Remove a list of SOS constraints from model.

**Synopsis**

```
void remove(Sos[] soss)
```

**Arguments**

`soss`: a list of SOS constraints.

### Model.remove()

Remove a list of SOS constraints from model.

**Synopsis**

```
void remove(SosArray soss)
```

**Arguments**

`soss`: an array of SOS constraints.

### Model.remove()

Remove a list of cone constraints from model.

**Synopsis**

```
void remove(Cone[] cones)
```

**Arguments**

`cones`: a list of cone constraints.

### Model.remove()

Remove a list of cone constraints from model.

**Synopsis**

```
void remove(ConeArray cones)
```

**Arguments**

`cones`: an array of cone constraints.

### Model.remove()

Remove a list of exponential cone constraints from model.

**Synopsis**

```
void remove(ExpCone[] cones)
```

**Arguments**

`cones`: a list of exponential cone constraints.

### Model.remove()

Remove a list of exponential cone constraints from model.

**Synopsis**

```
void remove(ExpConeArray cones)
```

**Arguments**

`cones`: an array of exponential cone constraints.

### Model.remove()

Remove a list of affine cone constraints from model.

**Synopsis**

```
void remove(AffineCone[] cones)
```

**Arguments**

`cones`: a list of affine cone constraints.

### Model.remove()

Remove an array of affine cone constraints from model.

**Synopsis**

```
void remove(AffineConeArray cones)
```

**Arguments**

`cones`: an array of affine cone constraints.

### Model.remove()

Remove a list of general constraints from model.

**Synopsis**

```
void remove(GenConstr[] genConstrs)
```

**Arguments**

`genConstrs`: a list of general constraints.

### Model.remove()

Remove a list of general constraints from model.

**Synopsis**

```
void remove(GenConstrArray genConstrs)
```

**Arguments**

`genConstrs`: an array of general constraints.

### Model.remove()

Remove a list of quadratic constraints from model.

**Synopsis**

```
void remove(QConstraint[] qconstrs)
```

**Arguments**

`qconstrs`: an array of quadratic constraints.

### Model.remove()

Remove a list of quadratic constraints from model.

**Synopsis**

```
void remove(QConstrArray qconstrs)
```

**Arguments**

`qconstrs`: an array of quadratic constraints.

### Model.remove()

Remove a list of PSD variables from model.

**Synopsis**

```
void remove(PsdVar[] vars)
```

**Arguments**

`vars`: an array of PSD variables.

### Model.remove()

Remove a list of PSD variables from model.

**Synopsis**

```
void remove(PsdVarArray vars)
```

**Arguments**

`vars`: an array of PSD variables.

### Model.remove()

Remove a list of PSD constraints from model.

**Synopsis**

```
void remove(PsdConstraint[] constrs)
```

**Arguments**

`constrs`: an array of PSD constraints.

### Model.remove()

Remove a list of PSD constraints from model.

**Synopsis**

```
void remove(PsdConstrArray constrs)
```

**Arguments**

`constrs`: an array of PSD constraints.

### Model.remove()

Remove a list of LMI constraints from model.

**Synopsis**

```
void remove(LmiConstrArray constrs)
```

**Arguments**

`constrs`: an array of LMI constraints.

### Model.remove()

Remove a list of LMI constraints from model.

**Synopsis**

```
void remove(LmiConstraint[] constrs)
```

**Arguments**

`constrs`: an array of LMI constraints.

### Model.reset()

Reset solution of problem only.

**Synopsis**

```
void reset()
```

### Model.resetAll()

Reset solution of problem, and additional information such as MIP start, etc.

**Synopsis**

```
void resetAll()
```

### Model.resetObjParamN()

Reset objective parameters of a multi-objective function.

**Synopsis**

```
void resetObjParamN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

### Model.resetParam()

Reset parameters to default settings.

**Synopsis**

```
void resetParam()
```

### Model.resetParamN()

Reset double and integer parameters of a multi-objective function.

**Synopsis**

```
void resetParamN(int idx)
```

**Arguments**

`idx`: index of a multi-objective function.

### Model.set()

Set values of information associated with variables.

**Synopsis**

```
void set(
    String name,
    Var[] vars,
    double[] vals)
```

**Arguments**

`name`: name of information.

`vars`: a list of interested variables.

`vals`: values of information.

### Model.set()

Set values of information associated with variables.

**Synopsis**

```
void set(
    String name,
    VarArray vars,
    double[] vals)
```

**Arguments**

`name`: name of information.

`vars`: array of interested variables.

`vals`: values of information.

### Model.set()

Set values of information associated with constraints.

**Synopsis**

```
void set(
    String name,
    Constraint[] constrs,
    double[] vals)
```

**Arguments**

`name`: name of information.

`constrs`: a list of interested constraints.

`vals`: values of information.

### Model.set()

Set values of information associated with constraints.

**Synopsis**

```
void set(
    String name,
    ConstrArray constrs,
    double[] vals)
```

**Arguments**

`name`: name of information.

`constrs`: array of interested constraints.

`vals`: values of information.

### Model.set()

Set values of information associated with nonlinear constraints.

**Synopsis**

```
void set(
    String name,
    NlConstraint[] constrs,
    double[] vals)
```

**Arguments**

`name`: name of double information.

`constrs`: array of desired nonlinear constraints.

`vals`: array of values of information.

### Model.set()

Set values of information associated with nonlinear constraints.

**Synopsis**

```
void set(
    String name,
    NlConstrArray constrs,
    double[] vals)
```

**Arguments**

`name`: name of double information.

`constrs`: an array object of desired nonlinear constraints.

`vals`: array of values of information.

### Model.set()

Set values of information associated with PSD constraints.

**Synopsis**

```
void set(
    String name,
    PsdConstraint[] constrs,
    double[] vals)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired PSD constraints.

`vals`: array of values of information.

### Model.set()

Set values of information associated with PSD constraints.

**Synopsis**

```
void set(
    String name,
    PsdConstrArray constrs,
    double[] vals)
```

**Arguments**

`name`: name of information.

`constrs`: a list of desired PSD constraints.

`vals`: array of values of information.

### Model.setBasis()

Set column and row basis status to model.

**Synopsis**

```
void setBasis(int[] colbasis, int[] rowbasis)
```

**Arguments**

`colbasis`: status of column basis.

`rowbasis`: status of row basis.

### Model.setCallback()

Set user callback to COPT model.

**Synopsis**

```
void setCallback(CallbackBase cb, int cbctx)
```

**Arguments**

`cb`: user callback instance, inheriting from CallbackBase class.

`cbctx`: COPT callback context.

### Model.setCoeff()

Set the coefficient of a variable in a linear constraint.

**Synopsis**

```
void setCoeff(
    Constraint constr,
    Var var,
    double newVal)
```

**Arguments**

`constr`: The requested constraint.

`var`: The requested variable.

`newVal`: New coefficient.

### Model.setCoeffs()

Set a list of coefficients in the model.

**Synopsis**

```
void setCoeffs(
    Constraint[] constrs,
    Var[] vars,
    double[] vals)
```

**Arguments**

`constrs`: Array of constraints for coefficients to be set.

`vars`: Array of vars for coefficients to be set.

`vals`: New values for coefficients.

### Model.setCoeffs()

Set a list of coefficients in the model.

**Synopsis**

```
void setCoeffs(
    ConstrArray constrs,
    VarArray vars,
    double[] vals)
```

**Arguments**

`constrs`: A list of constraints for coefficients to be set.

`vars`: A list of vars for coefficients to be set.

`vals`: New values for coefficients.

### Model.setDblParam()

Set value of a COPT double parameter.

**Synopsis**

```
void setDblParam(String param, double val)
```

**Arguments**

`param`: name of double parameter.

`val`: double value.

### Model.setDblParamN()

Set value of a double parameter of a multi-objective function.

**Synopsis**

```
void setDblParamN(
    int idx,
    String param,
    double val)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of double parameter.

`val`: new value of double parameter.

### Model.setIntParam()

Set value of a COPT integer parameter.

**Synopsis**

```
void setIntParam(String param, int val)
```

**Arguments**

`param`: name of integer parameter.

`val`: integer value.

### Model.setIntParamN()

Set value of an integer parameter of a multi-objective function.

**Synopsis**

```
void setIntParamN(
    int idx,
    String param,
    int val)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of integer parameter.

`val`: new value of integer parameter.

### Model.setLmiCoeff()

Set the coefficient matrix of a variable in LMI constraint.

**Synopsis**

```
void setLmiCoeff(
    LmiConstraint constr,
    Var var,
    SymMatrix mat)
```

**Arguments**

`constr`: The desired LMI constraint.

`var`: The desired variable.

`mat`: new coefficient matrix.

### Model.setLmiRhs()

Set constant matrix of LMI constraint.

**Synopsis**

```
void setLmiRhs(LmiConstraint constr, SymMatrix mat)
```

**Arguments**

`constr`: The desired LMI constraint.

`mat`: new constant matrix.

### Model.setLpSolution()

Set LP solution.

**Synopsis**

```
void setLpSolution(
    double[] value,
    double[] slack,
    double[] rowDual,
    double[] redCost)
```

**Arguments**

`value`: solution of variables.

`slack`: slack of constraints.

`rowDual`: dual value of constraints.

`redCost`: dual value of variables.

### Model.setMipStart()

Set initial values for variables of given number, starting from the first one.

**Synopsis**

```
void setMipStart(int count, double[] vals)
```

**Arguments**

`count`: the number of variables to set.

`vals`: values of variables.

### Model.setMipStart()

Set initial value for the specified variable.

**Synopsis**

```
void setMipStart(Var var, double val)
```

**Arguments**

`var`: an interested variable.

`val`: initial value of the variable.

### Model.setMipStart()

Set initial value for the specified variable.

**Synopsis**

```
void setMipStart(Var[] vars, double[] vals)
```

**Arguments**

`vars`: a list of interested variables.

`vals`: initial values of the variables.

### Model.setMipStart()

Set initial values for an array of variables.

**Synopsis**

```
void setMipStart(VarArray vars, double[] vals)
```

**Arguments**

`vars`: a list of interested variables.

`vals`: initial values of variables.

### Model.setNames()

Set names for given variables in model.

**Synopsis**

```
void setNames(Var[] vars, String[] names)
```

**Arguments**

`vars`: Array of variables.

`names`: String array of names for variables.

### Model.setNames()

Set names for given variables in model.

**Synopsis**

```
void setNames(VarArray vars, String[] names)
```

**Arguments**

`vars`: A list of variables.

`names`: string array of names for variables.

### Model.setNames()

Set names for given constraints in model.

**Synopsis**

```
void setNames(Constraint[] cons, String[] names)
```

**Arguments**

`cons`: array of constraints.

`names`: string array of names for constraints.

### Model.setNames()

Set names for given constraints in model.

**Synopsis**

```
void setNames(ConstrArray cons, String[] names)
```

**Arguments**

`cons`: a list of constraints.

`names`: string array of names for constraints.

### Model.setNames()

Set names for given general constraints in model.

**Synopsis**

```
void setNames(GenConstr[] genConstrs, String[] names)
```

**Arguments**

`genConstrs`: array of general constraints.

`names`: string array of names for general constraints.

### Model.setNames()

Set names for given general constraints in model.

**Synopsis**

```
void setNames(GenConstrArray genConstrs, String[] names)
```

**Arguments**

`genConstrs`: a list of general constraints.

`names`: string array of names for general constraints.

### Model.setNames()

Set names for given nonlinear constraints in model.

**Synopsis**

```
void setNames(NlConstraint[] cons, String[] names)
```

**Arguments**

`cons`: array of nonlinear constraints.

`names`: string array of names for nonlinear constraints.

### Model.setNames()

Set names for given nonlinear constraints in model.

**Synopsis**

```
void setNames(NlConstrArray cons, String[] names)
```

**Arguments**

`cons`: array object of nonlinear constraints.

`names`: string array of names for nonlinear constraints.

### Model.setNames()

Set names for given quadratic constraints in model.

**Synopsis**

```
void setNames(QConstraint[] cons, String[] names)
```

**Arguments**

`cons`: array of quadratic constraints.

`names`: string array of names for quadratic constraints.

### Model.setNames()

Set names for given quadratic constraints in model.

**Synopsis**

```
void setNames(QConstrArray cons, String[] names)
```

**Arguments**

`cons`: a list of quadratic constraints.

`names`: string array of names for quadratic constraints.

### Model.setNames()

Set names for given PSD variables in model.

**Synopsis**

```
void setNames(PsdVar[] vars, String[] names)
```

**Arguments**

`vars`: array of PSD variables.

`names`: string array of names for PSD variables.

### Model.setNames()

Set names for given PSD variables in model.

**Synopsis**

```
void setNames(PsdVarArray vars, String[] names)
```

**Arguments**

`vars`: a list of PSD variables.

`names`: string array of names for PSD variables.

### Model.setNames()

Set names for given PSD constraints in model.

**Synopsis**

```
void setNames(PsdConstraint[] cons, String[] names)
```

**Arguments**

`cons`: array of PSD constraints.

`names`: string array of names for PSD constraints.

### Model.setNames()

Set names for given PSD constraints in model.

**Synopsis**

```
void setNames(PsdConstrArray cons, String[] names)
```

**Arguments**

`cons`: a list of PSD constraints.

`names`: string array of names for PSD constraints.

### Model.setNames()

Set names for given LMI constraints in model.

**Synopsis**

```
void setNames(LmiConstraint[] cons, String[] names)
```

**Arguments**

`cons`: array of LMI constraints.

`names`: string array of names for LMI constraints.

### Model.setNames()

Set names for given LMI constraints in model.

**Synopsis**

```
void setNames(LmiConstrArray cons, String[] names)
```

**Arguments**

`cons`: a list of LMI constraints.

`names`: string array of names for LMI constraints.

### Model.setNames()

Set names for given affine cone constraints in model.

**Synopsis**

```
void setNames(AffineConeArray cones, String[] names)
```

**Arguments**

`cones`: an array of affine cone constraints.

`names`: string array of names for affine cone constraints.

### Model.setNlObjective()

Set nonlinear objective for model.

**Synopsis**

```
void setNlObjective(NlExpr expr, int sense)
```

**Arguments**

`expr`: nonlinear expression of the objective.

`sense`: optimization sense. optional, default value 0 does not change COPT sense.

### Model.setNlPrimalStart()

Given count, set initial values for variables of NLP from beginning.

**Synopsis**

```
void setNlPrimalStart(int count, double[] vals)
```

**Arguments**

`count`: the number of variables to set.

`vals`: initial values of variables.

### Model.setNlPrimalStart()

Set initial value for the specified variable of NLP.

**Synopsis**

```
void setNlPrimalStart(Var var, double val)
```

**Arguments**

`var`: an interested variable.

`val`: initial value of the variable.

### Model.setNlPrimalStart()

Set initial values for an array of variables of NLP.

**Synopsis**

```
void setNlPrimalStart(Var[] vars, double[] vals)
```

**Arguments**

`vars`: array of interested variables.

`vals`: initial values of variables.

### Model.setNlPrimalStart()

Set initial values for variable array of NLP.

**Synopsis**

```
void setNlPrimalStart(VarArray vars, double[] vals)
```

**Arguments**

`vars`: a list of interested variables.

`vals`: initial values of variables.

### Model.setObjConst()

Set objective constant.

**Synopsis**

```
void setObjConst(double constant)
```

**Arguments**

`constant`: constant value to set.

### Model.setObjective()

Set objective for model.

**Synopsis**

```
void setObjective(Expr expr, int sense)
```

**Arguments**

`expr`: expression of the objective.

`sense`: optimization sense, which is either Consts.MINIIMIZE or Consts.MAXIMIZE. Set sense to 0 if do not change current sense.

### Model.setObjectiveN()

Set a multi-objective function in model.

**Synopsis**

```
void setObjectiveN(
    int idx,
    Expr expr,
    int sense,
    double priority,
    double weight,
    double abstol,
    double reltol)
```

**Arguments**

`idx`: index of a multi-objective function.

`expr`: linear expression of the multi-objective function.

`sense`: optimization sense with value Consts.MINIIMIZE or Consts.MAXIIMIZE. Set sense to 0 if do not change current sense.

`priority`: objective parameter for priority.

`weight`: objective parameter for weight.

`abstol`: objective parameter for absolute tolerance.

`reltol`: objective parameter for relative tolerance.

### Model.setObjectiveN()

Set a multi-objective function in model, with default value of objective parameters.

**Synopsis**

```
void setObjectiveN(
    int idx,
    Expr expr,
    int sense)
```

**Arguments**

`idx`: index of a multi-objective function.

`expr`: linear expression of the multi-objective function.

`sense`: optimization sense with value Consts.MINIIMIZE or Consts.MAXIIMIZE. Set sense to 0 if do not change current sense.

### Model.setObjParamN()

Set value of objective parameter of a multi-objective function.

**Synopsis**

```
void setObjParamN(
    int idx,
    String param,
    double val)
```

**Arguments**

`idx`: index of a multi-objective function.

`param`: name of objective parameter, including priority, weight, abstol and reltol.

`val`: new value of objective parameter.

### Model.setObjSense()

Set objective sense for model.

**Synopsis**

```
void setObjSense(int sense)
```

**Arguments**

`sense`: the objective sense.

### Model.setPsdCoeff()

Set the coefficient matrix of a PSD variable in a PSD constraint.

**Synopsis**

```
void setPsdCoeff(
    PsdConstraint constr,
    PsdVar var,
    SymMatrix mat)
```

**Arguments**

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

`mat`: new coefficient matrix.

### Model.setPsdObjective()

Set PSD objective for model.

**Synopsis**

```
void setPsdObjective(PsdExpr expr, int sense)
```

**Arguments**

`expr`: PSD expression of the objective.

`sense`: optimization sense, which is either Consts.MINIIMIZE or Consts.MAXIMIZE. Set sense to 0 if do not change current sense.

### Model.setQuadObjective()

Set quadratic objective for model.

**Synopsis**

```
void setQuadObjective(QuadExpr expr, int sense)
```

**Arguments**

`expr`: quadratic expression of the objective.

`sense`: optimization sense, which is either Consts.MINIIMIZE or Consts.MAXIMIZE. Set sense to 0 if do not change current sense.

### Model.setSlackBasis()

Set slack basis to model.

**Synopsis**

```
void setSlackBasis()
```

### Model.setSolverLogCallback()

Set user log callback to COPT model. The model keeps a reference to the callback until it is replaced or the model is released.

**Synopsis**

```
void setSolverLogCallback(LogCallback cb)
```

**Arguments**

`cb`: user log callback instance, inheriting from LogCallback class.

### Model.setSolverLogFile()

Set log file for COPT.

**Synopsis**

```
void setSolverLogFile(String filename)
```

**Arguments**

`filename`: log file name.

### Model.solve()

Solve the model as MIP.

**Synopsis**

```
void solve()
```

### Model.solveLp()

Solve the model as LP.

**Synopsis**

```
void solveLp()
```

### Model.tune()

Tune model.

**Synopsis**

```
void tune()
```

### Model.update()

Update current model, including numerical range and removed items.

**Synopsis**

```
void update()
```

### Model.write()

Output problem, solution, basis, MIP start or modified COPT parameters to file.

**Synopsis**

```
void write(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writeBasis()

Output optimal basis to a file of type '.bas'.

**Synopsis**

```
void writeBasis(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writeBin()

Output problem to a file as COPT binary format.

**Synopsis**

```
void writeBin(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writeIIS()

Output IIS to file.

**Synopsis**

```
void writeIIS(String filename)
```

**Arguments**

`filename`: Output file name.

### Model.writeJsonSol()

Output solution to a file of type '.json'.

**Synopsis**

```
void writeJsonSol(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writeLp()

Output problem to a file as LP format.

**Synopsis**

```
void writeLp(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writeMps()

Output problem to a file as MPS format.

**Synopsis**

```
void writeMps(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writeMpsStr()

Output MPS problem to problem buffer.

**Synopsis**

```
ProbBuffer writeMpsStr()
```

**Return**

problem buffer for string of MPS problem.

### Model.writeMst()

Output MIP start information to a file of type '.mst'.

**Synopsis**

```
void writeMst(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writeNL()

Output problem to a file as NL format.

**Synopsis**

```
void writeNL(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writeOrd()

Output branching order to file.

**Synopsis**

```
void writeOrd(String filename)
```

**Arguments**

`filename`: Output file name.

### Model.writeParam()

Output modified COPT parameters to a file of type '.par'.

**Synopsis**

```
void writeParam(String filename)
```

**Arguments**

`filename`: an output file name.

### Model.writePoolSol()

Output selected pool solution to a file of type '.sol'.

**Synopsis**

```
void writePoolSol(int idx, String filename)
```

**Arguments**

`idx`: index of pool solution.

`filename`: an output file name.

#### Model.writeRelax()

Output feasibility relaxation problem to file.

**Synopsis**

```
void writeRelax(String filename)
```

**Arguments**

`filename`: Output file name.

#### Model.writeSol()

Output solution to a file of type '.sol'.

**Synopsis**

```
void writeSol(String filename)
```

**Arguments**

`filename`: an output file name.

#### Model.writeTuneParam()

Output specified tuned parameters to a file of type '.par'.

**Synopsis**

```
void writeTuneParam(int idx, String filename)
```

**Arguments**

`idx`: Index of tuned parameters.

`filename`: Output file name.

### 27.2.4 Var

COPT variable object. Variables are always associated with a particular model. User creates a variable object by adding a variable to a model, rather than by using constructor of Var class.

#### Var.get()

Get information value of the variable. Support information of "Value", "RedCost", "PrimalRay", "LB", "UB" and "Obj".

**Synopsis**

```
double get(String info)
```

**Arguments**

`info`: information name.

**Return**

information value.

### Var.getBasis()

Get basis status of the variable.

**Synopsis**

```
int getBasis()
```

**Return**

Basis status.

### Var.getIdx()

Get index of the variable.

**Synopsis**

```
int getIdx()
```

**Return**

variable index.

### Var.getLowerIIS()

Get IIS status for lower bound of the variable.

**Synopsis**

```
int getLowerIIS()
```

**Return**

IIS status.

### Var.getName()

Get name of the variable.

**Synopsis**

```
String getName()
```

**Return**

variable name.

### Var.getType()

Get type of the variable.

**Synopsis**

```
char getType()
```

**Return**

variable type.

### Var.getUpperIIS()

Get IIS status for upper bound of the variable.

**Synopsis**

```
int getUpperIIS()
```

**Return**

IIS status.

### Var.remove()

Remove variable from model.

**Synopsis**

```
void remove()
```

### Var.set()

Set information value of the variable. Support information of "LB", "UB" and "Obj".

**Synopsis**

```
void set(String info, double val)
```

**Arguments**

`info`: information name.

`val`: new information value.

### Var.setName()

Set name of the variable.

**Synopsis**

```
void setName(String name)
```

**Arguments**

`name`: variable name.

### Var.setType()

Set type of the variable.

**Synopsis**

```
void setType(char vtype)
```

**Arguments**

`vtype`: variable type.

### 27.2.5 VarArray

COPT variable array object. To store and access a set of Java *Var* objects, Cardinal Optimizer provides Java VarArray class, which defines the following methods.

#### VarArray.VarArray()

Constructor of vararray.

**Synopsis**

```
VarArray()
```

#### VarArray.getVar()

Get idx-th variable object.

**Synopsis**

```
Var getVar(int idx)
```

**Arguments**

`idx`: index of the variable.

**Return**

variable object with index idx.

#### VarArray.pushBack()

Add a variable object to variable array.

**Synopsis**

```
void pushBack(Var var)
```

**Arguments**

`var`: a variable object.

#### VarArray.size()

Get the number of variable objects.

**Synopsis**

```
int size()
```

**Return**

number of variable objects.

### 27.2.6 Expr

COPT linear expression object. A linear expression consists of a constant term, a list of terms of variables and associated coefficients. Linear expressions are used to build constraints.

#### Expr.Expr()

Constructor of a constant linear expression with constant 0.0

**Synopsis**

```
Expr()
```

#### Expr.Expr()

Constructor of a constant linear expression.

**Synopsis**

```
Expr(double constant)
```

**Arguments**

`constant`: constant value in expression object.

#### Expr.Expr()

Constructor of a linear expression with one term.

**Synopsis**

```
Expr(Var var)
```

**Arguments**

`var`: variable for the added term.

#### Expr.Expr()

Constructor of a linear expression with one term.

**Synopsis**

```
Expr(Var var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

#### Expr.add()

Add itself by a linear expression.

**Synopsis**

```
Expr add(Expr expr, double mult)
```

**Arguments**

`expr`: expression operand, including Expr, Var.

`mult`: constant multiplier.

**Return**

linear expression itself.

### Expr.add()

Add itself by a linear expression.

**Synopsis**

```
Expr add(Expr expr)
```

**Arguments**

`expr`: expression operand, including Expr, Var and constant.

**Return**

linear expression itself.

### Expr.addConstant()

Add extra constant to the expression.

**Synopsis**

```
void addConstant(double constant)
```

**Arguments**

`constant`: delta value to be added to expression constant.

### Expr.addExpr()

Add a linear expression to self.

**Synopsis**

```
void addExpr(Expr expr)
```

**Arguments**

`expr`: linear expression to be added.

### Expr.addExpr()

Add a linear expression to self.

**Synopsis**

```
void addExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: multiplier constant.

### Expr.addTerm()

Add a term to expression object.

**Synopsis**

```
void addTerm(Var var, double coeff)
```

**Arguments**

`var`: a variable for new term.

`coeff`: coefficient for new term.

### Expr.addTerms()

Add terms to expression object.

**Synopsis**

```
void addTerms(Var[] vars, double coeff)
```

**Arguments**

`vars`: variables for added terms.

`coeff`: one coefficient for added terms.

### Expr.addTerms()

Add terms to expression object.

**Synopsis**

```
void addTerms(Var[] vars, double[] coeffs)
```

**Arguments**

`vars`: variables for added terms.

`coeffs`: coefficients array for added terms.

### Expr.addTerms()

Add terms to expression object.

**Synopsis**

```
void addTerms(VarArray vars, double coeff)
```

**Arguments**

`vars`: variables for added terms.

`coeff`: one coefficient for added terms.

### Expr.addTerms()

Add terms to expression object.

**Synopsis**

```
void addTerms(VarArray vars, double[] coeffs)
```

**Arguments**

`vars`: variables for added terms.

`coeffs`: coefficients array for added terms.

### Expr.clone()

Deep copy linear expression object.

**Synopsis**

```
Expr clone()
```

**Return**

cloned linear expression object.

### Expr.divide()

Divide itself by double constant.

**Synopsis**

```
Expr divide(double c)
```

**Arguments**

`c`: constant operand.

**Return**

linear expression itself.

### Expr.evaluate()

Evaluate linear expression after solving.

**Synopsis**

```
double evaluate()
```

**Return**

value of linear expression.

### Expr.getCoeff()

Get coefficient from the i-th term in expression.

**Synopsis**

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

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term in expression object.

### Expr.getConstant()

Get constant in expression.

**Synopsis**

```
double getConstant()
```

**Return**

constant in expression.

### Expr.getVar()

Get variable from the i-th term in expression.

**Synopsis**

```
Var getVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

variable of the i-th term in expression object.

### Expr.multiply()

Multiply itself by double constant.

**Synopsis**

```
Expr multiply(double c)
```

**Arguments**

`c`: constant operand.

**Return**

linear expression itself.

### Expr.remove()

Remove idx-th term from expression object.

**Synopsis**

```
void remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

#### Expr.remove()

Remove the term associated with variable from expression.

**Synopsis**

```
void remove(Var var)
```

**Arguments**

`var`: a variable whose term should be removed.

#### Expr.setCoeff()

Set coefficient for the i-th term in expression.

**Synopsis**

```
void setCoeff(int i, double val)
```

**Arguments**

`i`: index of the term.

`val`: coefficient of the term.

#### Expr.setConstant()

Set constant for the expression.

**Synopsis**

```
void setConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### Expr.size()

Get number of terms in expression.

**Synopsis**

```
long size()
```

**Return**

number of terms.

### 27.2.7 Constraint

COPT constraint object. Constraints are always associated with a particular model. User creates a constraint object by adding a constraint to a model, rather than by using constructor of Constraint class.

### Constraint.computeShadowPriceRange()

Compute the shadow price range for a given constraint.

**Synopsis**

```
double[] computeShadowPriceRange(int ifRhs)
```

**Arguments**

`ifRhs`: for range constraint, whether to use RHS for computing shadow price.

**Return**

lower and upper bound of the shadow price.

### Constraint.get()

Get information value of the constraint. Support information of "Dual", "Slack", "LB" and "UB".

**Synopsis**

```
double get(String info)
```

**Arguments**

`info`: name of the information being queried.

**Return**

information value.

### Constraint.getBasis()

Get basis status of this constraint.

**Synopsis**

```
int getBasis()
```

**Return**

basis status.

### Constraint.getIdx()

Get index of the constraint.

**Synopsis**

```
int getIdx()
```

**Return**

the index of the constraint.

### Constraint.getLowerIIS()

Get IIS status for lower bound of the constraint.

**Synopsis**

```
int getLowerIIS()
```

**Return**

IIS status.

### Constraint.getName()

Get name of the constraint.

**Synopsis**

```
String getName()
```

**Return**

the name of the constraint.

### Constraint.getUpperIIS()

Get IIS status for upper bound of the constraint.

**Synopsis**

```
int getUpperIIS()
```

**Return**

IIS status.

### Constraint.remove()

Remove this constraint from model.

**Synopsis**

```
void remove()
```

### Constraint.set()

Set information value of the constraint. Support information of "LB" and "UB".

**Synopsis**

```
void set(String info, double val)
```

**Arguments**

`info`: name of the information.

`val`: new information value.

#### Constraint.setName()

Set name for the constraint.

**Synopsis**

```
void setName(String name)
```

**Arguments**

`name`: the name to set.

### 27.2.8 ConstrArray

COPT constraint array object. To store and access a set of Java *Constraint* objects, Cardinal Optimizer provides Java ConstrArray class, which defines the following methods.

#### ConstrArray.ConstrArray()

Constructor of constrarray object.

**Synopsis**

```
ConstrArray()
```

#### ConstrArray.getConstr()

Get idx-th constraint object.

**Synopsis**

```
Constraint getConstr(int idx)
```

**Arguments**

`idx`: index of the constraint.

**Return**

constraint object with index idx.

#### ConstrArray.pushBack()

Add a constraint object to constraint array.

**Synopsis**

```
void pushBack(Constraint constr)
```

**Arguments**

`constr`: a constraint object.

#### ConstrArray.size()

Get the number of constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of constraint objects.

### 27.2.9 ConstrBuilder

COPT constraint builder object. To help building a constraint, given a linear expression, constraint sense and right-hand side value, Cardinal Optimizer provides Java ConstrBuilder class, which defines the following methods.

#### ConstrBuilder.ConstrBuilder()

Constructor of constrbuilder object.

**Synopsis**

```
ConstrBuilder()
```

#### ConstrBuilder.getExpr()

Get expression associated with constraint.

**Synopsis**

```
Expr getExpr()
```

**Return**

expression object.

#### ConstrBuilder.getRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double getRange()
```

**Return**

length from lower bound to upper bound of the constraint.

#### ConstrBuilder.getSense()

Get sense associated with constraint.

**Synopsis**

```
char getSense()
```

**Return**

constraint sense.

#### ConstrBuilder.set()

Set detail of a constraint to its builder object.

**Synopsis**

```
void set(
    Expr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of the constraint.

#### ConstrBuilder.setRange()

Set a range constraint to its builder.

**Synopsis**

```
void setRange(Expr expr, double range)
```

**Arguments**

`expr`: expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must be greater than 0.

### 27.2.10 ConstrBuilderArray

COPT constraint builder array object. To store and access a set of Java *ConstrBuilder* objects, Cardinal Optimizer provides Java ConstrBuilderArray class, which defines the following methods.

#### ConstrBuilderArray.ConstrBuilderArray()

Constructor of constrbuilderarray object.

**Synopsis**

```
ConstrBuilderArray()
```

#### ConstrBuilderArray.getBuilder()

Get idx-th constraint builder object.

**Synopsis**

```
ConstrBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the constraint builder.

**Return**

constraint builder object with index idx.

#### ConstrBuilderArray.pushBack()

Add a constraint builder object to constraint builder array.

**Synopsis**

```
void pushBack(ConstrBuilder builder)
```

**Arguments**

`builder`: a constraint builder object.

#### ConstrBuilderArray.size()

Get the number of constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of constraint builder objects.

### 27.2.11 Column

COPT column object. A column consists of a list of constraints and associated coefficients. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients.

#### Column.Column()

Constructor of column.

**Synopsis**

```
Column()
```

#### Column.addColumn()

Add a column to self.

**Synopsis**

```
void addColumn(Column col, double mult)
```

**Arguments**

`col`: column object to be added.

`mult`: constant multiplier.

#### Column.addColumn()

Add a column to self.

**Synopsis**

```
void addColumn(Column col)
```

**Arguments**

`col`: column object to be added.

### Column.addTerm()

Add a term to column object.

**Synopsis**

```
void addTerm(Constraint constr, double coeff)
```

**Arguments**

`constr`: a constraint for new term.

`coeff`: coefficient for new term.

### Column.addTerms()

Add terms to column object.

**Synopsis**

```
void addTerms(Constraint[] constrs, double coeff)
```

**Arguments**

`constrs`: constraints for added terms.

`coeff`: coefficient for added terms.

### Column.addTerms()

Add terms to column object.

**Synopsis**

```
void addTerms(Constraint[] constrs, double[] coeffs)
```

**Arguments**

`constrs`: constraints for added terms.

`coeffs`: coefficients for added terms.

### Column.addTerms()

Add terms to column object.

**Synopsis**

```
void addTerms(ConstrArray constrs, double coeff)
```

**Arguments**

`constrs`: constraints for added terms.

`coeff`: coefficient for added terms.

### Column.addTerms()

Add terms to column object.

**Synopsis**

```
void addTerms(ConstrArray constrs, double[] coeffs)
```

**Arguments**

`constrs`: constraints for added terms.

`coeffs`: coefficients for added terms.

### Column.clear()

Clear all terms.

**Synopsis**

```
void clear()
```

### Column.clone()

Deep copy column object.

**Synopsis**

```
Column clone()
```

**Return**

cloned column object.

### Column.getCoeff()

Get coefficient from the i-th term in column object.

**Synopsis**

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

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term in column object.

### Column.getConstr()

Get constraint from the i-th term in column object.

**Synopsis**

```
Constraint getConstr(int i)
```

**Arguments**

`i`: index of the term.

**Return**

constraint of the i-th term in column object.

#### Column.remove()

Remove i-th term from column object.

**Synopsis**

```
void remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

#### Column.remove()

Remove the term associated with constraint from column object.

**Synopsis**

```
void remove(Constraint constr)
```

**Arguments**

`constr`: a constraint whose term should be removed.

#### Column.size()

Get number of terms in column object.

**Synopsis**

```
int size()
```

**Return**

number of terms.

### 27.2.12 ColumnArray

COPT column array object. To store and access a set of Java *Column* objects, Cardinal Optimizer provides Java ColumnArray class, which defines the following methods.

#### ColumnArray.ColumnArray()

Constructor of columnarray object.

**Synopsis**

```
ColumnArray()
```

#### ColumnArray.clear()

Clear all column objects.

**Synopsis**

```
void clear()
```

#### ColumnArray.getColumn()

Get idx-th column object.

**Synopsis**

```
Column getColumn(int idx)
```

**Arguments**

`idx`: index of the column.

**Return**

column object with index idx.

#### ColumnArray.pushBack()

Add a column object to column array.

**Synopsis**

```
void pushBack(Column col)
```

**Arguments**

`col`: a column object.

#### ColumnArray.size()

Get the number of column objects.

**Synopsis**

```
int size()
```

**Return**

number of column objects.

### 27.2.13 Sos

COPT SOS constraint object. SOS constraints are always associated with a particular model. User creates an SOS constraint object by adding an SOS constraint to a model, rather than by using constructor of Sos class.

An SOS constraint can be type 1 or 2 (`COPT_SOS_TYPE1` or `COPT_SOS_TYPE2`).

#### Sos.getIdx()

Get the index of SOS constraint.

**Synopsis**

```
int getIdx()
```

**Return**

index of SOS constraint.

#### Sos.getIIS()

Get IIS status of the SOS constraint.

**Synopsis**

```
int getIIS()
```

**Return**

IIS status.

#### Sos.remove()

Remove the SOS constraint from model.

**Synopsis**

```
void remove()
```

### 27.2.14 SosArray

COPT SOS constraint array object. To store and access a set of Java *Sos* objects, Cardinal Optimizer provides Java SosArray class, which defines the following methods.

#### SosArray.SosArray()

Constructor of sosarray object.

**Synopsis**

```
SosArray()
```

#### SosArray.getSos()

Get idx-th SOS object.

**Synopsis**

```
Sos getSos(int idx)
```

**Arguments**

`idx`: index of SOS.

**Return**

SOS object with index idx.

#### SosArray.pushBack()

Add a SOS constraint object to SOS constraint array.

**Synopsis**

```
void pushBack(Sos sos)
```

**Arguments**

`sos`: a SOS constraint object.

#### SosArray.size()

Get the number of SOS constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of SOS constraint objects.

### 27.2.15 SosBuilder

COPT SOS constraint builder object. To help building an SOS constraint, given the SOS type, a set of variables and associated weights, Cardinal Optimizer provides Java SosBuilder class, which defines the following methods.

#### SosBuilder.SosBuilder()

Constructor of sosbuilder object.

**Synopsis**

```
SosBuilder()
```

#### SosBuilder.getSize()

Get number of terms in SOS constraint.

**Synopsis**

```
int getSize()
```

**Return**

number of terms.

#### SosBuilder.getType()

Get type of SOS constraint.

**Synopsis**

```
int getType()
```

**Return**

type of SOS constraint.

#### SosBuilder.getVar()

Get variable from the idx-th term in SOS constraint.

**Synopsis**

```
Var getVar(int idx)
```

**Arguments**

`idx`: index of the term.

**Return**

variable of the idx-th term in SOS constraint.

### SosBuilder.GetVars()

Get all variables in a SOS constraint.

**Synopsis**

```
VarArray GetVars()
```

**Return**

variables in a SOS constraint.

### SosBuilder.getWeight()

Get weight from the idx-th term in SOS constraint.

**Synopsis**

```
double getWeight(int idx)
```

**Arguments**

`idx`: index of the term.

**Return**

weight of the idx-th term in SOS constraint.

### SosBuilder.getWeights()

Get weights of all terms in SOS constraint.

**Synopsis**

```
double[] getWeights()
```

**Return**

array of weights.

### SosBuilder.set()

Set variables and weights of SOS constraint.

**Synopsis**

```
void set(
    VarArray vars,
    double[] weights,
    int type)
```

**Arguments**

`vars`: variable array object.

`weights`: array of weights.

`type`: type of SOS constraint.

### 27.2.16 SosBuilderArray

COPT SOS constraint builder array object. To store and access a set of Java *SosBuilder* objects, Cardinal Optimizer provides Java SosBuilderArray class, which defines the following methods.

#### SosBuilderArray.SosBuilderArray()

Constructor of sosbuilderarray object.

**Synopsis**

```
SosBuilderArray()
```

#### SosBuilderArray.getBuilder()

Get idx-th SOS constraint builder object.

**Synopsis**

```
SosBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the SOS constraint builder.

**Return**

SOS constraint builder object with index idx.

#### SosBuilderArray.pushBack()

Add a SOS constraint builder object to SOS constraint builder array.

**Synopsis**

```
void pushBack(SosBuilder builder)
```

**Arguments**

`builder`: a SOS constraint builder object.

#### SosBuilderArray.size()

Get the number of SOS constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of SOS constraint builder objects.

### 27.2.17 GenConstr

COPT general constraint object. General constraints are always associated with a particular model. User creates a general constraint object by adding a general constraint to a model, rather than by using constructor of GenConstr class.

#### GenConstr.getIdx()

Get the index of the general constraint.

**Synopsis**

```
int getIdx()
```

**Return**

index of the general constraint.

#### GenConstr.getIIS()

Get IIS status of the general constraint.

**Synopsis**

```
int getIIS()
```

**Return**

IIS status.

#### GenConstr.getName()

Get name of general constraint.

**Synopsis**

```
String getName()
```

**Return**

the name of general constraint.

#### GenConstr.remove()

Remove the general constraint from model.

**Synopsis**

```
void remove()
```

#### GenConstr.setName()

Set name for general constraint.

**Synopsis**

```
void setName(String name)
```

**Arguments**

`name`: the name to set.

### 27.2.18 GenConstrArray

COPT general constraint array object. To store and access a set of Java *GenConstr* objects, Cardinal Optimizer provides Java GenConstrArray class, which defines the following methods.

#### GenConstrArray.GenConstrArray()

Constructor of genconstrarray.

**Synopsis**

```
GenConstrArray()
```

#### GenConstrArray.getGenConstr()

Get idx-th general constraint object.

**Synopsis**

```
GenConstr getGenConstr(int idx)
```

**Arguments**

`idx`: index of the general constraint.

**Return**

general constraint object with index idx.

#### GenConstrArray.pushBack()

Add a general constraint object to general constraint array.

**Synopsis**

```
void pushBack(GenConstr genconstr)
```

**Arguments**

`genconstr`: a general constraint object.

#### GenConstrArray.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: capacity number of general constraint object.

#### GenConstrArray.size()

Get the number of general constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of general constraint objects.

### 27.2.19 GenConstrBuilder

COPT general constraint builder object. To help building a general constraint, given a binary variable and associated value, a linear expression and constraint sense, Cardinal Optimizer provides Java GenConstrBuilder class, which defines the following methods.

#### GenConstrBuilder.GenConstrBuilder()

Constructor of genconstrbuilder.

**Synopsis**

```
GenConstrBuilder()
```

#### GenConstrBuilder.getBinVal()

Get binary value associated with general constraint.

**Synopsis**

```
int getBinVal()
```

**Return**

binary value.

#### GenConstrBuilder.getBinVar()

Get binary variable associated with general constraint.

**Synopsis**

```
Var getBinVar()
```

**Return**

binary vaiable object.

#### GenConstrBuilder.getExpr()

Get expression associated with general constraint.

**Synopsis**

```
Expr getExpr()
```

**Return**

expression object.

### GenConstrBuilder.getIndType()

Get type of general constraint.

**Synopsis**

```
int getIndType()
```

**Return**

type of general constraint.

### GenConstrBuilder.getSense()

Get sense associated with general constraint.

**Synopsis**

```
char getSense()
```

**Return**

constraint sense.

### GenConstrBuilder.set()

Set binary variable, binary value, expression and sense of general constraint.

**Synopsis**

```
void set(
    Var binvar,
    int binval,
    Expr expr,
    char sense)
```

**Arguments**

`binvar`: binary variable.

`binval`: binary value.

`expr`: expression object.

`sense`: general constraint sense.

### GenConstrBuilder.set()

Set binary variable, binary value, expression and sense of general constraint.

**Synopsis**

```
void set(
    Var binvar,
    int binval,
    Expr expr,
    char sense,
    int type)
```

**Arguments**

`binvar`: binary variable.

`binval`: binary value.

`expr`: expression object.

`sense`: general constraint sense.

`type`: type of general constraint.

### 27.2.20 GenConstrBuilderArray

COPT general constraint builder array object. To store and access a set of Java *GenConstrBuilder* objects, Cardinal Optimizer provides Java GenConstrBuilderArray class, which defines the following methods.

#### GenConstrBuilderArray.GenConstrBuilderArray()

Constructor of genconstrbuilderarray.

**Synopsis**

```
GenConstrBuilderArray()
```

#### GenConstrBuilderArray.getBuilder()

Get idx-th general constraint builder object.

**Synopsis**

```
GenConstrBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the general constraint builder.

**Return**

general constraint builder object with index idx.

#### GenConstrBuilderArray.pushBack()

Add a general constraint builder object to general constraint builder array.

**Synopsis**

```
void pushBack(GenConstrBuilder builder)
```

**Arguments**

`builder`: a general constraint builder object.

#### GenConstrBuilderArray.size()

Get the number of general constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of general constraint builder objects.

### 27.2.21 Cone

COPT cone constraint object. Cone constraints are always associated with a particular model. User creates a cone constraint object by adding a cone constraint to a model, rather than by using constructor of Cone class.

A cone constraint can be regular or rotated (`COPT_CONE_QUAD` or `COPT_CONE_RQUAD`).

#### Cone.getIdx()

Get the index of a cone constraint.

**Synopsis**

```
int getIdx()
```

**Return**

index of a cone constraint.

#### Cone.remove()

Remove the cone constraint from model.

**Synopsis**

```
void remove()
```

### 27.2.22 ConeArray

COPT cone constraint array object. To store and access a set of Java *Cone* objects, Cardinal Optimizer provides Java ConeArray class, which defines the following methods.

#### ConeArray.ConeArray()

Constructor of ConeArray object.

**Synopsis**

```
ConeArray()
```

#### ConeArray.getCone()

Get idx-th cone object.

**Synopsis**

```
Cone getCone(int idx)
```

**Arguments**

`idx`: index of cone.

**Return**

cone object with index idx.

#### ConeArray.pushBack()

Add a cone constraint object to cone constraint array.

**Synopsis**

```
void pushBack(Cone cone)
```

**Arguments**

`cone`: a cone constraint object.

#### ConeArray.size()

Get the number of cone constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of cone constraint objects.

### 27.2.23 ConeBuilder

COPT cone constraint builder object. To help building a cone constraint, given the cone type and a set of variables, Cardinal Optimizer provides Java ConeBuilder class, which defines the following methods.

#### ConeBuilder.ConeBuilder()

Constructor of ConeBuilder object.

**Synopsis**

```
ConeBuilder()
```

### ConeBuilder.getSize()

Get number of variables in a cone constraint.

**Synopsis**

```
int getSize()
```

**Return**

number of vars.

### ConeBuilder.getType()

Get type of a cone constraint.

**Synopsis**

```
int getType()
```

**Return**

type of a cone constraint.

### ConeBuilder.getVar()

Get idx-th variable in a cone constraint.

**Synopsis**

```
Var getVar(int idx)
```

**Arguments**

`idx`: index of variables.

**Return**

the idx-th variable in a cone constraint.

### ConeBuilder.getVars()

Get all variables in a cone constraint.

**Synopsis**

```
VarArray getVars()
```

**Return**

variables in a cone constraint.

### ConeBuilder.set()

Set variables and type of a cone constraint.

**Synopsis**

```
void set(VarArray vars, int type)
```

**Arguments**

`vars`: variable array object.

`type`: type of a cone constraint.

### 27.2.24 ConeBuilderArray

COPT cone constraint builder array object. To store and access a set of Java *ConeBuilder* objects, Cardinal Optimizer provides Java ConeBuilderArray class, which defines the following methods.

#### ConeBuilderArray.ConeBuilderArray()

Constructor of ConeBuilderArray object.

**Synopsis**

```
ConeBuilderArray()
```

#### ConeBuilderArray.getBuilder()

Get idx-th cone constraint builder object.

**Synopsis**

```
ConeBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the cone constraint builder.

**Return**

cone constraint builder object with index idx.

#### ConeBuilderArray.pushBack()

Add a cone constraint builder object to cone constraint builder array.

**Synopsis**

```
void pushBack(ConeBuilder builder)
```

**Arguments**

`builder`: a cone constraint builder object.

#### ConeBuilderArray.size()

Get the number of cone constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of cone constraint builder objects.

### 27.2.25 ExpCone

COPT exponential cone constraint object. ExpCone constraints are always associated with a particular model. User creates an exponential cone constraint object by adding an exponential cone constraint to a model, rather than by using constructor of ExpCone class.

#### ExpCone.getIdx()

Get the index of an exponential cone constraint.

**Synopsis**

```
int getIdx()
```

**Return**

index of an exponential cone constraint.

#### ExpCone.remove()

Remove the exponential cone constraint from model.

**Synopsis**

```
void remove()
```

### 27.2.26 ExpConeArray

COPT exponential cone constraint array object. To store and access a set of Java *ExpCone* objects, Cardinal Optimizer provides Java ExpConeArray class, which defines the following methods.

#### ExpConeArray.ExpConeArray()

Constructor of ExpConeArray object.

**Synopsis**

```
ExpConeArray()
```

#### ExpConeArray.getCone()

Get idx-th exponential cone object.

**Synopsis**

```
ExpCone getCone(int idx)
```

**Arguments**

`idx`: index of exponential cone.

**Return**

exponential cone object with index idx.

#### ExpConeArray.pushBack()

Add an exponential cone constraint object to exponential cone constraint array.

**Synopsis**

```
void pushBack(ExpCone cone)
```

**Arguments**

`cone`: an exponential cone constraint object.

#### ExpConeArray.size()

Get the number of exponential cone constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of exponential cone constraint objects.

### 27.2.27 ExpConeBuilder

COPT exponential cone constraint builder object. To help building an exponential cone constraint, given the exponential cone type and a set of variables, Cardinal Optimizer provides Java ExpConeBuilder class, which defines the following methods.

#### ExpConeBuilder.ExpConeBuilder()

Constructor of ExpConeBuilder object.

**Synopsis**

```
ExpConeBuilder()
```

#### ExpConeBuilder.getSize()

Get number of variables in an exponential cone constraint.

**Synopsis**

```
int getSize()
```

**Return**

number of vars.

#### ExpConeBuilder.getType()

Get type of an exponential cone constraint.

**Synopsis**

```
int getType()
```

**Return**

type of an exponential cone constraint.

#### ExpConeBuilder.getVar()

Get idx-th variable in an exponential cone constraint.

**Synopsis**

```
Var getVar(int idx)
```

**Arguments**

`idx`: index of variables.

**Return**

the idx-th variable in an exponential cone constraint.

#### ExpConeBuilder.getVars()

Get all variables in an exponential cone constraint.

**Synopsis**

```
VarArray getVars()
```

**Return**

variables in an exponential cone constraint.

#### ExpConeBuilder.set()

Set variables and type of an exponential cone constraint.

**Synopsis**

```
void set(VarArray vars, int type)
```

**Arguments**

`vars`: variable array object.

`type`: type of an exponential cone constraint.

### 27.2.28 ExpConeBuilderArray

COPT exponential cone constraint builder array object. To store and access a set of Java *ExpConeBuilder* objects, Cardinal Optimizer provides Java ExpConeBuilderArray class, which defines the following methods.

#### ExpConeBuilderArray.ExpConeBuilderArray()

Constructor of ExpConeBuilderArray object.

**Synopsis**

```
ExpConeBuilderArray()
```

#### ExpConeBuilderArray.getBuilder()

Get idx-th exponential cone constraint builder object.

**Synopsis**

```
ExpConeBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the exponential cone constraint builder.

**Return**

exponential cone constraint builder object with index idx.

#### ExpConeBuilderArray.pushBack()

Add an exponential cone constraint builder object to exponential cone constraint builder array.

**Synopsis**

```
void pushBack(ExpConeBuilder builder)
```

**Arguments**

`builder`: an exponential cone constraint builder object.

#### ExpConeBuilderArray.size()

Get the number of exponential cone constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of exponential cone constraint builder objects.

### 27.2.29 AffineCone Class

The *AffineCone* class in COPT encapsulates operations related to affine cones. The following methods are provided:

#### AffineCone.getIdx()

Get the index of an affine cone constraint.

**Synopsis**

```
int getIdx()
```

**Return**

index of an affine cone constraint.

#### AffineCone.getName()

Get name of the affine cone.

**Synopsis**

```
String getName()
```

**Return**

affine cone name.

#### AffineCone.remove()

Remove the affine cone constraint from model.

**Synopsis**

```
void remove()
```

#### AffineCone.setName()

Set name of the affine cone.

**Synopsis**

```
void setName(String name)
```

**Arguments**

`name`: affinecone name.

### 27.2.30 AffineConeArray Class

To facilitate user operations on a group of Java *AffineCone Class* objects, the Java interface of COPT introduces the *AffineConeArray* class. The following methods are provided:

#### AffineConeArray.AffineConeArray()

Constructor of AffineConeArray object.

**Synopsis**

```
AffineConeArray()
```

#### AffineConeArray.getCone()

Get idx-th affine cone object.

**Synopsis**

```
AffineCone getCone(int idx)
```

**Arguments**

`idx`: index of affine cone.

**Return**

affine cone object with index idx.

#### AffineConeArray.pushBack()

Add an affine cone constraint object to affine cone constraint array.

**Synopsis**

```
void pushBack(AffineCone cone)
```

**Arguments**

`cone`: an affine cone constraint object.

#### AffineConeArray.size()

Get the number of affine cone constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of affine cone constraint objects.

### 27.2.31 AffineConeBuilder Class

The *AffineConeBuilder* class in COPT encapsulates the builder for constructing affine cones. The following methods are provided:

#### AffineConeBuilder.AffineConeBuilder()

Constructor of AffineConeBuilder object.

**Synopsis**

```
AffineConeBuilder()
```

#### AffineConeBuilder.getExpr()

Get idx-th linear expression in an affine cone constraint.

**Synopsis**

```
Expr getExpr(int idx)
```

**Arguments**

`idx`: index of linear expression.

**Return**

the idx-th linear expression in an affine cone constraint.

### AffineConeBuilder.getExprs()

Get all linear expressions in an affine cone constraint.

**Synopsis**

```
Expr[] getExprs()
```

**Return**

array of linear expressions.

### AffineConeBuilder.getPsdExpr()

Get idx-th PSD expression in an affine cone constraint.

**Synopsis**

```
PsdExpr getPsdExpr(int idx)
```

**Arguments**

`idx`: index of PSD expression.

**Return**

the idx-th PSD expression in an affine cone constraint.

### AffineConeBuilder.getPsdExprs()

Get all PSD expressions in an affine cone constraint.

**Synopsis**

```
PsdExpr[] getPsdExprs()
```

**Return**

array of PSD expressions.

### AffineConeBuilder.getSize()

Get number of variables in an affine cone constraint.

**Synopsis**

```
int getSize()
```

**Return**

number of vars.

### AffineConeBuilder.getType()

Get type of an affine cone constraint.

**Synopsis**

```
int getType()
```

**Return**

type of an affine cone constraint.

#### AffineConeBuilder.hasPsdTerm()

Check whether affine cone has PSD terms.

**Synopsis**

```
Boolean hasPsdTerm()
```

**Return**

flag to indicate whether affine cone has PSD terms.

#### AffineConeBuilder.set()

Set linear expressions and type of an affine cone constraint.

**Synopsis**

```
void set(Expr[] exprs, int type)
```

**Arguments**

`exprs`: array of linear expressions.

`type`: type of an affine cone constraint.

#### AffineConeBuilder.set()

Set PSD expressions and type of an affine cone constraint.

**Synopsis**

```
void set(PsdExpr[] exprs, int type)
```

**Arguments**

`exprs`: array of PSD expressions.

`type`: type of an affine cone constraint.

### 27.2.32 AffineConeBuilderArray Class

To facilitate operations on a group of Java *AffineConeBuilder Class* objects, the Java interface of COPT introduces the *AffineConeBuilderArray* class. The following methods are provided:

#### AffineConeBuilderArray.AffineConeBuilderArray()

Constructor of AffineConeBuilderArray object.

**Synopsis**

```
AffineConeBuilderArray()
```

#### AffineConeBuilderArray.getBuilder()

Get idx-th affine cone constraint builder object.

**Synopsis**

```
AffineConeBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the affine cone constraint builder.

**Return**

affine cone constraint builder object with index idx.

#### AffineConeBuilderArray.pushBack()

Add an affine cone constraint builder object to affine cone constraint builder array.

**Synopsis**

```
void pushBack(AffineConeBuilder builder)
```

**Arguments**

`builder`: an affine cone constraint builder object.

#### AffineConeBuilderArray.size()

Get the number of affine cone constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of affine cone constraint builder objects.

### 27.2.33 QuadExpr

COPT quadratic expression object. A quadratic expression consists of a linear expression, a list of variable pairs and associated coefficients of quadratic terms. Quadratic expressions are used to build quadratic constraints and objectives.

#### QuadExpr.QuadExpr()

Constructor of a constant quadratic expression with constant 0.0

**Synopsis**

```
QuadExpr()
```

### QuadExpr.QuadExpr()

Constructor of a constant quadratic expression.

**Synopsis**

```
QuadExpr(double constant)
```

**Arguments**

`constant`: constant value in expression object.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with one term.

**Synopsis**

```
QuadExpr(Var var)
```

**Arguments**

`var`: variable for the added term.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with one term.

**Synopsis**

```
QuadExpr(Var var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with a linear expression.

**Synopsis**

```
QuadExpr(Expr expr)
```

**Arguments**

`expr`: linear expression added to the quadratic expression.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

**Synopsis**

```
QuadExpr(Var var1, Var var2)
```

**Arguments**

`var1`: one variable.

`var2`: another variable.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

**Synopsis**

```
QuadExpr(Expr expr, Var var)
```

**Arguments**

`expr`: one linear expression.

`var`: another variable.

### QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

**Synopsis**

```
QuadExpr(Expr left, Expr right)
```

**Arguments**

`left`: one linear expression.

`right`: another linear expression.

### QuadExpr.add()

Add itself by a quadratic expression.

**Synopsis**

```
QuadExpr add(QuadExpr expr, double mult)
```

**Arguments**

`expr`: expression operand, including QuadExpr, Expr, Var and constant.

`mult`: constant multiplier.

**Return**

quadratic expression itself.

### QuadExpr.add()

Add itself by a quadratic expression.

**Synopsis**

```
QuadExpr add(QuadExpr expr)
```

**Arguments**

`expr`: expression operand, including QuadExpr, Expr, Var and constant.

**Return**

quadratic expression itself.

### QuadExpr.addConstant()

Add a constant to the quadratic expression.

**Synopsis**

```
void addConstant(double constant)
```

**Arguments**

`constant`: value to be added.

### QuadExpr.addLinExpr()

Add a linear expression to self.

**Synopsis**

```
void addLinExpr(Expr expr)
```

**Arguments**

`expr`: linear expression to be added.

### QuadExpr.addLinExpr()

Add a linear expression to self.

**Synopsis**

```
void addLinExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: multiplier constant.

### QuadExpr.addQuadExpr()

Add a quadratic expression to self.

**Synopsis**

```
void addQuadExpr(QuadExpr expr)
```

**Arguments**

`expr`: quadratic expression to be added.

### QuadExpr.addQuadExpr()

Add a quadratic expression to self.

**Synopsis**

```
void addQuadExpr(QuadExpr expr, double mult)
```

**Arguments**

`expr`: quadratic expression to be added.

`mult`: multiplier constant.

### QuadExpr.addTerm()

Add a term to quadratic expression object.

**Synopsis**

```
void addTerm(Var var, double coeff)
```

**Arguments**

`var`: a variable of new term.

`coeff`: coefficient of new term.

### QuadExpr.addTerm()

Add a quadratic term to expression object.

**Synopsis**

```
void addTerm(
    Var var1,
    Var var2,
    double coeff)
```

**Arguments**

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

### QuadExpr.addTerms()

Add linear terms to quadratic expression object.

**Synopsis**

```
void addTerms(Var[] vars, double coeff)
```

**Arguments**

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

### QuadExpr.addTerms()

Add linear terms to quadratic expression object.

**Synopsis**

```
void addTerms(Var[] vars, double[] coeffs)
```

**Arguments**

`vars`: variables of added linear terms.

`coeffs`: coefficients of added linear terms.

### QuadExpr.addTerms()

Add linear terms to quadratic expression object.

**Synopsis**

```
void addTerms(VarArray vars, double coeff)
```

**Arguments**

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

### QuadExpr.addTerms()

Add linear terms to quadratic expression object.

**Synopsis**

```
void addTerms(VarArray vars, double[] coeffs)
```

**Arguments**

`vars`: variables of added terms.

`coeffs`: coefficients of added terms.

### QuadExpr.addTerms()

Add quadratic terms to expression object.

**Synopsis**

```
void addTerms(
    VarArray vars1,
    VarArray vars2,
    double[] coeffs)
```

**Arguments**

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`coeffs`: coefficient array for added quadratic terms.

### QuadExpr.addTerms()

Add quadratic terms to expression object.

**Synopsis**

```
void addTerms(
    Var[] vars1,
    Var[] vars2,
    double[] coeffs)
```

**Arguments**

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`coeffs`: coefficient array for added quadratic terms.

### QuadExpr.clone()

Deep copy quadratic expression object.

**Synopsis**

```
QuadExpr clone()
```

**Return**

cloned quadratic expression object.

### QuadExpr.divide()

Divide itself by double constant.

**Synopsis**

```
QuadExpr divide(double c)
```

**Arguments**

`c`: constant operand.

**Return**

quadratic expression itself.

### QuadExpr.evaluate()

evaluate quadratic expression after solving.

**Synopsis**

```
double evaluate()
```

**Return**

value of quadratic expression.

### QuadExpr.getCoeff()

Get coefficient from the i-th term in quadratic expression.

**Synopsis**

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

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term in quadratic expression object.

### QuadExpr.getConstant()

Get constant in quadratic expression.

**Synopsis**

```
double getConstant()
```

**Return**

constant in quadratic expression.

### QuadExpr.getLinExpr()

Get linear expression in quadratic expression.

**Synopsis**

```
Expr getLinExpr()
```

**Return**

linear expression object.

### QuadExpr.getVar1()

Get first variable from the i-th term in quadratic expression.

**Synopsis**

```
Var getVar1(int i)
```

**Arguments**

`i`: index of the term.

**Return**

first variable of the i-th term in quadratic expression object.

### QuadExpr.getVar2()

Get second variable from the i-th term in quadratic expression.

**Synopsis**

```
Var getVar2(int i)
```

**Arguments**

`i`: index of the term.

**Return**

second variable of the i-th term in quadratic expression object.

### QuadExpr.multiply()

Multiply itself by double constant.

**Synopsis**

```
QuadExpr multiply(double c)
```

**Arguments**

`c`: constant operand.

**Return**

quadratic expression itself.

### QuadExpr.remove()

Remove idx-th term from quadratic expression object.

**Synopsis**

```
void remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### QuadExpr.remove()

Remove the term associated with variable from quadratic expression.

**Synopsis**

```
void remove(Var var)
```

**Arguments**

`var`: a variable whose term should be removed.

### QuadExpr.setCoeff()

Set coefficient of the i-th term in quadratic expression.

**Synopsis**

```
void setCoeff(int i, double val)
```

**Arguments**

`i`: index of the quadratic term.

`val`: coefficient of the term.

### QuadExpr.setConstant()

Set constant for the quadratic expression.

**Synopsis**

```
void setConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### QuadExpr.size()

Get number of terms in quadratic expression.

**Synopsis**

```
long size()
```

**Return**

number of quadratic terms.

### 27.2.34 QConstraint

COPT quadratic constraint object. Quadratic constraints are always associated with a particular model. User creates a quadratic constraint object by adding a quadratic constraint to a model, rather than by using constructor of QConstraint class.

#### QConstraint.get()

Get information value of the quadratic constraint.

**Synopsis**

```
double get(String info)
```

**Arguments**

`info`: name of the information being queried.

**Return**

information value.

#### QConstraint.getIdx()

Get index of the quadratic constraint.

**Synopsis**

```
int getIdx()
```

**Return**

the index of the quadratic constraint.

#### QConstraint.getName()

Get name of the constraint.

**Synopsis**

```
String getName()
```

**Return**

the name of the constraint.

### QConstraint.getRhs()

Get rhs of quadratic constraint.

**Synopsis**

```
double getRhs()
```

**Return**

rhs of quadratic constraint.

### QConstraint.getSense()

Get rhs of quadratic constraint.

**Synopsis**

```
char getSense()
```

**Return**

rhs of quadratic constraint.

### QConstraint.remove()

Remove this constraint from model.

**Synopsis**

```
void remove()
```

### QConstraint.set()

Set information value of the quadratic constraint.

**Synopsis**

```
void set(String info, double val)
```

**Arguments**

`info`: name of the information.

`val`: new information value.

### QConstraint.setName()

Set name of quadratic constraint.

**Synopsis**

```
void setName(String name)
```

**Arguments**

`name`: the name to set.

#### QConstraint.setRhs()

Set rhs of quadratic constraint.

**Synopsis**

```
void setRhs(double rhs)
```

**Arguments**

`rhs`: rhs of quadratic constraint.

#### QConstraint.setSense()

Set sense of quadratic constraint.

**Synopsis**

```
void setSense(char sense)
```

**Arguments**

`sense`: sense of quadratic constraint.

### 27.2.35 QConstrArray

COPT quadratic constraint array object. To store and access a set of Java *QConstraint* objects, Cardinal Optimizer provides Java QConstrArray class, which defines the following methods.

#### QConstrArray.QConstrArray()

Constructor of constrarray object.

**Synopsis**

```
QConstrArray()
```

#### QConstrArray.getQConstr()

Get idx-th constraint object.

**Synopsis**

```
QConstraint getQConstr(int idx)
```

**Arguments**

`idx`: index of the constraint.

**Return**

constraint object with index idx.

#### QConstrArray.pushBack()

Add a constraint object to constraint array.

**Synopsis**

```
void pushBack(QConstraint constr)
```

**Arguments**

`constr`: a constraint object.

#### QConstrArray.size()

Get the number of constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of constraint objects.

### 27.2.36 QConstrBuilder

COPT quadratic constraint builder object. To help building a quadratic constraint, given a quadratic expression, constraint sense and right-hand side value, Cardinal Optimizer provides Java ConeBuilder class, which defines the following methods.

#### QConstrBuilder.QConstrBuilder()

Constructor of constrbuilder object.

**Synopsis**

```
QConstrBuilder()
```

#### QConstrBuilder.getQuadExpr()

Get expression associated with constraint.

**Synopsis**

```
QuadExpr getQuadExpr()
```

**Return**

quadratic expression object.

#### QConstrBuilder.getSense()

Get sense associated with quadratic constraint.

**Synopsis**

```
char getSense()
```

**Return**

quadratic constraint sense.

#### QConstrBuilder.set()

Set detail of a quadratic constraint to its builder object.

**Synopsis**

```
void set(
    QuadExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the quadratic constraint.

`sense`: quadratic constraint sense.

`rhs`: constant of right side of quadratic constraint.

### 27.2.37 QConstrBuilderArray

COPT quadratic constraint builder array object. To store and access a set of Java *QConstrBuilder* objects, Cardinal Optimizer provides Java QConstrBuilderArray class, which defines the following methods.

#### QConstrBuilderArray.QConstrBuilderArray()

Constructor of constrbuilderarray object.

**Synopsis**

```
QConstrBuilderArray()
```

#### QConstrBuilderArray.getBuilder()

Get idx-th constraint builder object.

**Synopsis**

```
QConstrBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the constraint builder.

**Return**

constraint builder object with index idx.

#### QConstrBuilderArray.pushBack()

Add a constraint builder object to constraint builder array.

**Synopsis**

```
void pushBack(QConstrBuilder builder)
```

**Arguments**

`builder`: a constraint builder object.

#### QConstrBuilderArray.size()

Get the number of constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of constraint builder objects.

### 27.2.38 PsdVar

COPT PSD variable object. PSD variables are always associated with a particular model. User creates a PSD variable object by adding a PSD variable to model, rather than by constructor of PsdVar class.

#### PsdVar.get()

Get information values of PSD variable.

**Synopsis**

```
double[] get(String info)
```

**Arguments**

`info`: information name.

**Return**

array of information values.

#### PsdVar.getDim()

Get dimension of PSD variable.

**Synopsis**

```
int getDim()
```

**Return**

dimension of PSD variable.

#### PsdVar.getIdx()

Get index of PSD variable.

**Synopsis**

```
int getIdx()
```

**Return**

index of PSD variable.

#### PsdVar.getLen()

Get length of PSD variable.

**Synopsis**

```
int getLen()
```

**Return**

length of PSD variable.

#### PsdVar.getName()

Get name of PSD variable.

**Synopsis**

```
String getName()
```

**Return**

name of PSD variable.

#### PsdVar.remove()

Remove PSD variable from model.

**Synopsis**

```
void remove()
```

### 27.2.39 PsdVarArray

COPT PSD variable array object. To store and access a set of *PsdVar* objects, Cardinal Optimizer provides PsdVarArray class, which defines the following methods.

#### PsdVarArray.PsdVarArray()

Constructor of PsdVarArray.

**Synopsis**

```
PsdVarArray()
```

#### PsdVarArray.getPsdVar()

Get idx-th PSD variable object.

**Synopsis**

```
PsdVar getPsdVar(int idx)
```

**Arguments**

`idx`: index of the PSD variable.

**Return**

PSD variable object with index idx.

#### PsdVarArray.pushBack()

Add a PSD variable object to PSD variable array.

**Synopsis**

```
void pushBack(PsdVar var)
```

**Arguments**

`var`: a PSD variable object.

#### PsdVarArray.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD variable object.

#### PsdVarArray.size()

Get the number of PSD variable objects.

**Synopsis**

```
int size()
```

**Return**

number of PSD variable objects.

### 27.2.40 PsdExpr

COPT PSD expression object. A PSD expression consists of a linear expression, a list of PSD variables and associated coefficient matrices of PSD terms. PSD expressions are used to build PSD constraints and objectives.

#### PsdExpr.PsdExpr()

Constructor of a PSD expression with default constant value 0.

**Synopsis**

```
PsdExpr(double constant)
```

**Arguments**

`constant`: constant value in PSD expression object.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(Var var)
```

**Arguments**

`var`: variable for the added term.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(Var var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with a linear expression.

**Synopsis**

```
PsdExpr(Expr expr)
```

**Arguments**

`expr`: input linear expression.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(PsdVar var, SymMatrix mat)
```

**Arguments**

`var`: PSD variable for the added term.

`mat`: coefficient matrix for the added term.

### PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

**Synopsis**

```
PsdExpr(PsdVar var, SymMatExpr expr)
```

**Arguments**

`var`: PSD variable for the added term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

### PsdExpr.addConstant()

Add constant to the PSD expression.

**Synopsis**

```
void addConstant(double constant)
```

**Arguments**

`constant`: value to be added.

### PsdExpr.addLinExpr()

Add a linear expression to PSD expression object.

**Synopsis**

```
void addLinExpr(Expr expr)
```

**Arguments**

`expr`: linear expression to be added.

### PsdExpr.addLinExpr()

Add a linear expression to PSD expression object.

**Synopsis**

```
void addLinExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: multiplier constant.

### PsdExpr.addPsdExpr()

Add a PSD expression to self.

**Synopsis**

```
void addPsdExpr(PsdExpr expr)
```

**Arguments**

`expr`: PSD expression to be added.

### PsdExpr.addPsdExpr()

Add a PSD expression to self.

**Synopsis**

```
void addPsdExpr(PsdExpr expr, double mult)
```

**Arguments**

`expr`: PSD expression to be added.

`mult`: multiplier constant.

### PsdExpr.addTerm()

Add a linear term to PSD expression object.

**Synopsis**

```
void addTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

### PsdExpr.addTerm()

Add a PSD term to PSD expression object.

**Synopsis**

```
void addTerm(PsdVar var, SymMatrix mat)
```

**Arguments**

`var`: PSD variable of new PSD term.

`mat`: coefficient matrix of new PSD term.

### PsdExpr.addTerm()

Add a PSD term to PSD expression object.

**Synopsis**

```
void addTerm(PsdVar var, SymMatExpr expr)
```

**Arguments**

`var`: PSD variable of new PSD term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

### PsdExpr.addTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
void addTerms(Var[] vars, double coeff)
```

**Arguments**

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

### PsdExpr.addTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
void addTerms(Var[] vars, double[] coeffs)
```

**Arguments**

`vars`: variables for added linear terms.

`coeffs`: coefficient array for added linear terms.

### PsdExpr.addTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
void addTerms(VarArray vars, double coeff)
```

**Arguments**

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

### PsdExpr.addTerms()

Add linear terms to PSD expression object.

**Synopsis**

```
void addTerms(VarArray vars, double[] coeffs)
```

**Arguments**

`vars`: variables of added terms.

`coeffs`: coefficients of added terms.

### PsdExpr.addTerms()

Add PSD terms to PSD expression object.

**Synopsis**

```
void addTerms(PsdVarArray vars, SymMatrixArray mats)
```

**Arguments**

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrices for added PSD terms.

### PsdExpr.addTerms()

Add PSD terms to PSD expression object.

**Synopsis**

```
void addTerms(PsdVar[] vars, SymMatrix[] mats)
```

**Arguments**

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrices for added PSD terms.

### PsdExpr.clone()

Deep copy PSD expression object.

**Synopsis**

```
PsdExpr clone()
```

**Return**

cloned PSD expression object.

### PsdExpr.evaluate()

Evaluate PSD expression after solving.

**Synopsis**

```
double evaluate()
```

**Return**

Value of PSD expression.

### PsdExpr.getCoeff()

Get coefficient from the i-th term in PSD expression.

**Synopsis**

```
SymMatExpr getCoeff(int i)
```

**Arguments**

`i`: index of the PSD term.

**Return**

coefficient expression of the i-th PSD term.

### PsdExpr.getConstant()

Get constant in PSD expression.

**Synopsis**

```
double getConstant()
```

**Return**

constant in PSD expression.

### PsdExpr.getLinExpr()

Get linear expression in PSD expression.

**Synopsis**

```
Expr getLinExpr()
```

**Return**

linear expression object.

### PsdExpr.getPsdVar()

Get the PSD variable from the i-th term in PSD expression.

**Synopsis**

```
PsdVar getPsdVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

the first variable of the i-th term in PSD expression object.

### PsdExpr.multiply()

Multiply itself by a constant.

**Synopsis**

```
PsdExpr multiply(double c)
```

**Arguments**

`c`: constant operand.

**Return**

PSD expression itself.

### PsdExpr.remove()

Remove i-th term from PSD expression object.

**Synopsis**

```
void remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### PsdExpr.remove()

Remove the term associated with variable from PSD expression.

**Synopsis**

```
void remove(Var var)
```

**Arguments**

`var`: a variable whose term should be removed.

### PsdExpr.remove()

Remove the term associated with PSD variable from PSD expression.

**Synopsis**

```
void remove(PsdVar var)
```

**Arguments**

`var`: a PSD variable whose term should be removed.

### PsdExpr.setCoeff()

Set coefficient matrix of the i-th term in PSD expression.

**Synopsis**

```
void setCoeff(int i, SymMatrix mat)
```

**Arguments**

`i`: index of the PSD term.

`mat`: coefficient matrix of the term.

### PsdExpr.setConstant()

Set constant for the PSD expression.

**Synopsis**

```
void setConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

### PsdExpr.size()

Get number of PSD terms in expression.

**Synopsis**

```
long size()
```

**Return**

number of PSD terms.

### 27.2.41 PsdConstraint

COPT PSD constraint object. PSD constraints are always associated with a particular model. User creates a PSD constraint object by adding a PSD constraint to model, rather than by constructor of PsdConstraint class.

#### PsdConstraint.get()

Get information value of the PSD constraint. Support related PSD information.

**Synopsis**

```
double get(String info)
```

**Arguments**

`info`: name of queried information.

**Return**

information value.

#### PsdConstraint.getIdx()

Get index of the PSD constraint.

**Synopsis**

```
int getIdx()
```

**Return**

the index of the PSD constraint.

#### PsdConstraint.getName()

Get name of the PSD constraint.

**Synopsis**

```
String getName()
```

**Return**

the name of the PSD constraint.

#### PsdConstraint.remove()

Remove this PSD constraint from model.

**Synopsis**

```
void remove()
```

#### PsdConstraint.set()

Set information value of the PSD constraint. Support related PSD information.

**Synopsis**

```
void set(String info, double value)
```

**Arguments**

`info`: name of queried information.

`value`: new information value.

#### PsdConstraint.setName()

Set name of a PSD constraint.

**Synopsis**

```
void setName(String name)
```

**Arguments**

`name`: the name to set.

### 27.2.42 PsdConstrArray

COPT PSD constraint array object. To store and access a set of *PsdConstraint* objects, Cardinal Optimizer provides PsdConstrArray class, which defines the following methods.

#### PsdConstrArray.PsdConstrArray()

Constructor of PsdConstrArray object.

**Synopsis**

```
PsdConstrArray()
```

#### PsdConstrArray.getPsdConstr()

Get idx-th PSD constraint object.

**Synopsis**

```
PsdConstraint getPsdConstr(int idx)
```

**Arguments**

`idx`: index of the PSD constraint.

**Return**

PSD constraint object with index idx.

#### PsdConstrArray.pushBack()

Add a PSD constraint object to PSD constraint array.

**Synopsis**

```
void pushBack(PsdConstraint constr)
```

**Arguments**

`constr`: a PSD constraint object.

#### PsdConstrArray.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD constraint objects.

#### PsdConstrArray.size()

Get the number of PSD constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of PSD constraint objects.

### 27.2.43 PsdConstrBuilder

COPT PSD constraint builder object. To help building a PSD constraint, given a PSD expression, constraint sense and right-hand side value, Cardinal Optimizer provides PsdConstrBuilder class, which defines the following methods.

#### PsdConstrBuilder.PsdConstrBuilder()

Constructor of PsdConstrBuilder object.

**Synopsis**

```
PsdConstrBuilder()
```

#### PsdConstrBuilder.getPsdExpr()

Get expression associated with PSD constraint.

**Synopsis**

```
PsdExpr getPsdExpr()
```

**Return**

PSD expression object.

### PsdConstrBuilder.getRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double getRange()
```

**Return**

length from lower bound to upper bound of the constraint.

### PsdConstrBuilder.getSense()

Get sense associated with PSD constraint.

**Synopsis**

```
char getSense()
```

**Return**

PSD constraint sense.

### PsdConstrBuilder.set()

Set detail of a PSD constraint to its builder object.

**Synopsis**

```
void set(
    PsdExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: expression object at one side of the PSD constraint.

`sense`: PSD constraint sense, other than COPT_RANGE.

`rhs`: constant at right side of the PSD constraint.

### PsdConstrBuilder.setRange()

Set a range constraint to its builder.

**Synopsis**

```
void setRange(PsdExpr expr, double range)
```

**Arguments**

`expr`: PSD expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must be greater than 0.

### 27.2.44 PsdConstrBuilderArray

COPT PSD constraint builder array object. To store and access a set of *PsdConstrBuilder* objects, Cardinal Optimizer provides PsdConstrBuilderArray class, which defines the following methods.

#### PsdConstrBuilderArray.PsdConstrBuilderArray()

Constructor of PsdConstrBuilderArray object.

**Synopsis**

```
PsdConstrBuilderArray()
```

#### PsdConstrBuilderArray.getBuilder()

Get idx-th PSD constraint builder object.

**Synopsis**

```
PsdConstrBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the PSD constraint builder.

**Return**

PSD constraint builder object with index idx.

#### PsdConstrBuilderArray.pushBack()

Add a PSD constraint builder to PSD constraint builder array.

**Synopsis**

```
void pushBack(PsdConstrBuilder builder)
```

**Arguments**

`builder`: a PSD constraint builder object.

#### PsdConstrBuilderArray.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: minimum capacity for PSD constraint builder object.

#### PsdConstrBuilderArray.size()

Get the number of PSD constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of PSD constraint builder objects.

### 27.2.45 LmiConstraint

COPT LMI constraint object. LMI constraints are always associated with a particular model. User creates a LMI constraint object by adding a LMI constraint to model, rather than by constructor of LmiConstraint class.

#### LmiConstraint.get()

Get information values of LMI constraint.

**Synopsis**

```
double[] get(String info)
```

**Arguments**

`info`: information name.

**Return**

array of information values.

#### LmiConstraint.getDim()

Get dimension of LMI constraint.

**Synopsis**

```
int getDim()
```

**Return**

dimension of LMI constraint.

#### LmiConstraint.getIdx()

Get index of LMI constraint.

**Synopsis**

```
int getIdx()
```

**Return**

index of LMI constraint.

#### LmiConstraint.getLen()

Get length of LMI constraint.

**Synopsis**

```
int getLen()
```

**Return**

length of LMI constraint.

#### LmiConstraint.getName()

Get name of LMI constraint.

**Synopsis**

```
String getName()
```

**Return**

name of LMI constraint.

#### LmiConstraint.remove()

Remove this LMI constraint from model.

**Synopsis**

```
void remove()
```

#### LmiConstraint.setRhs()

Set constant term of LMI constraint.

**Synopsis**

```
void setRhs(SymMatrix mat)
```

**Arguments**

`mat`: new symmetric matrix for constant term.

### 27.2.46 LmiConstrArray

COPT LMI constraint array object. To store and access a set of *LmiConstraint* objects, Cardinal Optimizer provides LmiConstrArray class, which defines the following methods.

#### LmiConstrArray.LmiConstrArray()

Constructor of LmiConstrArray.

**Synopsis**

```
LmiConstrArray()
```

#### LmiConstrArray.getLmiConstr()

Get idx-th LMI constraint object.

**Synopsis**

```
LmiConstraint getLmiConstr(int idx)
```

**Arguments**

`idx`: index of the LMI constraint.

**Return**

LMI constraint object with index idx.

#### LmiConstrArray.pushBack()

Add an LMI constraint to LMI constraint array.

**Synopsis**

```
void pushBack(LmiConstraint constr)
```

**Arguments**

`constr`: LMI constraint object.

#### LmiConstrArray.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: capacity number of LMI constraint objects.

#### LmiConstrArray.size()

Get the number of LMI constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of LMI constraint objects.

### 27.2.47 LmiExpr

COPT LMI expression object. A LMI expression consists of a list of variables, associated coefficient matrices of LMI term, and constant matrices. LMI expressions are used to build LMI constraints.

### LmiExpr.LmiExpr()

Default constructor of a LMI expression.

**Synopsis**

```
LmiExpr()
```

### LmiExpr.LmiExpr()

Constructor of LMI expression with constant term.

**Synopsis**

```
LmiExpr(SymMatrix mat)
```

**Arguments**

`mat`: symmetric matrix object.

### LmiExpr.LmiExpr()

Constructor of LMI expression with constant term.

**Synopsis**

```
LmiExpr(SymMatExpr expr)
```

**Arguments**

`expr`: matrix expression object.

### LmiExpr.LmiExpr()

Constructor of LMI expression with one term.

**Synopsis**

```
LmiExpr(Var var, SymMatrix mat)
```

**Arguments**

`var`: variable of the added term.

`mat`: coefficient matrix of the added term.

### LmiExpr.LmiExpr()

Constructor of LMI expression with one term.

**Synopsis**

```
LmiExpr(Var var, SymMatExpr expr)
```

**Arguments**

`var`: variable of the added term.

`expr`: coefficient expression of symmetric matrices of new LMI term.

### LmiExpr.addConstant()

Add to constant term of the LMI expression.

**Synopsis**

```
void addConstant(SymMatExpr expr)
```

**Arguments**

`expr`: matrix expression added to the constant term.

### LmiExpr.addLmiExpr()

Add an LMI expression to self.

**Synopsis**

```
void addLmiExpr(LmiExpr expr)
```

**Arguments**

`expr`: LMI expression to be added.

### LmiExpr.addLmiExpr()

Add an LMI expression to self.

**Synopsis**

```
void addLmiExpr(LmiExpr expr, double mult)
```

**Arguments**

`expr`: LMI expression to be added.

`mult`: multiplier constant.

### LmiExpr.addTerm()

Add an LMI term to LMI expression object.

**Synopsis**

```
void addTerm(Var var, SymMatrix mat)
```

**Arguments**

`var`: LMI variable of new LMI term.

`mat`: coefficient matrix of new LMI term.

### LmiExpr.addTerm()

Add an LMI term to LMI expression object.

**Synopsis**

```
void addTerm(Var var, SymMatExpr expr)
```

**Arguments**

`var`: variable of new LMI term.

`expr`: coefficient expression of symmetric matrices of new LMI term.

### LmiExpr.addTerms()

Add LMI terms to LMI expression object.

**Synopsis**

```
void addTerms(VarArray vars, SymMatrixArray mats)
```

**Arguments**

`vars`: variables for added LMI terms.

`mats`: coefficient matrices for added LMI terms.

### LmiExpr.addTerms()

Add LMI terms to LMI expression object.

**Synopsis**

```
void addTerms(Var[] vars, SymMatrix[] mats)
```

**Arguments**

`vars`: variables for added LMI terms.

`mats`: coefficient matrices for added LMI terms.

### LmiExpr.clone()

Deep copy LMI expression.

**Synopsis**

```
LmiExpr clone()
```

**Return**

cloned LMI expression object.

### LmiExpr.getCoeff()

Get coefficient from the i-th term in LMI expression.

**Synopsis**

```
SymMatExpr getCoeff(int i)
```

**Arguments**

`i`: index of the LMI term.

**Return**

coefficient expression of the i-th LMI term.

### LmiExpr.getConstant()

Get constant term in LMI expression.

**Synopsis**

```
SymMatExpr getConstant()
```

**Return**

symmetric matrix expression object.

### LmiExpr.getVar()

Get variable from the i-th term in LMI expression.

**Synopsis**

```
Var getVar(int i)
```

**Arguments**

`i`: index of the term.

**Return**

variable of the i-th term in LMI expression object.

### LmiExpr.multiply()

Multiply itself by a constant.

**Synopsis**

```
LmiExpr multiply(double c)
```

**Arguments**

`c`: constant operand.

**Return**

LMI expression itself.

### LmiExpr.remove()

Remove i-th term from LMI expression object.

**Synopsis**

```
void remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

#### LmiExpr.remove()

Remove the term associated with variable from LMI expression.

**Synopsis**

```
void remove(Var var)
```

**Arguments**

`var`: variable whose term should be removed.

#### LmiExpr.setCoeff()

Set coefficient matrix of the i-th term in LMI expression.

**Synopsis**

```
void setCoeff(int i, SymMatrix mat)
```

**Arguments**

`i`: index of the LMI term.

`mat`: coefficient matrix of the term.

#### LmiExpr.setConstant()

Set constant term of the LMI expression.

**Synopsis**

```
void setConstant(SymMatrix mat)
```

**Arguments**

`mat`: symmetric matrix of the constant term.

#### LmiExpr.size()

Get number of LMI terms in expression.

**Synopsis**

```
long size()
```

**Return**

number of LMI terms.

### 27.2.48 SymMatrix

COPT symmetric matrix object. Symmetric matrices are always associated with a particular model. User creates a symmetric matrix object by adding a symmetric matrix to model, rather than by constructor of SymMatrix class.

Symmetric matrices are used as coefficient matrices of PSD terms in PSD expressions, PSD constraints or PSD objectives.

#### SymMatrix.getDim()

Get the dimension of a symmetric matrix.

**Synopsis**

```
int getDim()
```

**Return**

Dimension of a symmetric matrix.

#### SymMatrix.getIdx()

Get the index of a symmetric matrix.

**Synopsis**

```
int getIdx()
```

**Return**

Index of a symmetric matrix.

### 27.2.49 SymMatrixArray

COPT symmetric matrix object. To store and access a set of *SymMatrix* objects, Cardinal Optimizer provides SymMatrixArray class, which defines the following methods.

#### SymMatrixArray.SymMatrixArray()

Constructor of SymMatrixAarray.

**Synopsis**

```
SymMatrixArray()
```

#### SymMatrixArray.getMatrix()

Get i-th SymMatrix object.

**Synopsis**

```
SymMatrix getMatrix(int idx)
```

**Arguments**

`idx`: index of the SymMatrix object.

**Return**

SymMatrix object with index idx.

#### SymMatrixArray.pushBack()

Add a SymMatrix object to SymMatrix array.

**Synopsis**

```
void pushBack(SymMatrix mat)
```

**Arguments**

`mat`: a SymMatrix object.

#### SymMatrixArray.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: minimum capacity for symmetric matrix object.

#### SymMatrixArray.size()

Get the number of SymMatrix objects.

**Synopsis**

```
int size()
```

**Return**

number of SymMatrix objects.

### 27.2.50 SymMatExpr

COPT symmetric matrix expression object. A symmetric matrix expression is a linear combination of symmetric matrices, which is still a symmetric matrix. However, by doing so, we are able to delay computing the final matrix until setting PSD constraints or PSD objective.

#### SymMatExpr.SymMatExpr()

Constructor of a symmetric matrix expression.

**Synopsis**

```
SymMatExpr()
```

#### SymMatExpr.SymMatExpr()

Constructor of a symmetric matrix expression with one term.

**Synopsis**

```
SymMatExpr(SymMatrix mat, double coeff)
```

**Arguments**

`mat`: symmetric matrix of the added term.

`coeff`: coefficient for the added term.

### SymMatExpr.addSymMatExpr()

Add a symmetric matrix expression to self.

**Synopsis**

```
void addSymMatExpr(SymMatExpr expr, double mult)
```

**Arguments**

`expr`: symmetric matrix expression to be added.

`mult`: constant multiplier.

### SymMatExpr.addTerm()

Add a term to symmetric matrix expression object.

**Synopsis**

```
Boolean addTerm(SymMatrix mat, double coeff)
```

**Arguments**

`mat`: symmetric matrix of the new term.

`coeff`: coefficient of the new term.

**Return**

True if the term is added successfully.

### SymMatExpr.addTerms()

Add multiple terms to expression object.

**Synopsis**

```
int addTerms(SymMatrix[] mats, double coeff)
```

**Arguments**

`mats`: symmetric matrix array object for added terms.

`coeff`: common coefficient for added terms.

**Return**

Number of added terms. If negative, fail to add one of terms.

### SymMatExpr.addTerms()

Add multiple terms to expression object.

**Synopsis**

```
int addTerms(SymMatrixArray mats, double[] coeffs)
```

**Arguments**

`mats`: symmetric matrix array object for added terms.

`coeffs`: coefficient array for added terms.

**Return**

Number of added terms. If negative, fail to add one of terms.

### SymMatExpr.addTerms()

Add multiple terms to expression object.

**Synopsis**

```
int addTerms(SymMatrix[] mats, double[] coeffs)
```

**Arguments**

`mats`: symmetric matrix array object for added terms.

`coeffs`: coefficient array for added terms.

**Return**

Number of added terms. If negative, fail to add one of terms.

### SymMatExpr.clone()

Deep copy symmetric matrix expression object.

**Synopsis**

```
SymMatExpr clone()
```

**Return**

cloned expression object.

### SymMatExpr.getCoeff()

Get coefficient of the i-th term in expression object.

**Synopsis**

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

**Arguments**

`i`: index of the term.

**Return**

coefficient of the i-th term.

### SymMatExpr.getDim()

Get dimension of symmetric matrix in expression.

**Synopsis**

```
int getDim()
```

**Return**

dimension of symmetric matrix.

### SymMatExpr.getSymMat()

Get symmetric matrix of the i-th term in expression object.

**Synopsis**

```
SymMatrix getSymMat(int i)
```

**Arguments**

`i`: index of the term.

**Return**

the symmetric matrix of the i-th term.

### SymMatExpr.multiply()

Multiply itself by a constant.

**Synopsis**

```
SymMatExpr multiply(double c)
```

**Arguments**

`c`: constant operand.

**Return**

symmetric matrix expression itself.

### SymMatExpr.remove()

Remove i-th term from expression object.

**Synopsis**

```
void remove(int idx)
```

**Arguments**

`idx`: index of the term to be removed.

### SymMatExpr.remove()

Remove the term associated with the symmetric matrix.

**Synopsis**

```
void remove(SymMatrix mat)
```

**Arguments**

`mat`: a symmetric matrix whose term should be removed.

#### SymMatExpr.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: minimum capacity for expression object.

#### SymMatExpr.setCoeff()

Set coefficient for the i-th term in expression object.

**Synopsis**

```
void setCoeff(int i, double val)
```

**Arguments**

`i`: index of the term.

`val`: coefficient of the term.

#### SymMatExpr.size()

Get number of terms in expression.

**Synopsis**

```
long size()
```

**Return**

number of terms.

### 27.2.51 NlExpr Class

COPT nonlinear expression object. The `NlExpr` class represents nonlinear expressions in COPT. The nonlinear expressions are used to build nonlinear constraints. The following methods are provided:

#### NlExpr.NlExpr()

Default constructor of a nonlinear expression.

**Synopsis**

```
NlExpr()
```

#### NlExpr.NlExpr()

Constructor of a nonlinear expression with a constant.

**Synopsis**

```
NlExpr(double constant)
```

**Arguments**

`constant`: constant value in nonlinear expression object.

### NlExpr.NlExpr()

Constructor of a nonlinear expression with a variable.

**Synopsis**

```
NlExpr(Var var)
```

**Arguments**

`var`: the added variable.

### NlExpr.NlExpr()

Constructor of a nonlinear expression with one linear term.

**Synopsis**

```
NlExpr(Var var, double coeff)
```

**Arguments**

`var`: variable for the added term.

`coeff`: coefficient for the added term.

### NlExpr.NlExpr()

Constructor of a nonlinear expression with a linear expression.

**Synopsis**

```
NlExpr(Expr expr)
```

**Arguments**

`expr`: the added linear expression.

### NlExpr.NlExpr()

Constructor of a nonlinear expression with a quadratic expression.

**Synopsis**

```
NlExpr(QuadExpr expr)
```

**Arguments**

`expr`: the added quadratic expression.

### NlExpr.add()

Add itself by an expression.

**Synopsis**

```
NlExpr add(NlExpr expr, double mult)
```

**Arguments**

`expr`: expression operand, including NlExpr, QuadExpr, Expr, Var.

`mult`: constant multiplier.

**Return**

nonlinear expression itself.

### NlExpr.add()

Add itself by an expression.

**Synopsis**

```
NlExpr add(NlExpr expr)
```

**Arguments**

`expr`: expression operand, including NlExpr, QuadExpr, Expr, Var and constant.

**Return**

nonlinear expression itself.

### NlExpr.addConstant()

Add constant to the nonlinear expression.

**Synopsis**

```
void addConstant(double constant)
```

**Arguments**

`constant`: value to be added.

### NlExpr.addLinExpr()

Add a linear expression to self.

**Synopsis**

```
void addLinExpr(Expr expr, double mult)
```

**Arguments**

`expr`: linear expression to be added.

`mult`: constant multiplier.

### NlExpr.addNlExpr()

Add a nonlinear expression to self.

**Synopsis**

```
void addNlExpr(NlExpr expr, double mult)
```

**Arguments**

`expr`: nonlinear expression to be added.

`mult`: constant multiplier.

### NlExpr.addQuadExpr()

Add a quadratic expression to self.

**Synopsis**

```
void addQuadExpr(QuadExpr expr, double mult)
```

**Arguments**

`expr`: quadratic expression to be added.

`mult`: constant multiplier.

### NlExpr.addTerm()

Add a linear term to nonlinear expression object.

**Synopsis**

```
void addTerm(Var var, double coeff)
```

**Arguments**

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

### NlExpr.addTerms()

Add linear terms to nonlinear expression object.

**Synopsis**

```
void addTerms(Var[] vars, double[] coeffs)
```

**Arguments**

`vars`: variable array for added linear terms.

`coeffs`: coefficient array for added linear terms.

### NlExpr.addTerms()

Add linear terms to nonlinear expression object.

**Synopsis**

```
void addTerms(VarArray vars, double[] coeffs)
```

**Arguments**

`vars`: variables for added linear terms.

`coeffs`: coefficient array for added linear terms.

### NlExpr.clear()

Clear nonlinear expression object.

**Synopsis**

```
void clear()
```

### NlExpr.clone()

Deep copy nonlinear expression object.

**Synopsis**

```
NlExpr clone()
```

**Return**

cloned nonlinear expression object.

### NlExpr.divide()

Divide itself by an expression.

**Synopsis**

```
NlExpr divide(NlExpr expr)
```

**Arguments**

`expr`: expression operand, including NlExpr, QuadExpr, Expr, Var and constant.

**Return**

nonlinear expression itself.

### NlExpr.evaluate()

Evaluate nonlinear expression after solving.

**Synopsis**

```
double evaluate()
```

**Return**

value of nonlinear expression.

### NlExpr.getConstant()

Get constant in nonlinear expression.

**Synopsis**

```
double getConstant()
```

**Return**

constant in nonlinear expression.

### NlExpr.getLinExpr()

Get linear expression of nonlinear expression.

**Synopsis**

```
Expr getLinExpr()
```

**Return**

linear expression object.

### NlExpr.multiply()

Multiply itself by an expression.

**Synopsis**

```
NlExpr multiply(NlExpr expr)
```

**Arguments**

`expr`: expression operand, including NlExpr, QuadExpr, Expr, Var and constant.

**Return**

nonlinear expression itself.

### NlExpr.negate()

Negate itself.

**Synopsis**

```
NlExpr negate()
```

**Return**

nonlinear expression itself.

### NlExpr.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: capacity of nonlinear constraint objects.

### NlExpr.setConstant()

Set constant for the nonlinear expression.

**Synopsis**

```
void setConstant(double constant)
```

**Arguments**

`constant`: the value of the constant.

#### NlExpr.size()

Get size of tokens in nonlinear expression.

**Synopsis**

```
long size()
```

**Return**

size of none-linear tokens.

### 27.2.52 NlConstraint Class

COPT nonlinear constraint object. The `NlConstraint` object is always associated with a particular model. User creates a `NlConstraint` object by adding a nonlinear constraint to model, rather than by constructor of `NlConstraint` class.

#### NlConstraint.get()

Get information value of the nonlinear constraint. Support information of "LB", "UB" and "Slack".

**Synopsis**

```
double get(String info)
```

**Arguments**

`info`: name of the information being queried.

**Return**

value of information.

#### NlConstraint.getIdx()

Get index of nonlinear constraint.

**Synopsis**

```
int getIdx()
```

**Return**

the index of nonlinear constraint.

#### NlConstraint.getName()

Get name of nonlinear constraint.

**Synopsis**

```
String getName()
```

**Return**

the name of nonlinear constraint.

#### NlConstraint.remove()

Remove this nonlinear constraint from model.

**Synopsis**

```
void remove()
```

#### NlConstraint.set()

Set information value of nonlinear constraint. Support information of "LB" and "UB".

**Synopsis**

```
void set(String info, double val)
```

**Arguments**

`info`: name of the information.

`val`: new information value.

#### NlConstraint.setName()

Set name for nonlinear constraint.

**Synopsis**

```
void setName(String name)
```

**Arguments**

`name`: the name to set.

### 27.2.53 NlConstrArray Class

COPT nonlinear constraint array object. To store and access a set of *NlConstraint Class* objects, Cardinal Optimizer provides NlConstrArray class, which defines the following methods.

#### NlConstrArray.NlConstrArray()

Constructor of NlConstrArray object.

**Synopsis**

```
NlConstrArray()
```

#### NlConstrArray.getNlConstr()

Get idx-th nonlinear constraint object.

**Synopsis**

```
NlConstraint getNlConstr(int idx)
```

**Arguments**

`idx`: index of the nonlinear constraint.

**Return**

nonlinear constraint object with index value.

#### NlConstrArray.pushBack()

Add a nonlinear constraint to nonlinear constraint array.

**Synopsis**

```
void pushBack(NlConstraint constr)
```

**Arguments**

`constr`: nonlinear constraint object.

#### NlConstrArray.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: capacity of nonlinear constraint objects.

#### NlConstrArray.size()

Get the number of nonlinear constraint objects.

**Synopsis**

```
int size()
```

**Return**

number of nonlinear constraint objects.

### 27.2.54 NlConstrBuilder Class

COPT nonlinear constraint builder object. To help building a nonlinear constraint, given a nonlinear expression, constraint sense and right-hand side value, Cardinal Optimizer provides NlConstrBuilder class, which defines the following methods.

#### NlConstrBuilder.NlConstrBuilder()

Constructor of NlConstrBuilder object.

**Synopsis**

```
NlConstrBuilder()
```

#### NlConstrBuilder.getNlExpr()

Get nonlinear expression associated with constraint.

**Synopsis**

```
NlExpr getNlExpr()
```

**Return**

nonlinear expression object.

### NlConstrBuilder.getRange()

Get range from lower bound to upper bound of range constraint.

**Synopsis**

```
double getRange()
```

**Return**

length from lower bound to upper bound of nonlinear constraint.

### NlConstrBuilder.getSense()

Get sense associated with nonlinear constraint.

**Synopsis**

```
char getSense()
```

**Return**

nonlinear constraint sense.

### NlConstrBuilder.set()

Set detail of a nonlinear constraint to its builder object.

**Synopsis**

```
void set(
    NlExpr expr,
    char sense,
    double rhs)
```

**Arguments**

`expr`: nonlinear expression object at one side of nonlinear constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of nonlinear constraint.

### NlConstrBuilder.setRange()

Set a range constraint to nonlinear constraint builder.

**Synopsis**

```
void setRange(NlExpr expr, double range)
```

**Arguments**

`expr`: nonlinear expression object, whose constant is negative upper bound.

`range`: length from lower to upper bound of nonlinear constraint. Must be greater than 0.

### 27.2.55 NlConstrBuilderArray Class

COPT nonlinear constraint builder array object. To store and access a set of *NlConstrBuilder Class* objects, Cardinal Optimizer provides `NlConstrBuilderArray` class, which defines the following methods.

#### NlConstrBuilderArray.NlConstrBuilderArray()

Constructor of NlConstrBuilderArray object.

**Synopsis**

```
NlConstrBuilderArray()
```

#### NlConstrBuilderArray.getBuilder()

Get idx-th nonlinear constraint builder object.

**Synopsis**

```
NlConstrBuilder getBuilder(int idx)
```

**Arguments**

`idx`: index of the nonlinear constraint builder.

**Return**

nonlinear constraint builder object with index idx.

#### NlConstrBuilderArray.pushBack()

Add a nonlinear constraint builder object to nonlinear constraint builder array.

**Synopsis**

```
void pushBack(NlConstrBuilder builder)
```

**Arguments**

`builder`: a nonlinear constraint builder object.

#### NlConstrBuilderArray.reserve()

Reserve capacity to contain at least n items.

**Synopsis**

```
void reserve(int n)
```

**Arguments**

`n`: capacity of nonlinear constraint objects.

#### NlConstrBuilderArray.size()

Get the number of nonlinear constraint builder objects.

**Synopsis**

```
int size()
```

**Return**

number of nonlinear constraint builder objects.

### 27.2.56 NL Namespace

Common nonlinear functions in the `NL` namespace are provided for constructing nonlinear expressions. The following methods are provided:

#### NL.abs()

Calculate absolute value of a nonlinear expression.

**Synopsis**

```
static NlExpr abs(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

#### NL.acos()

Calculate arccosine of a nonlinear expression.

**Synopsis**

```
static NlExpr acos(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

#### NL.acosh()

Calculate inverse hyperbolic cosine of a nonlinear expression.

**Synopsis**

```
static NlExpr acosh(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.asin()

Calculate arcsine of a nonlinear expression.

**Synopsis**

```
static NlExpr asin(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.asinh()

Calculate inverse hyperbolic sine of a nonlinear expression.

**Synopsis**

```
static NlExpr asinh(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.atan()

Calculate arctangent of a nonlinear expression.

**Synopsis**

```
static NlExpr atan(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.atan2()

Calculate two-argument arctangent of a nonlinear expression.

**Synopsis**

```
static NlExpr atan2(NlExpr y, NlExpr x)
```

**Arguments**

`y`: y coordinate as a nonlinear expression.

`x`: x coordinate as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.atanh()

Calculate inverse hyperbolic tangent of a nonlinear expression.

**Synopsis**

```
static NlExpr atanh(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.ceil()

Calculate ceiling value of a nonlinear expression.

**Synopsis**

```
static NlExpr ceil(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.cos()

Calculate cosine of a nonlinear expression.

**Synopsis**

```
static NlExpr cos(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.cosh()

Calculate hyperbolic cosine of a nonlinear expression.

**Synopsis**

```
static NlExpr cosh(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.exp()

Calculate exponential function of a nonlinear expression.

**Synopsis**

```
static NlExpr exp(NlExpr expo)
```

**Arguments**

`expo`: exponent as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.floor()

Calculate floor value of a nonlinear expression.

**Synopsis**

```
static NlExpr floor(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.log10()

Calculate logarithmic function of a nonlinear expression with base 10.

**Synopsis**

```
static NlExpr log10(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.log()

Calculate nature logarithmic function of a nonlinear expression.

**Synopsis**

```
static NlExpr log(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.neg()

Calculate negative value of a nonlinear expression.

**Synopsis**

```
static NlExpr neg(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.pow()

Calculate power function of a nonlinear expression.

**Synopsis**

```
static NlExpr pow(NlExpr base, NlExpr expo)
```

**Arguments**

`base`: base as a nonlinear expression.

`expo`: exponent as a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.sin()

Calculate sine of a nonlinear expression.

**Synopsis**

```
static NlExpr sin(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.sinh()

Calculate hyperbolic sine of a nonlinear expression.

**Synopsis**

```
static NlExpr sinh(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.sqrt()

Calculate square root of a nonlinear expression.

**Synopsis**

```
static NlExpr sqrt(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.sum()

Sum of nonlinear expressions.

**Synopsis**

```
static NlExpr sum(
    NlExpr op1,
    NlExpr op2,
    NlExpr op3,
    NlExpr op4)
```

**Arguments**

`op1`: first nonlinear expression.

`op2`: second nonlinear expression.

`op3`: third nonlinear expression.

`op4`: fourth nonlinear expression.

**Return**

result as a nonlinear expression.

### NL.sum()

Sum of nonlinear expressions.

**Synopsis**

```
static NlExpr sum(
    NlExpr op1,
    NlExpr op2,
    NlExpr op3)
```

**Arguments**

`op1`: first nonlinear expression.

`op2`: second nonlinear expression.

`op3`: third nonlinear expression.

**Return**

result as a nonlinear expression.

#### NL.sum()

Sum of nonlinear expressions.

**Synopsis**

```
static NlExpr sum(NlExpr[] exprs)
```

**Arguments**

`exprs`: array of nonlinear expressions.

**Return**

result as a nonlinear expression.

#### NL.tan()

Calculate tangent of a nonlinear expression.

**Synopsis**

```
static NlExpr tan(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

#### NL.tanh()

Calculate hyperbolic tangent of a nonlinear expression.

**Synopsis**

```
static NlExpr tanh(NlExpr expr)
```

**Arguments**

`expr`: a nonlinear expression.

**Return**

result as a nonlinear expression.

### 27.2.57 CallbackBase

COPT Callback abstract base object. Users must implement its virtual method `virtual void CallbackBase::callback()` to instantiate an instance, which pass to `Model::SetCallback(CallbackBase cb, int cbctx)` as the first parameter. Subclass of CallbackBase inherits the following member methods:

### CallbackBase.CallbackBase()

Constructor of CallbackBase, implementing ICallback interface.

**Synopsis**

```
CallbackBase()
```

### CallbackBase.addLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void addLazyConstr(
    Expr lhs,
    char sense,
    double rhs)
```

**Arguments**

`lhs`: expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

### CallbackBase.addLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void addLazyConstr(
    Expr lhs,
    char sense,
    Expr rhs)
```

**Arguments**

`lhs`: left hand side expression for lazy constraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy constraint.

### CallbackBase.addLazyConstr()

Add a lazy constraint to model.

**Synopsis**

```
void addLazyConstr(ConstrBuilder builder)
```

**Arguments**

`builder`: builder for lazy constraint.

### CallbackBase.addLazyConstrs()

Add lazy constraints to model.

**Synopsis**

```
void addLazyConstrs(ConstrBuilderArray builders)
```

**Arguments**

`builders`: array of builders for lazy constraints.

### CallbackBase.addUserCut()

Add a user cut to model.

**Synopsis**

```
void addUserCut(
    Expr lhs,
    char sense,
    double rhs)
```

**Arguments**

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

### CallbackBase.addUserCut()

Add a user cut to model.

**Synopsis**

```
void addUserCut(
    Expr lhs,
    char sense,
    Expr rhs)
```

**Arguments**

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

### CallbackBase.addUserCut()

Add a user cut to model.

**Synopsis**

```
void addUserCut(ConstrBuilder builder)
```

**Arguments**

`builder`: builder for user cut.

### CallbackBase.addUserCuts()

Add user cuts to model.

**Synopsis**

```
void addUserCuts(ConstrBuilderArray builders)
```

**Arguments**

`builders`: array of builders for user cuts.

### CallbackBase.callback()

Pure virtual function defined in ICallback interface. User must implement it.

**Synopsis**

```
void callback()
```

### CallbackBase.getDblInfo()

Get double value of given information name in callback.

**Synopsis**

```
double getDblInfo(String cbinfo)
```

**Arguments**

`cbinfo`: name of callback info.

**Return**

value of desired information.

### CallbackBase.getIncumbent()

Get best feasible solution of given variable in callback.

**Synopsis**

```
double getIncumbent(Var var)
```

**Arguments**

`var`: given variable.

**Return**

best feasible solution of given variable.

### CallbackBase.getIncumbent()

Get best feasible solution of variables in callback.

**Synopsis**

```
double[] getIncumbent(VarArray vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

best feasible solution of desired variables.

### CallbackBase.getIncumbent()

Get best feasible solution of variables in callback.

**Synopsis**

```
double[] getIncumbent(Var[] vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

best feasible solution of desired variables.

### CallbackBase.getIncumbent()

Get best feasible solution of all variables in callback.

**Synopsis**

```
double[] getIncumbent()
```

**Return**

best feasible solution of all variables.

### CallbackBase.getIntInfo()

Get integer value of given information name in callback.

**Synopsis**

```
int getIntInfo(String cbinfo)
```

**Arguments**

`cbinfo`: name of callback info.

**Return**

value of desired information.

### CallbackBase.getRelaxSol()

Get LP-relaxation solution of given variable in callback.

**Synopsis**

```
double getRelaxSol(Var var)
```

**Arguments**

`var`: given variable.

**Return**

LP-relaxation solution of given variable.

### CallbackBase.getRelaxSol()

Get LP-relaxation solution of variables in callback.

**Synopsis**

```
double[] getRelaxSol(VarArray vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

LP-relaxation solution of variables.

### CallbackBase.getRelaxSol()

Get LP-relaxation solution of variables in callback.

**Synopsis**

```
double[] getRelaxSol(Var[] vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

LP-relaxation solution of variables.

### CallbackBase.getRelaxSol()

Get LP-relaxation solution of all variables in callback.

**Synopsis**

```
double[] getRelaxSol()
```

**Return**

LP-relaxation solution of all variables.

### CallbackBase.getSolution()

Get solution of given variable in callback.

**Synopsis**

```
double getSolution(Var var)
```

**Arguments**

`var`: given variable.

**Return**

solution of given variable.

### CallbackBase.getSolution()

Get solution of variables in callback.

**Synopsis**

```
double[] getSolution(VarArray vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

solution of variables.

### CallbackBase.getSolution()

Get solution of variables in callback.

**Synopsis**

```
double[] getSolution(Var[] vars)
```

**Arguments**

`vars`: an array of variables.

**Return**

solution of variables.

### CallbackBase.getSolution()

Get solution of all variables in callback.

**Synopsis**

```
double[] getSolution()
```

**Return**

solution of all variables.

### CallbackBase.interrupt()

Interrupt solving problems in callback

**Synopsis**

```
void interrupt()
```

### CallbackBase.loadSolution()

Load customized solution to model.

**Synopsis**

```
double loadSolution()
```

**Return**

objective value of given solution.

#### CallbackBase.setSolution()

Set solution of a given variable in callback.

**Synopsis**

```
void setSolution(Var var, double val)
```

**Arguments**

`var`: a variable object.

`val`: double value.

#### CallbackBase.setSolution()

Set solution of variables in callback.

**Synopsis**

```
void setSolution(VarArray vars, double[] vals)
```

**Arguments**

`vars`: an array of variable objects.

`vals`: an array of double values.

#### CallbackBase.setSolution()

Set solution of variables in callback.

**Synopsis**

```
void setSolution(Var[] vars, double[] vals)
```

**Arguments**

`vars`: an array of variable objects.

`vals`: an array of double values.

#### CallbackBase.where()

Get context in callback.

**Synopsis**

```
int where()
```

**Return**

integer value of context.

### 27.2.58 ProbBuffer

Buffer object for COPT problem. ProbBuffer object holds the (MPS) problem in string format.

#### ProbBuffer.ProbBuffer()

Constructor of ProbBuffer object.

**Synopsis**

```
ProbBuffer(int sz)
```

**Arguments**

`sz`: initial size of the problem buffer.

#### ProbBuffer.getData()

Get string of problem in problem buffer.

**Synopsis**

```
String getData()
```

**Return**

string of problem in problem buffer.

#### ProbBuffer.resize()

Resize buffer to given size, and zero-ended.

**Synopsis**

```
void resize(int sz)
```

**Arguments**

`sz`: new buffer size.

#### ProbBuffer.size()

Get the size of problem buffer.

**Synopsis**

```
int size()
```

**Return**

size of problem buffer.

### 27.2.59 CoptException

Copt exception object.

### CoptException.CoptException()

Constructor of coptexception.

**Synopsis**

```
CoptException(int code, String msg)
```

**Arguments**

`code`: error code for exception.

`msg`: error message for exception.

### CoptException.getCode()

Get the error code associated with the exception.

**Synopsis**

```
int getCode()
```

**Return**

the error code.