

Solver SDK 2020 Reference Guide

By Frontline Systems, Inc.

Copyright

Software copyright 1991-2020 by Frontline Systems, Inc.
User Guide copyright 2020 by Frontline Systems, Inc.
GRG/LSGRG Solver: Portions copyright 1989 by Optimal Methods, Inc. SOCP Barrier Solver: Portions copyright 2002 by Masakazu Muramatsu. LP/QP Solver: Portions copyright 2000-2010 by International Business Machines Corp. and others. Neither the Software nor this User Guide may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without the express written consent of Frontline Systems, Inc., except as permitted by the Software License agreement below.

Trademarks

Frontline Solvers®, XLMiner®, Analytic Solver®, Risk Solver®, Premium Solver®, Solver SDK®, and RASON® are trademarks of Frontline Systems, Inc. Windows and Excel are trademarks of Microsoft Corp. Gurobi is a trademark of Gurobi Optimization, Inc. Knitro is a trademark of Artelys. MOSEK is a trademark of MOSEK ApS. OptQuest is a trademark of OptTek Systems, Inc. Xpress^{MP} is a trademark of FICO, Inc.

Acknowledgements

Thanks to Dan Fylstra and the Frontline Systems development team for a 25-year cumulative effort to build the best possible optimization and simulation software for Microsoft Excel. Thanks to Frontline's customers who have built many thousands of successful applications, and have given us many suggestions for improvements.

Risk Solver Pro and Risk Solver Platform have benefited from reviews, critiques, and suggestions from several risk analysis experts:

- Sam Savage (Stanford Univ. and AnalyCorp Inc.) for Probability Management concepts including SIPs, SLURPs, DISTs, and Certified Distributions.
- Sam Sugiyama (EC Risk USA & Europe LLC) for evaluation of advanced distributions, correlations, and alternate parameters for continuous distributions.
- Savvakis C. Savvides for global bounds, censor bounds, base case values, the Normal Skewed distribution and new risk measures.

How to Order

Contact Frontline Systems, Inc., P.O. Box 4288, Incline Village, NV 89450.
Tel (775) 831-0300 Fax (775) 831-0314 Email info@solver.com Web http://www.solver.com

Contents at a Glance

Conventions and Acronyms Used in This Document i
Solver SDK Platform Object-Oriented API Reference 1
Solver SDK Platform Procedural API Reference 229
Solver SDK Platform Legacy API Reference 516
Optimization Problems Reference and Code Examples 599
Simulation Reference and Code Examples 669
Appendix 692
Index 696

Contents

Conventions and Acronyms Used in This Document i

Formatting Conventions ii

Code Formatting ii

Concepts Formatting ii

Design Conventions iii

Cross-References iii

Acronyms Used iv

Solver SDK Platform Object-Oriented API Reference 1

Classes and Enumerations Overview 2

Primary Classes 5

class CProblem 6

class CSolver 11

class CEngine 16

class CEvaluator 21

class CModel 25

class CVariable 32

class CFunction 37

class CEnvironment 42

Collections Classes 45

Secondary Classes 63

Matrix Classes 64

class CStatistics 83

class COptIIS 88

class CEngineStat 91

class CEngineParam 105

class CEngineParam 106

class CDistribution 110

class CSolverException 127

class CInputData 129

class CModelParam 135

Enumerations 159

enum APIerr 161

enum License_Loc 166

enum Solver_Type 168

enum Depend_Type ... 170
enum Problem_Type ... 172
enum Variable_Type ... 174
enum Function_Type ... 176
enum Eval_Type ... 178
enum Engine_Action ... 180
enum Array_Order ... 182
enum File_Format ... 184
enum Distribution_Type ... 186
enum DistFit_Type ... 190
enum Frequency_Type ... 193
enum Simulate_Status ... 195
enum Optimize_Status ... 197
enum Integer_Type ... 200
enum Cone_Type ... 202
enum Cons_Rel ... 204
enum IIS_Status ... 206
enum License_SelectorCode ... 208
enum DistParam_Type ... 210
enum Dist_VarFcn ... 213
enum DistFit_SortType ... 215
enum Check_Type ... 217
enum Convex_Type ... 219
enum Stochastic_Type ... 221
enum Solve_Type ... 223
enum Simulate_Type ... 225
enum Chance_Type ... 226
enum Parameter_Type ... 228

Solver SDK Platform Procedural API Reference ... 229

Method and Property Functions Overview ... 230
Problem Method Functions ... 241
SolverProblem ... 242
SolverProbInit ... 243
SolverProbFree ... 244
SolverProbLoad ... 245
SolverProbSave ... 246
SolverProbTrialPeriod ... 247
Problem Property Functions ... 248

SolverProbTypeGet ... 249
SolverProbTypeSet ... 250
SolverProbNameGet ... 251
SolverProbNameSet ... 252
SolverProbEngineGet ... 253
SolverProbEngineSet ... 254
Solver Method Functions ... 255
SolverOptimize ... 257
SolverOptIISFind ... 258
SolverReport ... 259
SolverAnalyze ... 260
SolverSolveRecourse ... 261
SolverSimOpt ... 262
Solver Property Functions ... 263
SolverNumSimulationsGet ... 264
SolverNumSimulationsSet ... 265
SolverNumTrialsGet ... 266
SolverNumTrialsSet ... 267
SolverSimulateStatus ... 268
SolverOptimizeStatus ... 269
SolverOptIIS ... 270
SolverNumSolutionsGet ... 271
SolverNumSolutionsSet ... 272
SolverSolutionsIndexGet ... 273
SolverSolutionsIndexSet ... 274
SolverFunctionCurrent ... 275
SolverFunctionType ... 276
SolverFunctionIndex ... 277
SolverOptIndexGet ... 278
SolverOptIndexSet ... 279
SolverNumOptimizationsGet ... 280
SolverNumOptimizationsSet ... 281
Engine Method Functions ... 282
SolverEngineAdd ... 283
SolverEngineRemove ... 284
SolverEngineDiscover ... 285
SolverEngParamLoad ... 286
SolverEngParamSave ... 287
SolverEngParamReset ... 288

Engine Property Functions ... 289
SolverEngCount ... 290
SolverEngName ... 291
SolverEngType ... 292
SolverEngParamCount ... 293
SolverEngParamName ... 294
SolverEngParamGet ... 295
SolverEngParamSet ... 296
SolverEngParamInfo ... 297
SolverEngLimit ... 298
SolverEngStat ... 299
Evaluator Method Functions ... 300
SolverEvaluatorAdd ... 301
SolverEvaluatorRemove ... 302
Model Method Functions ... 303
SolverModArrayOrder ... 304
SolverModDependTest ... 305
Model Property Functions ... 306
SolverModNumVariables ... 307
SolverModNumFunctions ... 308
SolverModNumAllGrad ... 309
SolverModNumFcnHess ... 310
SolverModNumDistArguments ... 311
SolverModNumCorrelations ... 312
SolverModAllLinearGet ... 313
SolverModAllLinearSet ... 314
SolverModAllSpLinearGet ... 315
SolverModAllSpLinearSet ... 316
SolverModAllGradDependGet ... 317
SolverModAllGradDependSet ... 318
SolverModAllGradientGet ... 319
SolverModAllGradientSet ... 320
SolverModAllSpGradientGet ... 321
SolverModAllSpGradientSet ... 322
SolverModFcnHessianGet ... 323
SolverModFcnHessianSet ... 324
SolverModFcnSpHessianGet ... 325
SolverModFcnSpHessianSet ... 326
SolverModFcnQuadraticGet ... 327

SolverModFcnQuadraticSet ... 328
SolverModFcnSpQuadraticGet ... 329
SolverModFcnSpQuadraticSet ... 330
SolverModFcnHessDependGet ... 331
SolverModFcnHessDependSet ... 332
SolverModDistTypeGet ... 333
SolverModDistTypeSet ... 334
SolverModDistNameGet ... 335
SolverModParamCount ... 336
SolverModParamName ... 337
SolverModParamGet ... 338
SolverModParamSet ... 339
SolverModParamInfo ... 340
SolverModDistCorrelationsGet ... 341
SolverModDistCorrelationsSet ... 342
SolverModVarIndexName ... 343
SolverModFcnIndexName ... 344
SolverModNameVarIndex ... 345
SolverModNameFcnIndex ... 346
Variable Method Functions ... 347
SolverVariableAdd ... 348
SolverVariableRemove ... 349
SolverVarGetCorrelation ... 350
SolverVarGetFrequency ... 351
SolverVarNonNegative ... 352
Variable Property Functions ... 353
SolverVarCount ... 354
SolverVarNameGet ... 355
SolverVarNameSet ... 356
SolverVarTypeGet ... 357
SolverVarTypeSet ... 358
SolverVarValue ... 359
SolverVarSizeGet ... 360
SolverVarSizeSet ... 361
SolverVarPosition ... 362
SolverVarStatistics ... 363
SolverVarPercentiles ... 364
SolverVarAllTrialsGet ... 365
SolverVarAllTrialsSet ... 366

SolverVarLowerBoundGet 367
SolverVarLowerBoundSet 368
SolverVarUpperBoundGet 369
SolverVarUpperBoundSet 370
SolverVarIntegerTypeGet 371
SolverVarIntegerTypeSet 372
SolverVarGroupIndexGet 373
SolverVarGroupIndexSet 374
SolverVarConeTypeGet 375
SolverVarConeTypeSet 376
SolverVarConeIndexGet 377
SolverVarConeIndexSet 378
SolverVarInitialValueGet 379
SolverVarInitialValueSet 380
SolverVarFinalValue 381
SolverVarDualValue 382
SolverVarDualLower 383
SolverVarDualUpper 384
SolverVarMeanAbsDev 385
SolverVarSemiVariance 386
SolverVarSemiDeviation 388
SolverVarValueAtRisk 389
SolverVarCondValueAtRisk 390
SolverVarMeanCI 391
SolverVarStdDevCI 393
SolverVarCITrials 394
SolverVarTarget 395
Function Method Functions 396
SolverFunctionAdd 397
SolverFunctionRemove 398
SolverFcnGetCorrelation 399
SolverFcnGetFrequency 400
SolverFcnNonNegative 401
SolverFcnRelation 402
Function Property Functions 403
SolverFcnCount 404
SolverFcnNameGet 405
SolverFcnNameSet 406
SolverFcnTypeGet 407

SolverFcnTypeSet 408
SolverFcnValueGet 409
SolverFcnValueSet 410
SolverFcnSizeGet 411
SolverFcnSizeSet 412
SolverFcnPosition 413
SolverFcnStatistics 414
SolverFcnPercentiles 415
SolverFcnAllTrials 416
SolverFcnLowerBoundGet 417
SolverFcnLowerBoundSet 418
SolverFcnUpperBoundGet 419
SolverFcnUpperBoundSet 420
SolverFcnFinalValue 421
SolverFcnDualValue 422
SolverFcnDualLower 423
SolverFcnDualUpper 424
SolverFcnMeanAbsDev 425
SolverFcnSemiVariance 426
SolverFcnSemiDeviation 427
SolverFcnValueAtRisk 428
SolverFcnCondValueAtRisk 429
SolverFcnMeanCI 430
SolverFcnStdDevCI 432
SolverFcnCITrials 433
SolverFcnTarget 434
SolverFcnChanceTypeGet 435
SolverFcnChanceTypeSet 436
SolverFcnChanceGet 437
SolverFcnChanceSet 438
SolverFcnInitValueGet 439
SolverFcnInitValueSet 440
Environment Method Functions 441
SolverEnvInit 442
SolverEnvFree 443
SolverEnvRuntimeIDGet 444
SolverEnvRuntimeIDSet 445
SolverEnvLockCodeGet 446
SolverEnvLicenseGet 447

SolverEnvLicenseSet 448
SolverEngTrialPeriod 449
SolverEnvLicenseFileGet 450
SolverEnvLicenseFileSet 451
SolverEnvExecutableLocation 452
Probability Distribution Method Functions 453
SolverModDistAutoFit 454
SolverModDistCDF 455
SolverModDistCDFInv 456
SolverModDistPDF 457
Simulation Distribution Arguments Functions 458
SolverModDistControlParamsGet 459
SolverModDistControlParamsSet 460
Fitting Method Functions 461
SolverModDistFit 462
SolverModDistInitSample 463
Fitting Quality Criteria Functions 464
SolverModDistFitLimit 465
SolverModDistFitSigLevel 466
SolverModDistFitStatistic 467
SolverModDistFitSuccess 468
SolverModDistFitTest 469
SolverModDistIndepTestGet 470
Initialization Method Function 471
SolverModDistInit 471
Distribution Parameter Functions 472
SolverModDistKurtosis 473
SolverModDistMean 474
SolverModDistMedian 475
SolverModDistMode 476
SolverModDistSkewness 477
SolverModDistVariance 478
Distribution Arguments Functions 479
SolverModDistNumControlParams 480
SolverModDistNumParams 481
SolverModDistParamsGet 482
SolverModDistParamsSet 483
Sample Management Functions 484

SolverModDistSampleGet 485
SolverModDistSampleSet 486
SolverModDistSizeGet 487
SolverModDistSizeSet 488
Parameter Functions 489
SolverParamCount 490
SolverParamTypeGet 491
SolverParamTypeSet 492
SolverParamNameGet 493
SolverParamNameSet 494
SolverParamSizeGet 495
SolverParamSizeSet 496
SolverParamNumTicksGet 497
SolverParamNumTicksSet 498
SolverParamReturnValue 499
SolverParamValue 500
SolverParamAllValuesGet 501
SolverParamAllValuesSet 502
SolverParamLowerBoundGet 503
SolverParamLowerBoundSet 504
SolverParamUpperBoundGet 505
SolverParamUpperBoundSet 506
Worksheet Input Functions 507
SolverInputCount 508
SolverInputName 509
SolverInputValueGet 510
SolverInputValueSet 511
SolverInputSize 512
Functions Error Codes 513
Solver SDK Platform Legacy API Reference 516
Functions and Parameters Overview 517
Functions Overview 518
Solver Parameters Overview 521
Problem Definition Functions 524
loadlp 525
loaquad 527
loadctype 529

loadnlp 530
loadnltype 533
testnltype 535
unloadprob 538
Solution Functions 539
optimize 540
mipoptimize 541
solution 542
objsa 545
rhssa 546
varstat 547
constat 549
Diagnostic Functions 551
findiis 552
getiis 553
iiswrite 555
lpwrite 556
lpread 558
Use Control Functions 561
getuse 562
reportuse 564
getproblimits 565
Callback Functions for Nonlinear Problems 566
funceval 567
jacobian 569
Other Callback Functions 571
setlpcallbackfunc 572
getlpcallbackfunc 573
setmipcallbackfunc 574
getmipcallbackfunc 575
getcallbackinfo 576
Parameter Management Functions 578
setintparam 579
getintparam 580
infointparam 581
setdblparam 583
getdblparam 584
infodblparam 585

setdefaults ... 587
Solver Parameters ... 588
Integer Parameters (General) ... 589
Integer Parameters (Solver Engines) ... 591
Integer Parameters (Mixed-Integer Problems) ... 594
Double Parameters ... 596
Optimization Problems Reference and Code Examples ... 599
Linear Problems ... 600
Definition ... 600
Linear Problem Examples ... 600
Quadratic Problems ... 611
Definition ... 611
Quadratic Problem Examples ... 611
Nonlinear Problems ... 633
Definition ... 633
Nonlinear Problem Examples ... 633
Quadratically Constrained Problems ... 643
Definition ... 643
Quadratically Constrained Problem Examples ... 643
Nonsmooth Problems ... 658
Definition ... 658
Nonsmooth Problem Examples ... 658
Simulation Reference and Code Examples ... 669
Definition of Monte Carlo Simulation ... 669
Simulation Problem Example ... 669
Appendix ... 692
Solver SDK Object-Oriented API Objects Hierarchy ... 692
Index ... 696

Conventions and Acronyms Used in This Document

Click to navigate to the desired topic:

- Formatting Conventions
  - Code Formatting
  - Concepts Formatting
- Design Conventions
- Cross-References
- Acronyms Used

Formatting Conventions

This document uses special text and design conventions to make it easier for you to find and understand the desired information.

Code Formatting

Programming code is formatted in `Courier New font` and, where applicable, MS Visual Studio color conventions are used, i.e. blue for data types, green for comments e.g.:

```
// Creating a new Problem:
CProblem (Solver_Type nSolverType, int nNumVars, int nNumFcns)
```

Concepts Formatting

The Solver SDK Platform reserved words (methods, properties, enumerations, etc.) appear in `Courier New font` not only in the code samples but also in the explanatory text of the document, e.g.:

> You can add a new Engine to the Problem's Engines Collection either to the bottom using `Add(...)` or to a specific position in the `CEngineCollection` (including top) with the `Insert(...)` method.

An exception to this rule is made for the topics in which a reserved word is first introduced and defined. In these topics, the reserved word appears **bold and in the regular paragraph font**, e.g.:

> Class **CSolver()** represents either simulation or optimization.
>
> Class **CSolver()** provides the methods to carry out the simulation, optimization, or some other kind of analysis, using a specific Solver Engine.

All concepts that represent a group of Solver SDK Platform objects (e.g. Engines, Solvers) have their first letter capitalized in order to distinguish this specialized meaning (SDK objects group) from the common meaning of the respective noun.

Design Conventions

Notes offer additional information related to the topic being discussed. They usually appear at the end of the topic and are formatted in a way that makes them stand out from the rest of the text:

Notes:

1. The Solver class is the class in the Solver SDK Platform that performs the analysis or provides a solution such as simulation or optimization, using the currently selected Solver Engine to do the work.
2. The Solver class contains the methods to carry out an analysis or solution for a specific Case, which selects a Model, Engine, and customized Variables and Functions.
3. The Solver class objects for Monte Carlo simulation and deterministic optimization are predefined.

Cross-References

When viewing the document online you use the navigation links to go to to other locations in the document that offer additional information about the concept being cross-referenced. The navigation links can be found either throughout the explanatory text as it is in the example below ("enum Problem_Type" is cross-referenced) ...

> Gets/Sets the Problem type. Indicates if a problem is linear, nonlinear, etc. See enum Problem_Type.

... and at the beginning of each section. In the latter case, right after the section heading, there is a list of links to all the topics in the section in order to make navigation easier.

Acronyms Used

Following is a dictionary of the acronyms used in this document:

Acronym	Stands for
API	**A**pplication **P**rogramming **I**nterface
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
CD-ROM	**C**ompact **D**isc
CDF	**C**umulative **D**istribution **F**unction
DLL	**D**ynamic **L**ink **L**ibrary
EV	**EV**olutionary
GRG	**G**eneralized **R**educed **G**radient
ID	**I**dentifier
IIS	**I**rreducibly **I**nfeasible **S**et
IP	**I**nternet **P**rotocol
IPX	**I**nter-network **P**acket **E**xchange
ISV	**I**ndependent **S**oftware **V**endor
LP	**L**inear **P**rogram
LS	**L**icense **S**erver
MAC address	**M**edia **A**ccess **C**ontrol address
MAPI	**M**essaging **A**pplication **P**rogramming **I**nterface
MIP	**M**ixed-**I**nteger **P**roblem
MSL	**M**ultilevel **S**ingle **L**inkage
NLP	**N**on-**L**inear **P**roblem
NSP	**N**on**S**mooth **P**roblem
OLE	**O**bject **L**inking and **E**mbedding
PC	**P**ersonal **C**omputer
PSI	**P**olymorphic **S**preadsheet **I**nterpreter

QCP	**Q**uadratically **C**onstrained **P**rogram/**P**roblem
QP	**Q**uadratic **P**roblem
SB (Johnson SB distribution)	**S**ystem **B**ounded
SDK	**S**oftware **D**evelopment **K**it
SOCP	**S**econd **O**rder **C**one **P**rogramming
SPX	**S**equenced **P**acket **E**xchange
SU (Johnson SU distribution)	**S**ystem **U**nbounded
TCP	**T**ransmission **C**ontrol **P**rotocol
VB	**V**isual **B**asic

Solver SDK Platform Object-Oriented API Reference

Click to navigate to the desired section:

- Classes and Enumerations Overview
- Primary Classes
- Secondary Classes

Classes and Enumerations Overview

Primary Classes

Member Name	Description
CProblem	Represents a class that creates a new Problem.
CSolver	Represents either simulation or optimization.
CEngine	Represents implementation of a Solver that can carry out its type of analysis or solution.
CEvaluator	Represents user-written code to be called by an Engine when various events occur during the solution of a Problem.
CModel	Defines how the user's model can be evaluated.
CVariable	Represents a vector of variables, all of the same type, in the Model.
CFunction	Represents a vector of functions, all of the same type, in the Model.
CCase	Represents a Case or scenario (with associated Case Variable or parameter values) for which the user wants to find a Solution.
CEnvironment	Contains global data and methods that are independent of any Problem.
CEngineCollection	Implements a Collection of Engines.
CEngineParamCollection	Implements a Collection of Engines.
CSolverCollection	Implements a Collection of Solvers.
CEvaluatorCollection	Implements a Collection of Evaluators.
CFunctionCollection	Implements a Collection of Functions.
CVariableCollection	Implements a Collection of Variables.
CCaseCollection	Implements a Collection of Cases.

Secondary Classes

Member Name	Description
CDoubleMatrix	Represents an indexable array of doubles of dimension *m* (rows) by

	n (columns).
CDependMatrix	Represents an indexable array of integers of values drawn from the Depend_Type enum.
CStatistics	Implements statistics for variable and function values across a set of trial points.
COptIIS	Represents an Irreducibly Infeasible Subset of the constraints and variable bounds in an optimization problem.
CEngineStat	Implements current status of an Engine object.
CEngineLimit	Implements the limits for an Engine.
CEngineParam	Implements a single Engine parameter.
CDistribution	Fits a distribution to sample data and, inversely, constructs distribution of known type and parameters.
CSolverException	Implements the exception class for the Solver SDK Platform.

Enumerations

Member Name	Description
APIerr	Specifies different types of error conditions encountered during program execution.
License_Loc	Specifies the meaning of the licensing string.
Solver_Type	Specifies optimization objective or simulation.
Depend_Type	Specifies the nature of an element in the matrix of function values.
Problem_Type	Specifies the problem type.
Variable_Type	Specifies the variable type.
Function_Type	Specifies the function type.
Eval_Type	Specifies the evaluation type.
Engine_Action	Specifies the Engine action.
Array_Order	Specifies the arrangement of elements in an array.
File_Format	Specifies file format.
Distribution_Type	Specifying the distribution type for an uncertain variable.
DistFit_Type	Enumeration for specifying the distribution types fitted by methods Fit and Autofit.

`Frequency_Type`	Specifies the frequency type to be returned.
`Simulate_Status`	Specifies the status of a simulation.
`Optimize_Status`	Specifies the status of an optimization.
`Integer_Type`	Specifies the type of an integer variable.
`Cone_Type`	Specifies the cone type.
`Cons_Rel`	Specifies the relation between the left-hand side and the right-hand side of the constraints.
`IIS_Status`	Specifies the status of the Irreducibly Infeasible Subset (IIS) of constraints.
`License_SelectorCode`	Specifies the selector code for the license.
`DistParam_Type`	Specifies the type of the distribution parameters.
`Dist_VarFcn`	Specifyies whether the call applies to an uncertain variable or function.

Primary Classes

Click to navigate to the desired topic:

- class CProblem
- class CSolver
- class CEngine
- class CEvaluator
- class CModel
- class CVariable
- class CFunction
- class CEnvironment
- class CEngineCollection
- class CEngineParamCollection
- class CSolverCollection
- class CEvaluatorCollection
- class CFunctionCollection
- class CVariableCollection

class CProblem

Class **CProblem()** represents a function that creates a new problem.

Class **CProblem()** properties describe the type of analysis to be performed.

Syntax:

```
CProblem()
```

Usage:

C++	`CProblem()`
C#	`public Problem()`
VB .NET	`Public Sub New()`
Description	Creates a new Problem.

C++	`CProblem(const CProblem& rCopy)`
Description	Copy constructor - shares the internal handle with the input problem parameter.

C++	`CProblem (Solver_Type nSolverType, int nNumVars, int nNumFcns)`
C#	`public Problem(SolverPlatform.Solver_Type nSolverType, System.Int32 nNumVars, System.Int32 nNumFcns)`
VB .NET	`Public Sub New(ByVal nSolverType As SolverPlatform.Solver_Type, ByVal nNumVars As Integer, ByVal nNumFcns As Integer)`
Description	Creates a new Problem, initializes the type of the Solver used and the number of variables and functions.

Properties:

Property Name	Property Type	Property Attributes	Description
`Name`	String	get/set	Gets/Sets the name of the Problem.
`ProblemType`	Problem_Type	get/set	Gets/Sets the Problem type. Indicates if a problem is linear, nonlinear, etc. See enum Problem_Type.
`ObjectiveIndex`	Integer	get/set	Gets/Sets the index to the current objective function in the optimization.

`Solver`	CSolver	get/set	Gets/Sets the current Solver.
`Engine`	CEngine	get/set	Gets/Sets the current Engine.
`Model`	CModel	get	Retrieves the current Model.
`Case`	CCase	get/set	For future use.
`VarCase`	CVariable	get/set	For future use.
`VarUncertain`	CVariable	get/set	Gets/Sets the current uncertain variable block.
`FcnUncertain`	CFunction	get/set	Gets/Sets the current uncertain function.
`VarDecision`	CVariable	get/set	Gets/Sets the current decision variable block.
`FcnConstraint`	CFunction	get/set	Gets/Sets the current constraint function block.
`FcnObjective`	CFunction	get/set	Gets/Sets the current objective function block.
`VarRecourse`	CVariable	get/set	Gets/Set the current recourse variable block.
`VarCase`	CVariable	get/set	For future use.
`FcnCase`	CFunction	get/set	For future use.
`FcnChance`	CFunction	get/set	Gets/Sets the current chance constraint function block.
`TrialPeriod`	integer	get	Gets the number of days left until the Solver Platform trial license expires.

Collections:

- Solvers
- Engines
- Evaluators
- Variables
- Functions
- Cases
- Parameters

Methods:

Method Name	Description
`Init`	Initializes the problem from particular type and number of variables

	and functions. This method could be called once after the problem has been constructed by its empty constructor.
`Load`	Loads a problem from a file.
`Save`	Saves the problem to a file.

Methods Usage:

C++

```
Init(Solver_Type SolverType, int NumVars, int NumFcns)
Load(string ProblemSpecFile, File_Format FileFormat)
Save(string ProblemSpecFile, File_Format FileFormat)
```

C#

```
public void Init(SolverPlatform.Solver_Type SolverType,
              System.Int32 NumVars , System.Int32 NumFcns)
public void Load(System.String ProblemSpecFile)
public void Load(System.String ProblemSpecFile,
              SolverPlatform.File_Format FileFormat)
public void Save(System.String ProblemSpecFile,
              SolverPlatform.File_Format FileFormat)
```

VB .NET

```
Public Sub Init(ByVal SolverType As SolverPlatform.Solver_Type,
              ByVal NumVars As Integer, ByVal NumFcns As
              Integer)
Public Sub Load(ByVal ProblemSpecFile As String)
Public Sub Load(ByVal ProblemSpecFile As String, ByVal
              FileFormat As SolverPlatform.File_Format)
Public Sub Save(ByVal ProblemSpecFile As String, ByVal
              FileFormat As SolverPlatform.File_Format)
```

VB 6

```
Sub Init(SolverType As Solver_Type, NumVars As Long, NumFcns As
              Long)
Sub Load(FileName As String, [FileFormat As File_Format =
              File_Format_All])
Sub Save(FileName As String, FileFormat As File_Format)
```

Methods Properties:

Method `Init`

Property Name	Property Type	Description
`SolverType`	Solver_Type	Solver type: Solver_Type_Maximize, Solver_Type_Minimize, or Solver_Type_Simulate

`NumVars`	integer	Number of variables.
`NumFcns`	integer	Number of functions.

Method `Load`

Property Name	Property Type	Description
`ProblemSpecFile`	String	Name and path of the file from which the Problem is loaded.
`FileFormat`	File_Format	Format of the file. If not specified, the file format is recognized automatically. If `FileFormat` is omitted, the Solver Platform tries to choose the file format.

Method `Save`

Property Name	Property Type	Description
`ProblemSpecFile`	String	Name and path of the file to which the Problem is saved.
`FileFormat`	File_Format	Format of the file.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet12.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

Note:

You must use either `CProblem prob(...)` or `CProblem* prob = new CProblem` to create an instance of a problem on the stack or on the heap, respectively. In the latter case, to release memory when finished, you need to invoke the destructor manually with `delete prob`.

You can call the `Load()` method to read and load a problem from files in certain common file formats.

class CSolver

Class **CSolver()** represents either a simulation or an optimization.

Class **CSolver()** provides the methods to carry out the simulation or optimization process using a specific Solver Engine. The results of these methods are anticipated by the API.

Syntax:

```
CSolver()
```

Usage:

C++	`CSolver()`
C#	`public Solver()`
VB .NET	`Public Sub New()`
Description	Creates a new Solver object. In most cases, used internally.

C++	`CSolver(_Solver_Type solvertype)`
C#	`public Solver(SolverPlatform.Solver_Type solverType)`
VB .NET	`Public Sub New(ByVal solverType As SolverPlatform.Solver_Type)`
Description	Creates a new Solver object and initializes its type.

C++	`CSolver(const CSolver& rSolver)`
Description	Copy constructor - shares the internal handle with the input Solver parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`SolverType`	Solver_Type	get/set	Gets/Sets the Solver type. (The Solver type depends on the Problem's objective: minimization, maximization, simulation or finding a feasible solution.)
`Problem`	CProblem	get	Retrieves the Problem associated with ths Solver.
`Index`	integer	get	Retrieves the index in the Solver Collection.

NumSimulations	integer	get/set	Gets/Sets the number of simulations to run.
NumTrials	integer	get/set	Gets/Sets the number of trials per simulation. If not set, the default number of 1000 simulations will pe performed.
NumSolutions	integer	get/set	Gets/Sets the number of solutions to return. Used for multiple solutions.
NumOptimizations	integer	get/set	Gets/Sets the number of optimizations to be run. Used for multiple optimizations.
OptimizationIndex	integer	get/set	Gets/Sets the index for the current optimization. Used for multiple optimizations.
SolutionIndex	integer	get/set	Gets/Sets the index of the current solution. Used for multiple solutions.
SimulationIndex	integer	get/set	Gets/Sets the index for the current simulation. Used for multiple solutions.
FunctionCurrent	CFunction	get	Retrieves the currently evaluated function.
FunctionType	Function_Type	get	Retrieves the type of the currently evaluated function.
FunctionIndex	integer	get	Gets the index of the currently evaluated function in the flat address space. Used in Hessian evaluators.
SimulateStatus	Simulate_Status	get	Retrieves the status after simulation. Indicates the outcome of a simulation.
OptimizeStatus	Optimize_Status	get	Retrieves the status after optimization. Indicates the outcome of a optimization.
OptIIS	COptIIS	get	Retrieves the Irreducible Infeasible Subset found by IISFind.

Methods:

Method Name	Description
MakeCurrent	Makes the specified Solver current for the associated Problem.
Report	Creates report for the Solver's activity.
Simulate	Invokes the Simulate functionality of the Solver.
Optimize	Invokes the Optimize functionality of the Solver.
IISFind	Searches for an Irreducible Infeasible Subset for an infeasible problem.

Methods Usage:

C++

```
MakeCurrent()

Report (string ReportSpecFile, string ReportName, int
              ReportOutline)

Simulate()

Optimize()

IISFind()
```

C#

```
public void MakeCurrent()

public void Report(System.String ReportSpecFile, System.String
              ReportName, System.Int32 ReportOutline)

public void Simulate()

public void Optimize()

public void IISFind()
```

VB .NET

```
Public Sub MakeCurrent()

Public Sub Report(ByVal ReportSpecFile As String, ByVal
              ReportName As String, ByVal ReportOutline As
              Integer)

Public Sub Simulate()

Public Sub Optimize()

Public Sub IISFind()
```

VB 6

```
Sub MakeCurrent()

Sub Report(ReportSpecFile As String, ReportName As String,
              ReportOutline As Long)

Sub Simulate()

Sub Optimize()
```

```
Sub IISFind()
```

Methods Properties:

Method `Report`

Property Name	Property Type	Description
`ProblemSpecFile`	string	Name and path of the file (or worksheet name string) to store the report information.
`ReportName`	string	Name of the report.
`ReportOutline`	integer	Specifies whether to outline the report: `0` = don't outline; `1` = outline the report

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

Notes:

1. The Solver class is the class in the Solver SDK Platform that performs the analysis or provides a solution to a simulation or optimization problem, using the currently selected Solver Engine to do the work.
2. The Solver class objects for Monte Carlo simulation and deterministic optimization are predefined.

class CEngine

Class **CEngine()** represents Solver Engines.

Solver Engines are the implementations of individual Solvers. Each of them can carry out its own type of analysis or solution performed on a Problem. Solver Engines deliver two kinds of results:

1. Solution values (accessed through the Variable and Function objects)
2. Performance statistics (accessed through the Engine object)

An Engine has properties describing the type of analysis it can perform and the solution information it can produce, the types and the sizes of the models it can solve, and its own options and parameters for controlling the solution process.

Syntax:

```
CEngine()
```

Usage:

C++	`CEngine()`
C#	`public Engine()`
VB .NET	`Public Sub New()`
Description	Creates an empty Engine.

C++	`CEngine(string szName, string szFileSpec)`
C#	`public Engine(System.String Name, System.String FileSpec)`
VB .NET	`Public Sub New(ByVal Name As String, ByVal FileSpec As String)`
Description	Creates an Engine object and loads external Engine dynamic library.

C++	`CEngine(const CEngine& copy)`
Description	Copy constructor - shares the internal handle with the input Engine parameter.

Static Constants:

Constant Name	Description
`LPQPName`	Name of the built-in LP/QP Engine.
`LPName`	Name of the built-in LP Simplex Engine.
`SOCPName`	Name of the built-in SOCP Barrier Engine.

GRGName	Name of the built-in GRG Nonlinear Engine.
EVOName	Name of the built-in Evolutionary Engine.
MCSName	Monte Carlo Simulation.

Parameters:

Property Name	Property Type	Property Attributes	Description
Problem	Problem	get	Retrieves the Problem associated with this Engine.
SolverType	Solver_Type	get	Retrieves the Solver type. See enum Solver_Type.
ProblemType	Problem_Type	get	Retrieves the Problem type. See enum Problem_Type.
Name	string	get	Retrieves the name of the current Engine. If NULL, then the name defined inside Engine DLL is used.
FileSpec	string	get	Retrieves the file location of the Engine's dynamic library. For built-in Engines, returns an empty string.
Params	CEngineParam	get	Retrieves all parameters for the current Engine.
Limit	CEngineLimit	get	Retrieves the Engine limits indexed by Problem Type.
Index	integer	get	Retrieves the index in the EngineCollection the current Engine belongs to.
TrialPeriod	integer	get	Retrieves the number of days until Solver SDK Platform trial license expires.
Stat	EngineStat	get	Retrieves the current Engine status.

Methods:

Method Name	Description
Init	Loads external Engine dynamic library. The method is valid only when an empty constructor has been used.
MakeCurrent	Makes the specified Engine current for the associated Problem.

`ParamLoad`	Loads Engine's parameters from a file.
`ParamSave`	Saves Engine's parameters to a file.
`ParamReset`	Resets all Engine's parameters to their default values.

Methods Usage:

C++

```
Init(string Name, string FileSpec)
MakeCurrent()
ParamLoad(string FileName)
ParamSave(string FileName)
ParamReset()
```

C#

```
public void Init(System.String Name, System.String FileSpec)
public void MakeCurrent()
public void ParamLoad(System.String FileName)
public void ParamSave(System.String FileName)
public void ParamReset()
```

VB .NET

```
Public Sub Init(ByVal Name As String, ByVal FileSpec As String)
Public Sub MakeCurrent()
Public Sub ParamLoad(ByVal FileName As String)
Public Sub ParamSave(ByVal FileName As String)
Public Sub ParamReset()
```

VB 6

```
Sub Init(Name As String, FileSpec As String)
Sub MakeCurrent()
Sub ParamLoad(FileName As String)
Sub ParamSave(FileName As String)
Sub ParamReset()
```

Methods Properties:

Method `Init`

Property Name	Property Type	Description
`Name`	string	Name of the Engine. If NULL, then a name defined inside the Engine DLL is used.
`FileSpec`	string	Path to the Engine dynamic library.

Method `ParamLoad`

Property Name	Property Type	Description
`FileName`	string	Path to the Engine parameter file.

Method `ParamSave`

Property Name	Property Type	Description
`FileName`	string	Path to the Engine parameter file.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

Notes:

1. When a problem is first created, it has a Collection of Engines for different types of problems: the LP/Quadratic, SOCP Barrier, GRG Nonlinear, and Evolutionary Solvers for optimization problems, and the simulation Engine.
2. You can add a new Engine to the problem's Engines Collection either to the bottom using `Add(...)` or to a specific position in the `CEngineCollection` (including top) with the `Insert(...)` method.

To make the new Engine current, the user has to call Engine's `MakeCurrent()` method or must set the Engine property of the Problem object. If not, then the best Engine for the problem type will be selected by the SDK during the call to `Optimize()`.

3. The `EngParam`, `EngLimit` and `EngStat` objects are returned by the Engine and provide access to Engine properties that are specific to the type of Solver Engine.

class CEvaluator

Class **CEvaluator()** is a user-defined callback interface. It represents custom code to be called by an Engine when various events occur during the solution of a Problem.

The various APIs define different callback interfaces. They can be distinguished in three groups- interfaces, events and callback functions. For each API the most common (and the most supported) callback interface is used:

- C++ - interface (IEvaluator)
- COM - event
- .NET - event (EvaluateEventHandler) and interface (INEvaluator)
- Java - interface (IEvaluator)
- Procedural - callback function
- Matlab - callback function

You can supply an `EvaluateEventHandler` event or `INEvaluator` interface to compute values for problem functions, samples for uncertain variables or just simply to iterate during the calculation process.

When a Problem is first created, its Evaluators Collection is empty. You can add Evaluators to the Collection, specifying the type of Solver and type of event or evaluation it is designed to handle. Some of these types (e.g. subproblems and iterations) are designed only for Engine progress monitoring and reporting; others (such as functions and gradients) call for evaluation of the Model, i.e. computation of outputs from inputs. The Evaluator receives a reference to the Problem, which it can use to access the Model and/or the Variable(s) and Function(s) you have defined.

Syntax:

```
CEvaluator()
```

Usage:

C++	`CEvaluator()`
C#	`public Evaluator()`
VB .NET	`Public Sub New()`
Description	Constructs an Evaluator object.

C++	`CEvaluator(const CEvaluator& evaluator)`
Description	Copy constructor - shares the internal handle with the input evaluator parameter.

C++	`CEvaluator(IEvaluator* pEvaluator, void* pRefUser)`
C#	`public Evaluator(SolverPlatform.INEvaluator EvalI, System.Object RefUser)`
VB .NET	`Public Sub New(ByVal EvalI As SolverPlatform.INEvaluator, ByVal RefUser As Object)`
Description	Constructs an empty evaluator and initializes it with the user specific data.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Problem`	Problem	get	Retrieves the Problem associated with this evaluator.
`EvalType`	Eval_Type	get	Retrieves the Evaluator type. Each Evaluator is called to perform a different action during the calculation process. See enum Eval_Type.
`RefUser`	void	get/set	Gets/Sets user-specific data.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

Notes:

1. An Evaluator must be defined if the API cannot compute the values of variables and/or functions itself; Evaluators may also be defined to assess or report Engine progress towards a solution, or to compute values, such as function gradients, that may improve the solution process.
2. Depending on the type of event, the Evaluator may need or want to get or set properties of the Model, or (for programming convenience) of Variables or Functions. On return, the Evaluator indicates whether the Engine currently in use should take certain prescribed actions (e.g. stop, restart, continue).
3. For .NET Users, the SDK has an Evaluate event for CEvaluator class.

interface IEvaluator

Interface **IEvaluator** is a user-defined callback interface. You can supply an **IEvaluator** interface to compute values for problem functions, samples for uncertain variables or just simply to iterate during the calculation process.

Syntax:

```
virtual _Engine_Status Evaluate(CEvaluator& evaluator)
```

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CModel

Class **CModel()** defines how the user's Model can be evaluated.

A Model represents user-written code that can be supplied with numeric inputs and evaluated to compute numeric outputs. Normally, you write an Evaluator in some language (e.g. C/C++, Visual Basic, Excel, Matlab) that accepts input values and computes output values; but for linear and quadratic optimization models where you supply all required coefficients, the API can evaluate the model without any user-written code.

Model inputs and outputs are associated with Variable and Function objects, i.e. the Model class also maps between model names for variables and functions and their corresponding objects and indices.

Syntax:

```
CModel()
```

Usage:

C++	`CModel()`
C#	`public Model()`
VB .NET	`Public Sub New()`
Description	Constructs a Model object. In most cases, used internally.

C++	`CModel(const CModel& copy)`
Description	Copy constructor - shares the internal handle with the input Model parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Problem`	Problem	get	Retrieves the Problem associated with this model.
`NumVariables[]`	Integer	get	Retrieves the number of variables of a particular type. Indexed by `Variable_Type`.
`NumFunctions[]`	Integer	get	Retrieves the number of functions of a particular type. Indexed by `Function_Type`.

NumAllGrad	Integer	get	Retrieves the number of elements (nonzeroes) in the Jacobian matrix of first derivative (i.e. gradient) values associated with Variable_Type and Function_Type. (The same as prob.Model. AllGradValue[ftype] .NumElements.) Indexed by both Variable_Type and Function_Type. Variable_Type can be omitted.
NumFcnHess	NumFcnHessCollection	get	Retrieves the number of elements (nonzeroes) in the Hessian matrix of second derivative values for a specific function. (The same as prob.Model. FcnHessValue [ftype][index].NumElements Indexed by Function_Type, function index[1], and Variable_Type. Variable_Type can be omitted.
AllGradValue	CDoubleMatrix	get/sets	Gets/Sets the Jacobian matrix of first derivative (i.e. gradient) values for a model. Indexed by both Variable_Type and Function_Type. Variable_Type can be omitted.
AllGradError	CDoubleMatrix	get	Gets the matrix of AllGradError elements. (Can be used with a Jacobian evaluator.) Indexed by both Variable_Type and Function_Type. Variable_Type can be omitted. The AllGradError matrix elements contain the difference between a value computed by the Solver (using finite differencing)

[1] *index* is the function number in flat address space.

			and the value passed by your Jacobian (i.e. gradient) Evaluator. It is used only if you have defined an `Eval_Type_GradError` Evaluator.
`AllGradDepend`	CdependMatrix	get/set	Gets/Sets the dependency matrix associated with the Jacobian matrix of first derivative (i.e. gradient) values for the model. Indexed by both `Variable_Type` and `Function_Type`. `Variable_Type` can be omitted.
`FcnHessValue`	CDoubleMatrix	get/set	Gets/Sets the coefficients of the Hessian matrix of second derivative values for a specific function. Indexed by Function_Type, function index and Variable_Type `Variable_Type` can be omitted.
`FcnHessDepend`	CDependMatrix	get/set	Gets/Sets the dependency matrix associated with the Hessian matrix of second derivative values for a specific function. Property of linear and quadratic functions. Indexed by `Function_Type`, function index, and Variable_Type. Variable_Type can be omitted.
`NameVariable`	CVariable	get	Retrieves `NameVariable`. Reserved for future use.
`NameFunction`	CFunction	get	Retrieves `NameFunction`. Reserved for future use.
`IndexVariable`	CVariable	get	Retrieves a variable pointed to by the index in the flat address space.
`IndexFunction`	CFunction	get	Retrieves a function pointed to by the function in the flat address space.
`Params`	Integer	get	Gets parameters from the Excel model.
Properties of uncertain functions only:			
`FcnDistribution`	CDistribution	get	Gets/Sets the distributions

			associated with uncertain functions. Indexed by the function index.[1] Refer to class CDistribution for distribution properties and methods.
Properties of uncertain variables only:			
`NumCorrelations`	Integer	get	Retrieves the number of elements (nonzeroes) in the matrix of correlations between uncertain variables.
`DistCorrelations`	CDoubleMatrix	get/set	Gets/Sets the matrix of correlations between uncertain variables.
`VarDistribution`	CDistribution	get/set	Gets/Sets the distributions associated with uncertain variables. Indexed by the variable index.[2] Refer to class CDistribution for distribution properties and methods.
Properties of linear and quadratic functions only:			
`AllLinear`	CDoubleMatrix	get/set	Gets/Sets the coefficient matrix of a linear model. This is the same as the `AllGradValue` matrix, when all problem functions are linear, and its elements are constant in this case. Indexed by both `Variable_Type` and `Function_Type`. `Variable_Type` can be omitted.
`FcnQuadratic`	CDoubleMatrix	get/set	Gets/Sets the coefficient matrix of a quadratic function. This is the same as the Hessian matrix for a function, when that function is quadratic, and its elements are constant in this case. Indexed by `Function_Type`, function index[3], and `Variable_Type`. `Variable_Type` can be omitted.

[1] *index* is the function number in flat address space.

[2] *index* is the function number in flat address space.

[3] *index* is the function number in flat address space.

Methods:

Method Name	Description
DependTest	Determines the Problem Type from random or specified test values. If the model is linear, DependTest fills in the All_Linear matrix for you. So, if you have an Evaluator, but the model is linear, you can use DependTest to fill in the AllLinear matrix once and then reuse it in the future.
DependCheck	For backward compatibility only.
VarIndexName	Reserved for future use.
FcnIndexName	Reserved for future use.
NameVarIndex	Reserved for future use.
NameFcnIndex	Reserved for future use.

Methods Usage:

C++

```
DependTest (Variable_Type vartypes, int numtests, double
            []testvals)
DependTest(Variable_Type vartypes, int numtests)
DependCheck(bool bTransformed, Check_Type nCheckFor)
VarIndexName (int index, int elements, out string Name)
FcnIndexName (int index, int elements, out string Name)
NameVarIndex (string Name, out int Index, out int Elements)
NameFcnIndex (string Name, out int Index, out int Elements)
```

C#

```
public void DependTest(SolverPlatform.Variable_Type vartypes,
            System.Int32 numtests)
public void DependTest(SolverPlatform.Variable_Type vartypes,
            System.Int32 numtests, double[] testvals)
public void DependCheck(bool Transformed)
public void DependCheck(bool Transformed, Check_Type CheckFor)
public void VarIndexName(System.Int32 index, System.Int32
            elements, System.String Name)
public void FcnIndexName(System.Int32 index, System.Int32
            elements, System.String Name)
public void NameVarIndex(System.String pszName, int * Index, out
            int[] Elements)
public void NameFcnIndex(System.String pszName, int * Index, out
            int[] Elements)
```

VB .NET

```
Public Sub DependTest(ByVal vartypes As
            SolverPlatform.Variable_Type, ByVal numtests As
            Integer, ByVal testvals() As Double)
Public Sub DependTest(ByVal vartypes As
            SolverPlatform.Variable_Type, ByVal numtests As
            Integer)
public Sub DependCheck(bool Transformed)
public Sub DependCheck(bool Transformed, Check_Type CheckFor)
Public Sub VarIndexName(ByVal index As Integer, ByVal elements
            As Integer, ByRef Name As String)
Public Sub FcnIndexName(ByVal index As Integer, ByVal elements
            As Integer, ByRef Name As String)
```

VB 6

```
Sub DependTest(VarType As Variable_Type, NumTests As Long,
            [TestVals])
Sub DependCheck(Transformed As Boolean, CheckFor As Check_Type)
Sub VarIndexName(Index As Long, Elements As Long, Name As
            String)
Sub FcnIndexName(Index As Long, Elements As Long, Name As
            String)
Sub NameVarIndex(Name As String, Index, Elements)
Sub NameFcnIndex(Name As String, Index, Elements)
```

Methods Properties:

Method `DependTest`

Property Name	Property Type	Description
`Vartypes`	Variable_Type	Type of variables.
`Numtests`	Integer	Number of tests to be performed.
`Testvals`	Double	Array of test values. If `NULL` or not defined, the Solver uses random algorithm to generate values.

Method `DependCheck` **For backward compatibility only, not implemented in the Solver SDK Platform 2020.**

Property Name	Property Type	Description
`transformed`	Boolean	Indicates if the Transformed property is set to True or False in the Excel model. For backward compatibility only, not implemented in 2020.
`checkFor`	Check_Type	Type of analysis requested: Check_Type_Convexity, Check_Type_Default, Check_Type_Gradients, or Check_Type_Structure. For backward compatibility only, not implemented in 2020.

Method `VarIndexName`

Property Name	Property Type	Description
`Index`	integer	Element's index to translate.
`Elements`	integer	The count of elements to be translated.
`Name`	string	Output name.

Method `FcnIndexName`

Property Name	Property Type	Description
`Index`	integer	Element's index to translate.
`Elements`	integer	The count of elements to be translated.
`Name`	string	Output name.

Method `NameVarIndex`

Property Name	Property Type	Description
`Name`	string	Model name to be translated.
`Index`	integer	Starting index.
`Elements`	integer	The count of elements to be translated.

Method `NameFcnIndex`

Property Name	Property Type	Description
`Name`	string	Model name to be translated.
`Index`	integer	Starting index.
`Elements`	integer	The count of elements to be translated.

Notes:

1. For linear and quadratic functions, you can set the `AllLinear` and `FcnQuadratic` properties to supply coefficients, allowing the Solver SDK Platform to compute values, gradients and Hessians for the functions of the model.
2. For uncertain variables defined by known distribution functions that don't depend on decision variables, you can set the appropriate element of the `VarDistribution` array to an initialized CDistribution object, allowing the Solver SDK Platform to compute sample values for these variables. `CDistribution` can be initialized by supplying a string such as "PsiNormal(1,2)" in its constructor, or by setting its `DistType` and `Params` properties. The `Fit` and `AutoFit` methods may also be used to automatically set the distribution type and parameters, based on user-supplied sample data.

class CVariable

Class **CVariable()** implements a vector of variables, all of the same type, in the Model.

Class **CVariable()** has a `VariableType` and a `Name`[1]. A Variable has properties that are "inputs" to the solution process, such as `LowerBound`, `UpperBound`, `IntegerType`, and `InitialValue`, and properties that are "outputs" from the solution process, such as `FinalValue`, `DualValue`, `DualLower`, and `DualUpper`.

Syntax:

```
CVariable()
```

Usage:

C++	`CVariable()`
C#	`public Variable()`
VB .NET	`Public Sub New()`
Description	Constructs a Variable object.

C++	`CVariable(_Variable_Type nVarType, string szName = string(), int nSize = 0)`
C#	`public Variable(SolverPlatform.Variable_Type vartype, System.String Name, System.Int32 Size)`
VB .NET	`Public Sub New(ByVal vartype As SolverPlatform.Variable_Type, ByVal Name As String, ByVal Size As Integer)`
Description	Constructs a Variable object with a specified type, name and size. Name and size are optional, but size can be used only if there is a name specified, i.e. you can specify either `nVarType` only or `nVarType` + `szName` or `nVarType` + `szName` + `nSize`.

[1] The Name property must be set either in the constructor or afterwards by setting the property. If you add the variable to the `CVariableCollection` without a name, the first consistency check will fail and the `APIerr_Variable_NameInvalid` exception will be returned. The Name of the variable is not checked in the constructor, but if you add the variable to the collection of variables, then the following two checks over its name are performed:

1. Check for an empty name. If the name is empty, then an exception is returned.
2. Check for a duplicate name. If the name is the same as the name of an existing variable in the collection the exception is returned.

The checks are not case-sensitive.
If you don't specify the variable name in the constructor then you have to specify it just before adding to the collection (property Name).

C++	`CVariable(const CVariable& var)`
Description	Copy constructor - shares the internal handle with the input variable parameter.

Properties:

Property Name	Property Type	Property Attributes	Description
`Problem`	Problem	Get	Retrieves the Problem associated with this variable.
`VariableType`	Variable_Type	get/set	Gets/Sets the type of the variable vector.
`Name`	string	get/set	Gets/Sets the name of the variable vector.
`Size`	integer	get/set	Gets/Sets the number of elements in the variable vector.
`Index`	integer	Get	Gets the index in Variables Collection.
`Position`	integer	Get	Gets the position of the variable vector; Starting position of the variable vector in the Model flat address space.
`Value`	double	Get	Retrieves the current values of the variable vector.
`Statistics`	CStatistics	Get	Retrieves the statistics for this variable.
Properties of Uncertain Variables only:			
`Percentiles`	CDoubleMatrix	get/set	Gets/Sets the matrix of percentiles.
`AllTrials`	CDoubleMatrix	get/set	Gets/Sets the matrix of `AllTrials`.
Properties of Decision Variables only:			
`LowerBound`	double	get/set	Gets/Sets the lower bound of the function vector.
`UpperBound`	double	get/set	Gets/Sets the upper bound of the function vector.
`IntegerType`	Integer_Type	get/set	Gets/Sets the integer type. Indicates what type of integer a variable is, if integer.
`GroupIndex`	integer	get/set	Gets/Sets the group's index. (The group's index indicates to which alldifferent group the variables belong.) 0 indicates that a variable is not a part of any alldifferent group.

ConeType	Cone_Type	get/set	Gets/Sets the cone type. (The cone type indicates what type of cone a variable belongs to, if part of a cone.)
ConeIndex	integer	get/set	Gets/Sets the cone index. (The cone index indicates to which cone group a variable belongs.) 0 indicates that a variable is not a part of a cone.
InitialValue	double	get/set	Gets/Sets the initial value of the variable vector.
FinalValue	double	Get	Retrieves the final values of the variable vector after optimization or simulation.
DualValue	double	Get	Retrieves the dual values of the variable vector.
DualLower	double	Get	Retrieves the dual lower values of the variable vector. (“Dual lower values” are the lower range values of the objective coefficients for which the dual values are valid.)
DualUpper	double	Get	Retrieves the dual upper values of the variable vector. (“Dual upper values” are the upper range values of the objective coefficients for which the dual values are valid.)

Methods:

Method Name	Description
MakeCurrent	Makes the variable current for the associated problem.
GetFrequency	Returns frequency of observations in each bin – sorted up or down if Frequency_Type is cumulative.
GetCorrelation	Returns the correlation values between the uncertain variables.
NonNegative	Sets the lower bound of the variable vector to 0, if no bound has been specified.

Methods Usage:

C++

```
MakeCurrent()
GetCorrelation(int varindex, out double []correlation)
GetFrequency (Frequency_Type frequencytype, int numbins,double
              [] binBoundary)
NonNegative()
```

C#

```
public void MakeCurrent()
public double[] GetCorrelation(System.Int32 varindex)
public SolverPlatform.DoubleMatrix
              GetFrequency(SolverPlatform.Frequency_Type
              frequencytype, double[] binBoundary)
public void NonNegative()
```

VB .NET

```
Public Sub MakeCurrent()
Public Function GetCorrelation(ByVal varindex As Integer) As
              Double()
Public Function GetFrequency(ByVal frequencytype As
              SolverPlatform.Frequency_Type, ByVal
              binBoundary() As Double) As
              SolverPlatform.DoubleMatrix
Public Sub NonNegative()
```

VB 6

```
Sub MakeCurrent()
Function GetCorrelation(varindex As Long) As Double()
Function GetFrequency(FrequencyType As Frequency_Type,
              BinBoundary) As DoubleMatrix
Sub NonNegative()
```

Methods Properties:

Method `GetCorrelation`

Property Name	Property Type	Description
`Varindex`	Integer	Index of the other uncertain variable to which the elements of this variable vector are correlated.
`Correlation`	Double	Coefficients of correlation between the elements of the variable vector and the other uncertain variable.

Method `GetFrequency`

Property Name	Property Type	Description
`frequencytype`	Frequency_Type	Density (Frequency_Type_Density, Cumulative (Frequency_Type_Cumulative), or Reverse Cumulative (Frequency_Type_RevCumulative)
`Numbins`	Integer	Number of histogram bars or cumulative plot points.
`binBoundary`	Double	`numbins`+1. Lower bound of each bin, the extra

element is the upper bound of the last bin.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CFunction

Class **CFunction()** implements a vector of functions, all of the same type, in the Model.

A Function has a `FunctionType` and an optional `Name`[1]. A Function has properties that are "inputs" to the solution process, such as `LowerBound` and `UpperBound`, and properties that are "outputs" from the solution process, such as `FinalValue`, `DualValue`, `DualLower`, and `DualUpper`.

Syntax:

```
CFunction()
```

Usage:

C++	`CFunction()`
C#	`public Function()`
VB .NET	`Public Sub New()`
Description	Constructs a Function object.

C++	`CFunction(_Function_Type nFunctionType, string szName = string(), int nSize = 0)`
C#	`public Function(SolverPlatform.Function_Type FuncType, System.String Name, System.Int32 Size)`
VB .NET	`Public Sub New(ByVal FuncType As SolverPlatform.Function_Type, ByVal Name As String, ByVal Size As Integer)`
Description	Constructs a Function object with a specified type, name and size. Name and size are optional, but size can be used only if there is a name specified, i.e. you can specify either `nFunctionType` only or `nFunctionType` + `szName` or `nFunctionType` + `szName` + `nSize`.

C++	`CFunction(const CFunction& fn)`
Description	Copy constructor - shares the internal handle with the input function parameter.

[1] The Name is optional until the function is associated with a `CFunctionCollection`.

Properties:

Property Name	Property Type	Property Attributes	Note
`Problem`	Property	Get	Retrieves the Problem associated with the Function.
`FunctionType`	Function_Type	get/set	Gets/Sets the type of the function vector.
`Name`	string	get/set	Gets/Sets the name of the function vector.
`Value`	double	get/set	Gets/Sets the type of the function vector.
`Size`	integer	get/set	Gets/Sets the number of elements in the function vector.
`Index`	integer	Get	Retrieves the index in Functions Collection.
`Position`	integer	Get	Retrieves the position of the function vector (the starting position of the function vector in the Model flat address space).
`Statistics`	Statistics	Get	Retrieves statistics for the Function.
`LowerBound`	double	get/set	Gets/Sets the lower bound of the function vector.
`UpperBound`	double	get/set	Gets/Sets the upper bound of the function vector.
`DualValue`	double	Get	Retrieves the dual values of the function vector.
`DualLower`	double	Get	Retrieves the dual lower values of the function vector. (“Dual lower values” are the lower range values of the right-hand sides for which the dual values are valid.)
`DualUpper`	double	Get	Retrieves the dual upper values of the function vector. (“Dual upper values” are the upper range values of the right-hand sides for which the dual values are valid.)
`InitialValue`	double	get/set	Gets/Sets the initial value of the function vector.
`FinalValue`	double	Get	Gets the final values of the function vector after optimization or simulation.
`Chance`	double	get/set	Gets/Sets the chance for the function block.

ChanceType	Chance_Type	get/set	Gets/Sets the chance type for the function block - Chance_Type_CVar, Chance_Type_Expval, Chance_Type_None, Chance_Type_Uset, or Chance_Type_VaR.
Slack	double[]	gets	Gets slack values of the function block.

Properties of Uncertain Functions only:

Percentiles	CDoubleMatrix	Get	Retrieves the matrix of percentiles.
AllTrials	CDoubleMatrix	Get	Retrieves the matrix of all trials.

Methods:

Method Name	Description
MakeCurrent	Makes the function current for the associated Problem.
GetCorrelation	Returns the correlation values for some functions.
GetFrequency	Returns the frequency of observations in each bin – sorted up or down if Frequency_Type is cumulative.
NonNegative	Sets the lower bound of the function vector to 0.
Relation	Sets the upper and lower bounds on the function vector, based on the relation (greater than, less than or equal to) and the right-hand side.

Methods Usage:

C++

```
MakeCurrent()
GetCorrelation(int fcnindex, out double []correlation)
GetFrequency(Frequency_Type frequencytype, int numbins, out
              double binBoundary, out double binFrequency)
NonNegative()
Relation(Cons_Rel []relation, double []rhs)
```

C#

```
public void MakeCurrent()
public double[] GetCorrelation(System.Int32 fcnindex)
public SolverPlatform.DoubleMatrix
              GetFrequency(SolverPlatform.Frequency_Type
              frequencytype, double[] binBoundary)
public void NonNegative()
public void Relation(SolverPlatform.Cons_Rel[] relation,
              double[] rhs)
```

VB .NET

```
Public Sub MakeCurrent()
Public Function GetCorrelation(ByVal fcnindex As Integer) As
              Double()
Public Function GetFrequency(ByVal frequencytype As
              SolverPlatform.Frequency_Type, ByVal
              binBoundary() As Double) As
              SolverPlatform.DoubleMatrix
Public Sub NonNegative()
Public Sub Relation(ByVal relation() As SolverPlatform.Cons_Rel,
              ByVal rhs() As Double)
```

VB 6

```
Sub MakeCurrent()
Function GetCorrelation(FcnIndex As Long) As Double()
Function GetFrequency(FrequencyType As Frequency_Type,
              BinBoundary) As DoubleMatrix
Sub NonNegative()
Sub Relation(Relation, rhs)
```

Methods Properties:

Method `GetCorrelation`

Property Name	Property Type	Description
`fcnindex`	Integer	Index of the uncertain function with which elements of the Function vector are correlated.
`correlation`	Double	Correlation coefficients between uncertain function and elements of this function vector.

Method `GetFrequency`

Property Name	Property Type	Description
`frequencytype`	Frequency_Type	Density or cumulative. For more information, refer to enum Frequency_Type.
`numbins`	Integer	Number of histogram bars or cumulative plot points.
`binBoundary`	Double	`numbins`+1. Lower bound of each bin, the extra element is the upper bound of the last bin.
`binFrequency`	Double	Frequency of observations in each bin – sorted up or down if `Frequency_Type` is cumulative. For more information, refer to enum Frequency_Type.

Method `Relation`

Property Name	Property Type	Description
`relation`	Cons_Rel	Array of relations.
`rhs`	Double	Array of values.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CEnvironment

Class **CEnvironment()** contains global data and methods that are independent of any Problem.

There can be only one instance of class **CEnvironment()**. If there are more than one instance of an Environment object, then all instances share the same data.

An instance **CEnvironment()** may be created by any ISV application that will be shipped to customers with its own form of license control. The ISV application must set the Environment object's ISVID property (RuntimeID) before creating any Problem instances.

Syntax:

```
CEnvironment()
```

Usage:

C++	`CEnvironment()`
C#	`public Environment()`
VB .NET	`Public Sub New()`
Description	Constructs an Environment object.

C++	`CEnvironment(string strRuntimeID)`
C#	`public Environment(System.String RuntimeID)`
VB .NET	`Public Sub New(ByVal RuntimeID As String)`
Description	Constructs CEnvironment based on the RuntimeID.

Parameters:

Property Name	Property Type	Property Attributes	Description
`License`	string	get/set	Gets/Sets the usage license.
`LicenseLoc`	License_Loc	get/set	Specifies the type of location of the license. Currently, it can be only License_Loc_File. (See enum License_Loc.) If changed to something different, an APIerr_NotYetImplemented exception will be thrown. License_Loc_Str – The string contains the licensing information.

			License_Loc_Variable – The string is the environment variable name. License_Loc_File – The string is the name/path of the license file. License_Loc_Registry – The string is the name/path of the Windows Registry entry. When searching for the license, the behavior of the Solver SDK Platform property depends on whether the FRONTLIC environment variable is set. If FRONTLIC exists, then the Solver SDK Platform will look for the default license in the path indicated by the FRONTLIC environment variable. If FRONTLIC does not exist. The Solver Platform will look for the default license in the default location (file solver.lic in the same folder as the SolverSDK.dll file).
`RuntimeID`	string	get/set	Gets/Sets runtime application identifier.
`LockCode`	integer	get	Retrieves the lock code for the machine on which Solver SDK Platform is running. Accepts no value or any valid value of the `License_SelectorCode` enum. For a list of the valid values, refer to enum License_SelectorCode
`TrialPeriod`	integer	get	Retrieves the number of days until Solver SDK Platform trial license expires.
`ExecutableLocation`	string	get	Retrieves the location of the Solver SDK executable.
`LicenseFile`	string	get/set	Gets/Sets the license file.

Methods:

Method Name	Description
`Create`	Initializes CEnvironment object based on RuntimeID.

Method `Create`

Property Name	Property Type	Description
`runtimeID`	string	Passes RuntimeID to Solver SDK for use with a runtime license.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

Note:

You must use either `CProblem prob(...)` or `CProblem* prob = new CProblem` to create an instance of a problem on the stack or on the heap, respectively. In the latter case, to release memory when finished, you need to invoke the destructor manually with `delete prob`.

The `License` and `LicenseLoc` properties indicate where the Platform License may be found. If this is not done, the Solver SDK Platform will look for a valid license in a default location (e.g. in the same direcory as where the dll/.so library is located).

You can call the `Load()` method to read and load a problem from files in certain common file formats.

Collections Classes

Click to navigate to the desired topic:

- class CEngineCollection
- class CEngineParamCollection
- class CSolverCollection
- class CEvaluatorCollection
- class CFunctionCollection
- class CVariableCollection

class CEngineCollection

Class **CEngineCollection()** implements a Collection of Engines.

Syntax:

```
CEngineCollection()
```

Usage:

C++	`CEngineCollection()`
C#	`public EngineCollection()`
VB .NET	`Public Sub New()`
Description	Constructs an Engine Collection object. In most cases, used internally.

C++	`CEngineCollection(const CEngineCollection& rCopy)`
Description	Copy constructor - shares the internal handle with the input Engine Collection parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Count`	Integer	get	Retrieves the number of elements in the Collection.
Indexed Properties:			
`CEngine [int index]`	Integer	get	Retrieves the Engine at the specified index.
`CEngine [string str]`	String	get	Retrieves the Engine with the specified name. The name is not case-sensirive.

Methods:

Method Name	Description
`Add`	Adds a new Engine at the end of the Collection. Returns the index (as integer) at which the Engine has been added.
`Insert`	Inserts an Engine into the Collection at a specified index.
`Discover`	Searches an optionally given path for any external Engines and adds the Engines it will automatically add them to the Engine Collection. The method returns an integer, indicating how many Engines have been added. If no path or an empty string has been passed, it searches in the same folder where the SolverSDK.DLL resides.

Clear	Removes external Engines from the Collection. (Internal Engines cannot be removed.)
Remove	Removes an Engine. It can be either an Engine at a specified index or with a specified name. (Only external Engines can be removed.)

Methods Usage:

C++

```
Add(CEngine engine)
Insert(int Index , Engine engine)
SOLVER_INT Discover(CConstString szPath)
SOLVER_INT Discover()
Clear()
Remove(int Index)
Remove(String Index)
```

C#

```
public System.Int32 Add(SolverPlatform.Engine engine)
public void Insert(System.Int32 Index, SolverPlatform.Engine
            engine)
public System.Int32 Discover(System.String Path)
public System.Int32 Discover()
public void Clear()
public void Remove(System.Int32 Index)
public void Remove(System.String Index)
```

VB .NET

```
Public Function Add(ByVal engine As SolverPlatform.Engine) As
            Integer
Public Sub Insert(ByVal Index As Integer, ByVal engine As
            SolverPlatform.Engine)
Public Function Discover(ByVal Path As String) As Integer
Public Function Discover() As Integer
Public Sub Clear()
Public Sub Remove(ByVal Index As Integer)
Public Sub Remove(ByVal Index As String)
```

VB 6

```
Function Add(pVal As Engine) As Long
Sub Insert(Index As Long, pVal As Engine)
Function Discover([Path As String]) As Long
Sub Clear()
Sub Remove(Index)
```

Methods Properties:

Method `Add`

Property Name	Property Type	Description
`engine`	Engine	Engine to be added at the end of the Collection.

Method `Insert`

Property Name	Property Type	Description
`Index`	integer	Zero-based index at which new Engine should be inserted.
`engine`	Engine	The Engine to insert.

Method `Discover`

Property Name	Property Type	Description
`szPath`	string	Path to be searched for external Engines.

Method `Remove` (engine at index position)

Property Name	Property Type	Description
`Index`	integer	Zero-based index of the Engine to remove.

Method `Remove` (engine with specified name)

Property Name	Property Type	Description
`Index`	string	The name of the Engine to remove.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CEngineParamCollection

Class **CEngineParamCollection()** implements a Collection of Engines.

Syntax:

```
CEngineParamCollection()
```

Usage:

C++	`CEngineParamCollection()`
C#	`public EngineParamCollection()`
VB .NET	`Public Sub New()`
Description	Constructs an Engine Parameter Collection object. In most cases, used internally.

C++	`CEngineParamCollection(const CEngineParamCollection& rCopy)`
Description	Copy constructor - shares the internal handle with the input Engine Collection parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Count`	integer	get	Retrieves the number of elements in the Collection.
Indexed Properties:			
`EngineParam [int index]`	integer	get	Retrieves the parameter at the specified index.
`EngineParam [string str]`	string	get	Retrieves the parameter with the specified name. The name is not case-sensirive.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CSolverCollection

Class **CSolverCollection()** implements a Collection of Solvers.

Syntax:

```
CSolverCollection()
```

Usage:

C++	`CSolverCollection()`
C#	`public SolverCollection()`
VB .NET	`Public Sub New()`
Description	Constructs a Solver Collection object. In most cases, used internally.

C++	`CSolverCollection(const CSolverCollection& rCopy)`
Description	Copy constructor - shares the internal handle with the input Solver Collection parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Count`	Integer	get	Retrieves the number of elements in the Collection.
Indexed Property:			
`CSolver [int Index]`	Integer	get	Retrieves the the Solver at the specified index.

Methods:

Method Name	Description
`Clear`	Removes all Solvers associated to the Collection.
`Add`	Adds a new Solver to the end of the Collection. Returns the index (as integer) at which the Solver has been added.
`Insert`	Inserts a Solver into the Collection at a specified index.
`Remove`	Removes the Solver at a specified index.

Methods Usage:

C++

```
Clear()
Add(CSolver s)
Insert(int Index , Solver solver)
Remove(int Index)
```

C#

```
public void Clear()
public System.Int32 Add(SolverPlatform.Solver s)
public void Insert(System.Int32 Index, SolverPlatform.Solver s)
public void Remove(System.Int32 Index)
```

VB .NET

```
Public Sub Clear()
Public Function Add(ByVal s As SolverPlatform.Solver) As Integer
Public Sub Insert(ByVal Index As Integer, ByVal s As
                SolverPlatform.Solver)
Public Sub Remove(ByVal Index As Integer)
```

VB 6

```
Sub Clear()
Function Add(pVal As Engine) As Long
Sub Insert(Index As Long, pVal As Engine)
Sub Remove(Index)
```

Methods Properties:

Method `Add`

Property Name	Property Type	Description
`s`	Solver	The Solver to be added at the end of the Collection.

Method `Insert`

Property Name	Property Type	Description
`Index`	integer	Zero-based index for the Solver to be inserted at.
`solver`	Solver	The Solver to insert.

Method `Remove`

Property Name	Property Type	Description
`Index`	integer	Zero-based index of the Solver to remove.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CEvaluatorCollection

Class **CEvaluatorCollection()** implements a Collection of Evaluators.

Syntax:

```
CEvaluatorCollection()
```

Usage:

C++	`CEvaluatorCollection()`
C#	`public EvaluatorCollection()`
VB .NET	`Public Sub New()`
Description	Constructs an Evaluator Collection object. In most cases, used internally.

C++	`CEvaluatorCollection(const CEvaluatorCollection& rCopy)`
Description	Copy constructor - shares the internal handle with the input Evaluator Collection parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
Indexed Property:			
`Evaluator [int index]`	Eval_Type	get/set	Gets/Set the element at the specified index.

Methods:

Method Name	Description
`Clear`	Removes all elements from the Collection.
`Remove`	Removes the Evaluator of a specified type.

Methods Usage:

C++

```
Clear()
Remove(Eval_Type nEvalType)
```

C#

```
public void Clear()
```

```
public void Remove(System.Int32 nEvalType)
```

VB .NET

```
Public Sub Clear()
Public Sub Remove(ByVal nEvalType As Integer)
```

VB 6

```
Sub Clear()
Sub Remove(Index As Eval_Type)
```

Methods Properties:

Method `Remove`

Property Name	Property Type	Description
`nEvalType`	Eval_Type	Evaluator to remove. For more information, refer to enum Eval_Type.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CFunctionCollection

Class **CFunctionCollection()** implements a Collection of Functions.

Syntax:

```
CFunctionCollection()
```

Usage:

C++	`CFunctionCollection()`
C#	`public FunctionCollection()`
VB .NET	`Public Sub New()`
Description	Constructs a Function Collection object. In most cases, used internally.

C++	`CFunctionCollection(const CFunctionCollection& rCopy)`
Description	Copy constructor - shares the internal handle with the input function Collection parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Count`	integer	get	Retrieves the number of elements in the Collection.
Indexed Properties:			
`Function [int Index]`	integer	get	Retrieves the Function at the specified index.
`Function [string Index]`	string	get	Retrieves the Function with the specified name. The name is not case-sensitive.

Methods:

Method Name	Description
`Clear`	Removes all Functions associated to this Collection.
`Add`	Adds a new Function at the end of the Collection. Returns the index at which the Function has been added.
`Insert`	Inserts a function into the Collection at a specified index.

Remove	Removes the Function at a specified index or with a specified name.

Methods Usage:

C++

```
Clear()
int Add(Function fn)
Insert(int Index, Function fn)
Remove(int Index)
Remove(string Index)
```

C#

```
public void Clear()
public System.Int32 Add(SolverPlatform.Function fn)
public void Insert(System.Int32 Index, SolverPlatform.Function
            fn)
public void Remove(System.Int32 Index)
public void Remove(System.String Index)
```

VB .NET

```
Public Sub Clear()
Public Function Add(ByVal fn As SolverPlatform.Function) As
            Integer
Public Sub Insert(ByVal Index As Integer, ByVal fn As
            SolverPlatform.Function)
Public Sub Remove(ByVal Index As Integer)
Public Sub Remove(ByVal Index As String)
```

VB 6

```
Sub Clear()
Function Add(pVal As Function) As Long
Sub Insert(Index As Long, pVal As Function)
Sub Remove(Index)
```

Methods Properties:

Method `Add`

Property Name	Property Type	Description
`fn`	Function	The Function to be added at the end of the Collection.

Method `Insert`

Property Name	Property Type	Description
`Index`	Integer	Zero-based index of the Function to be inserted at.

`fn`	Function	The Function to insert.

Method `Remove` (function at a specified index)

Property Name	Property Type	Description
`Index`	Integer	Zero-based index of the Function to remove.

Method `Remove` (function with the specified name)

Property Name	Property Type	Description
`Index`	String	name of the function to remove.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CVariableCollection

Class **CVariableCollection()** implements a Collection of Variables.

Syntax:

```
CVariableCollection()
```

Usage:

C++	`CVariableCollection()`
C#	`public VariableCollection()`
VB .NET	`Public Sub New()`
Description	Constructs a Variable Collection object. In most cases, used internally.

C++	`CVariableCollection(const CVariableCollection& rCopy)`
Description	Copy constructor - shares the internal handle with the input Variable Collection parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Count`	integer	get	Retrieves the number of elements in the Collection.
Indexed Properties:			
`CVariable [int Index]`	integer	get	Retrieves the variable at the specified index.
`CVariable [string Index]`	integer	get	Retrieves the variable with the specified name. The name is not case-sensirive.

Methods:

Method Name	Description
`Add`	Adds a new variable at the end of the Collection.
`Insert`	Inserts a variable into the Collection at a specified index.
`Remove`	Removes a variable at a specified index or with a specified name.

Clear	Removes all variables associated with the collection.

Methods Usage:

C++

```
int Add(Variable s)
Insert (int Index, Variable var)
Remove(int Index)
Remove(string Index)
void Clear()
```

C#

```
public System.Int32 Add(SolverPlatform.Variable var)
public void Insert(System.Int32 Index, SolverPlatform.Variable
            var)
public void Remove(System.Int32 Index)
public void Remove(System.String Index)
public void Clear()
```

VB .NET

```
Public Function Add(ByVal var As SolverPlatform.Variable) As
            Integer
Public Sub Insert(ByVal Index As Integer, ByVal var As
            SolverPlatform.Variable)
Public Sub Remove(ByVal Index As Integer)
Public Sub Remove(ByVal Index As String)
Public Sub Clear()
```

VB 6

```
Function Add(pVal As Variable) As Long
Sub Insert(Index As Long, pVal As Variable)
Sub Remove(Index)
Sub Clear()
```

Methods Properties:

Method `Add`

Property Name	Property Type	Description
`S`	Variable	The variable to be added at the end of the Collection.

Method `Insert`

Property Name	Property Type	Description
`Index`	integer	Zero-based index for the variable to be inserted at.

`Var`	Variable	The variable to insert.

Method `Remove` (variable at a specified index)

Property Name	Property Type	Description
`Index`	integer	Zero-based index of the variable to remove.

Method `Remove` (variable at a specified name)

Property Name	Property Type	Description
`Index`	string	Name of the variable to remove.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

Secondary Classes

Click to navigate to the desired topic:

- Matrix Classes
  - class CDoubleMatrix
  - class CDependMatrix
- class CStatistics
- class COptIIS
- class CEngineStat
- class CEngineLimit
- class CEngineParam
- Class CDistribution
- class CSolverException
- class CInputData
- class CModelParam
- Class CParameters**()**

Matrix Classes

A matrix is a rectangular array of numbers, with *m* rows and *n* columns. The SDK provides two classes to easily create and manipulate a matrix - CDoubleMatrix() and CDependMatrix().

The internal representation of a matrix may be dense or sparse. A dense matrix contains a 1D array of *m*n* doubles, one for each vector element. A sparse matrix contains an array of non-zero doubles, one for each non-zero element, and row and column indices in the matrix of the non-zero elements. In the Object-Oriented API CDoubleMatrix() and CDependMatrix() objects are accepted as arguments.

class CDoubleMatrix

Class **CDoubleMatrix()** represents a rectangular (*m* rows by *n* columns) matrix of *doubles* (floating point values).

Syntax:

```
CDoubleMatrix()
```

Usage:

C++	`CDoubleMatrix()`
C#	`public DoubleMatrix()`
VB .NET	`Public Sub New()`
Description	Creates an empty double matrix. The number of rows and the number of columns are both zero.

C++	`CDoubleMatrix(const CDoubleMatrix& matrix)`
Description	Copy constructor - shares the internal handle with the input double matrix parameter.

C++	`CDoubleMatrix (int nRows, int nColumns)`
C#	`public DoubleMatrix(System.Int32 Rows, System.Int32 Columns)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer`
Description	Creates a sparse `nRows` by `nColumns` matrix ordered by column.

C++	`CDoubleMatrix (_Array_Order nOrder, int nRows, int nColumns)`
C#	`public DoubleMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer)`
Description	Creates a sparse `nRows` by `nColumns` matrix ordered by input parameter order.

C++	`CDoubleMatrix (int nRows, int nColumns, double fElement)`
C#	`public DoubleMatrix(System.Int32 Rows, System.Int32 Columns, System.Double Element)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer, ByVal Element as Double)`
Description	Creates a dense matrix and initializes all elements to `fElement` parameter.

C++	`CDoubleMatrix (_Array_Order nOrder, int nRows, int nColumns, double fElement)`
C#	`public DoubleMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns, System.Double Element)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer, ByVal Element As Double)`
Description	Creates dense matrix ordered by parameter order and initializes all elements to the `fElement` parameter.

C++	`CDoubleMatrix(int nRows, int nColumns, const double* pElements)`
C#	`public DoubleMatrix(System.Int32 Rows, System.Int32 Columns, double[] Elements)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer, ByVal Elements() As Double)`
Description	Creates dense matrix and initializes it with the `pElements` array. The `pElements` array must be a one-dimensional array of size *rows * columns*.

C++	`CDoubleMatrix(_Array_Order nOrder, int nRows, int nColumns, const double* pElements)`
C#	`public DoubleMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns, double[] Elements)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer, ByVal Elements() As Double)`
Description	Creates a dense matrix ordered by parameter order and initializes it with the `pElements` array. The `pElements` array has to be one-dimensional of size *rows * columns* array.

C++	`CDoubleMatrix (int nRows, int nColumns, int nNumelements, const int* pStarts, const int* pIndices, const double* pElements)`
C#	`public DoubleMatrix(System.Int32 Rows, System.Int32 Columns ,int[] Starts, int[] Indices, double[] Elements)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer, ByVal Starts() As Integer, ByVal Indices() As Integer, ByVal Elements() As Double)`
Description	Creates a sparse matrix ordered by column and initializes it with three arrays – starts, indices and elements.

C++	`CDoubleMatrix (_Array_Order nOrder, int nRows, int nColumns, int nNumElements, const int* pStarts, const int* pIndices, const double* pElements)`
C#	`public DoubleMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns, int[] Starts, int[] Indices, double[] Elements)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer, ByVal Starts() As Integer, ByVal Indices() As Integer, ByVal Elements() As Double)`
Description	Creates a sparse matrix ordered by parameter order and initializes it with the `pElements` array.

C++	`CDoubleMatrix (int nRows, int nColumns, int nNumElements, const int* pStarts, const int* pIndices, double fElement)`
C#	`public DoubleMatrix(System.Int32 Rows, System.Int32 Columns ,int[] Starts, int[] Indices, System.Double Element)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer, ByVal Starts() As Integer, ByVal Indices() As Integer, ByVal Element As Double)`
Description	Creates a sparse matrix ordered by column and initializes it with three arrays – starts, indices and elements. The `pElements` array is automatically created and populated with the pElement value.

C++	`CDoubleMatrix (_Array_Order nOrder, int nRows, int nColumns, int nNumElements, const int* pStarts, const int* pIndices, double fElement)`
C#	`public DoubleMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns, int[] Starts, int[] Indices, System.Double Element)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer, ByVal Starts() As Integer, ByVal Indices() As Integer, ByVal Element As Double)`
Description	Creates a sparse matrix ordered by parameter order and initializes it with the `pElements` array. The `pElements` array is automatically created and populated with the pElement value.

Parameters:

Parameter Name	Description
`nRows`	Number of rows.
`nColumns`	Number of columns.
`nOrder`	Internal order of the matrix – by row or by column.
`fElement`	Element to fill/initialize with.
`pElements`	Array of particular elements values.
`Elements`	Array of particular elements values.
`nNumElements`	Number of non zero-elements of the matrix – the size of the indices and elements arrays.
`pStarts`	Array of starting points. Each element indicates where the particular major dimension starts in the array of indices. The size of `pStarts` is equal to the major dimension.
`pIndices`	Array of indices. Each element is the number of the minor (second) dimension.

Properties:

Property Name	Property Type	Property Attributes	Description
`IsEmpty`	Boolean	get	Flag indicating the matrix is empty. (Empty matrices have zero dimensions.)

IsSparse	Boolean	get	Flag indicating sparse or dense storage of the matrix.
IsPSD	Boolean	get	Flag indicating if the matrix is positive-definite.
ArrayOrder	Array_Order	get/set	Get/Sets the array order of the matrix. (The array order specifies which dimension is major (first) - columns or rows. Default array order is by column (Array_Order_ByCol). Changing the ArrayOrder property transposes the matrix. When the versions without ArrayOrder are used, then the default (by column) ordering is used.
MajorDim	Integer	get	Retrieves the major (first) dimension (rows or columns depending on the ArrayOrder property setting). Columns are the major dimension if the array is arranged column-wise, rows - if row-wise.
MinorDim	Integer	get	Retrieves the minor (second) dimension (rows or columns depending on the ArrayOrder property setting). Rows are the minor dimension if the array is arranged column-wise, columns - if row-wise.
Rows	Integer	get	Retrieves the number of rows (Y dimension) in the matrix. Works the same for either dense or sparse matrices. Does not depend on Array_Order.
Columns	Integer	get	Retrieves the number of columns (X dimension) in the matrix. Works the same for dense or sparse matrices. Doesn't depend on Array_Order.
NumElements	integer	get	In a sparse matrix: Retrieves the actual number of

			non-zero elements stored. In a dense matrix: Retrieves the number of all elements (Rows*Columns).
`Element`	Double	get	Accesses the elements of the matrix as one-dimensional vector. The size of the vector is equal to `NumElements`.
`MajorStart`	Integer	get	Retrieves the start index for the specified major dimension. In either dense or sparse case returns the one-dimensional index where the major dimension starts. The values returned are constant pointers to internal buffers (valid values in the sparse case and `NULL` in the dense case).
`InternalMajorStarts`	Integer	get	Retrieves all the major starts as one-dimensional array. In the dense case, returns `NULL` (nothing in VB.NET).
`MajorLength`	Integer	get	Retrieves the the number of elements in the major (first) dimension of the array. (I.e. the number of elements in one column (if the matrix is arranged column-wise) or in one row (if row-wise).) Works the same with either dense or sparse matrices.
`MinorIndex`	Integer	get	Retrieves the minor indices of the non-zero elements in the matrix. ("Minor indices" of the elements are the indices (i.e. the positions) of the elements in the minor (second) dimension, that is, in the row if the matrix is arranged column-wise, or in the column if the matrix is row-wise.) `MinorIndex` returns constant pointers to internal buffers. The values are valid in the sparse case and `NULL` in the dense case.
`InternalMinorIndices`	Integer	get	Retrieves all minor indices as one dimensional array. In the dense case, returns `NULL` (nothing in

			VB.NET).
`Value`	Double	get/set	Gets/Sets the value of an element of the matrix.
`InternalValues`	Double	get	Retrieves all values as one-dimensional array.

Methods:

Method Name	Description
`Create`	Creates a sparse `nRows` by `nColumns` matrix.
`InitDense`	Creates a dense matrix and initializes it.
`InitSparse`	Creates a sparse matrix and initializes it with two or three arrays – starts, indices (and elements).
`Destroy`	Destroys the matrix. After destroying, the new matrix is empty (both the number of rows and the number of columns equal to 0).
`Clear`	Clears matrix' content. (Makes all elements equal to zero). The number rows and the number of columns remain unchanged.
`NZBgn`	Launches fast update of a matrix' content. Used to quickly populate a matrix object. After calling `NZbgn()`, follow by multiple `.Value(x,y)=z` statements to fill in the values. This will temporarily store these elements until calling `NZEnd()` which puts them in the matrix in one swoop. Once you call `NZBgn()`, you **must** also call either `NZEnd()` to permanently store the values or `NZCancel()` to abort the operation. In the latter case, the matrix objects will preserve its old content.
`NZEnd`	Finishes the fast update operation started with `NZBgn`.
`NZCancel`	Cancels fast update operation started with `NZBgn`.

Methods Usage:

C++

```
Create(int Rows, int Columns)

InitDense(int Rows, int Columns, double [] Elements)
InitDense(int Rows, int Columns, double Element)

InitSparse(int Rows, int Columns, int [] Starts, int [] *,
              double[] Elements)
InitSparse(int Rows, int Columns, int [] Starts, int [] Indices,
              double Element)

Destroy()

Clear()

void NZBgn()
```

```
void NZEnd()
void NZCancel()
```

C#

```
public void Create(System.Int32 Rows, System.Int32 Columns)
public void InitDense(System.Int32 Rows, System.Int32 Columns,
            double[] Elements)
public void InitDense(System.Int32 Rows, System.Int32 Columns,
            System.Double Element)
public void InitSparse(System.Int32 Rows, System.Int32 Columns,
            int[] Starts, int[] Indices, double[] Elements)
public void InitSparse(System.Int32 Rows, System.Int32 Columns,
            int[] Starts, int[] Indices, System.Double
            Element)
public void Destroy()
public void Clear()
public void NZBgn()
public void NZEnd()
public void NZCancel()
```

VB .NET

```
Public Sub Create(ByVal Rows As Integer, ByVal Columns As
            Integer)
Public Sub InitDense(ByVal Rows As Integer, ByVal Columns As
            Integer, ByVal Elements() As Double)
Public Sub InitDense(ByVal Rows As Integer, ByVal Columns As
            Integer, ByVal Element As Double)
Public Sub InitSparse(ByVal Rows As Integer, ByVal Columns As
            Integer, ByVal Starts() As Integer, ByVal
            Indices() As Integer, ByVal Elements() As Double)
Public Sub InitSparse(ByVal Rows As Integer, ByVal Columns As
            Integer, ByVal Starts() As Integer, ByVal
            Indices() As Integer, ByVal Element As Double)
Public Sub Destroy()
Public Sub Clear()
Public Sub NZBgn()
Public Sub NZEnd()
Public Sub NZCancel()
```

VB 6

```
Sub Create(Rows As Long, Columns As Long)
Sub InitDense(Rows As Long, Columns As Long, [Elements])
Sub InitSparse(Rows As Long, Columns As Long, [Starts],
            [Indices], [Elements])
Sub Destroy()
Sub Clear()
```

```
Sub NZBgn()
Sub NZEnd()
Sub NZCancel()
```

Methods Properties:

Method `Create`

Property Name	Property Type	Description
`Rows`	integer	Number of rows.
`Columns`	integer	Number of columns.

Method `InitDense`

Property Name	Property Type	Description
`Rows`	integer	Number of rows.
`Columns`	integer	Number of columns.
`Elements`	double	Array to initialize with.
`Element`	double	Element to initialize with.

Method `InitSparse`

Property Name	Property Type	Description
`Rows`	integer	Number of rows.
`Columns`	integer	Number of columns.
`Starts`	integer	Array of starting points. Each element indicates where the particular major dimension starts in the array of indices. The size of `pStarts` is equal to major dimension.
`Indices`	integer	Array of indices. Each element is the number of minor (second) dimension.
`Elements`	double	Array of particular elements values.
`Element`	double	The element to fill with.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNe40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

Notes:

1. The purpose of the DoubleMatrix() and IntegerMatrix() classes is to make it easier to create a sparse matrix of linear or quadratic coefficients, or of dependency flags, to be passed to the Model class. These classes are not intended to support general-purpose matrix arithmetic or arbitrary manipulation of matrix objects and elements.
2. The `Array_Order` argument is optional – if omitted, `Array_Order_ByCol` is assumed.

class CDependMatrix

Class **CDependMatrix()** represents a matrix of `Depend_Type` values (integers with symbolic names such as `Depend_Type_Linear and Depend_Type_Smooth`. For more information, refer to enum Depend_Type) of dimension *m* rows by *n* columns. The use of `CDependMatrix()` is optional – such a matrix can provide details of the dependencies of functions on variables, for example, in a nonlinear optimization problem, that can be exploited by certain nonlinear Solver Engines.

Syntax:
`CDependMatrix()`

Usage:

C++	`CDependMatrix()`
C#	`public DependMatrix()`
VB .NET	`Public Sub New()`
Description	Creates an empty `_Depend_Type` matrix. The number of rows and the number of columns are both zero.

C++	`CDependMatrix(const CDependMatrix& matrix)`
Description	Copy constructor - shares the internal handle with the input `DependMatrix` parameter.

C++	`CDependMatrix(int nRows, int nColumns)`
C#	`public DependMatrix(System.Int32 Rows, System.Int32 Columns)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer)`
Description	Creates a sparse `nRows` by `nColumns` matrix ordered by column.

C++	`CDependMatrix(_Array_Order nOrder, int nRows, int nColumns)`
C#	`public DependMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer)`
Description	Creates a sparse `nRows` by `nColumns` matrix ordered by input parameter order.

C++	`CDependMatrix(int nRows, int nColumns, _Depend_Type nElement)`
C#	`public DependMatrix(System.Int32 Rows, System.Int32 Columns, SolverPlatform.Depend_Type Element)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer, ByVal Elements() As SolverPlatform.Depend_Type)`
Description	Creates a dense matrix and initializes all elements to the `nElement` parameter.

C++	`CDependMatrix(_Array_Order order, int nRows, int nColumns, _Depend_Type nElement)`
C#	`public DependMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns, SolverPlatform.Depend_Type Element)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer, ByVal Elements() As SolverPlatform.Depend_Type)`
Description	Creates a dense matrix ordered by parameter order and initializes all elements to the `nElement` parameter.

C++	`CDependMatrix(int nRows, int nColumns, const _Depend_Type* pElements)`
C#	`public DependMatrix(System.Int32 Rows, System.Int32 Columns, SolverPlatform.Depend_Type[] Elements)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer, ByVal Elements() As SolverPlatform.Depend_Type)`
Description	Creates a dense matrix and initializes it with the `pElements` array. The `pElements` array has to be a one-dimensional *rows* * *columns* array.

C++	`CDependMatrix(_Array_Order nOrder, int nRows, int nColumns, const _Depend_Type* pElements)`
C#	`public DependMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns, SolverPlatform.Depend_Type[] Elements)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer, ByValElements() AsSolverPlatform.Depend_Type)`
Description	Creates a dense matrix ordered by parameter order and initializes it with the `pElements` array. The `pElements` array has to be a one-dimensional *rows* * *columns* array.

C++	`CDependMatrix (int nRows, int nColumns, int nNumElements, const int* pStarts, const int* pIndices, const_Depend_Type* pElements)`
C#	`public DependMatrix(System.Int32 Rows, System.Int32 Columns, int[] Starts, int[] Indices, SolverPlatform.Depend_Type[] Elements)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer, ByVal Starts() As Integer, ByVal Indices() As Integer, ByVal Elements() As SolverPlatform.Depend_Type)`
Description	Creates a sparse matrix ordered by column and initializes it using three arrays – starts, indices and elements.

C++	`CDependMatrix (_Array_Order nOrder, int nRows, int nColumns, int nNumElements, const int* pStarts, const int* pIndices, const _Depend_Type* pElements)`
C#	`public DependMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns, int[] Starts, int[] Indices, SolverPlatform.Depend_Type[] Elements)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer,ByVal Starts() As Integer, ByVal Indices() As Integer, ByVal Elements() As SolverPlatform.Depend_Type)`
Description	Creates a sparse matrix ordered by order parameter and initializes it using three arrays – starts, indices and elements.

C++	`CDependMatrix (int nRows, int nColumns, int nNumElements, const int* pStarts, const int* pIndices, _Depend_Type nElement)`
C#	`public DependMatrix(System.Int32 Rows, System.Int32 Columns, int[] Starts, int[] Indices, SolverPlatform.Depend_Type Element)`
VB .NET	`Public Sub New(ByVal Rows As Integer, ByVal Columns As Integer, ByVal Starts() As Integer, ByVal Indices() As Integer, ByVal Element As SolverPlatform.Depend_Type)`
Description	Creates a sparse matrix and initializes it using two arrays – starts and indices. The elements array is automatically created and populated with the `fElement` value.

C++	`CDependMatrix (_Array_Order nOrder, int nRows, int nColumns, int nNumElements, const int* pStarts, const int* pIndices, _Depend_Type nElement)`
C#	`public DependMatrix(SolverPlatform.Array_Order order, System.Int32 Rows, System.Int32 Columns, int[] Starts, int[] Indices, SolverPlatform.Depend_Type Element)`
VB .NET	`Public Sub New(ByVal order As SolverPlatform.Array_Order, ByVal Rows As Integer, ByVal Columns As Integer, ByVal Starts() As Integer, ByVal Indices() As Integer, ByVal Element As SolverPlatform.Depend_Type)`
Description	Creates a sparse matrix, ordered by order parameter and initializes it with two arrays – starts and indices. The elements array is automatically created and populated with the `fElement` value.

Parameters:

Refer to class CDoubleMatrix.

Parameters:

Refer to class CDoubleMatrix.

Methods:

Refer to class CDoubleMatrix.

Methods Usage:

C++

```
Create(int Rows, int Columns)

InitDense(int Rows, int Columns, Depend_Type[] Elements)
InitDense(int Rows, int Columns, Depend_Type Element)

InitSparse(int Rows, int Columns, int [] Starts, int [] Indices,
           Depend_Type[] Elements)
InitSparse(int Rows, int Columns, int [] Starts, int [] Indices,
           Depend_Type Element)

Destroy()

Clear()

void NZBgn()

void NZEnd()

void NZCancel()
```

C#

```
public void Create(System.Int32 Rows, System.Int32 Columns)
```

```
public void InitDense (System.Int32 Rows, System.Int32 Columns,
                SolverPlatform.Depend_Type[] Elements)
public void InitDense(System.Int32 Rows, System.Int32 Columns,
                SolverPlatform.Depend_Type Element)
public void InitSparse(System.Int32 Rows, System.Int32 Columns,
                int[] Starts, int[] Indices,
                SolverPlatform.Depend_Type[] Elements)
public void InitSparse(System.Int32 Rows, System.Int32 Columns,
                int[] Starts, int[] Indices,
                SolverPlatform.Depend_Type Element)
public void Destroy()
public void Clear()
public void NZBgn()
public void NZEnd()
public void NZCancel()
```

VB .NET

```
Public Sub Create(ByVal Rows As Integer, ByVal Columns As
                Integer)
Public Sub InitDense(ByVal Rows As Integer, ByVal Columns As
                Integer, ByVal Elements() As
                SolverPlatform.Depend_Type)
Public Sub InitDense(ByVal Rows As Integer, ByVal Columns As
                Integer, ByVal Element As
                SolverPlatform.Depend_Type)
Public Sub InitSparse(ByVal Rows As Integer, ByVal Columns As
                Integer, ByVal Starts() As Integer, ByVal
                Indices() As Integer, ByVal Elements() As
                SolverPlatform.Depend_Type)
Public Sub InitSparse(ByVal Rows As Integer, ByVal Columns As
                Integer, ByVal Starts() As Integer, ByVal
                Indices() As Integer, ByVal Element As
                SolverPlatform.Depend_Type)
Public Sub Destroy()
Public Sub Clear()
Public Sub NZBgn()
Public Sub NZEnd()
Public Sub NZCancel()
```

VB 6

```
Sub Create(Rows As Long, Columns As Long)
Sub InitDense(Rows As Long, Columns As Long, [Elements])
Sub InitSparse(Rows As Long, Columns As Long, [Starts],
                [Indices], [Elements])
Sub Destroy()
Sub Clear()
Sub NZBgn()
```

```
Sub NZEnd()
Sub NZCancel()
```

Methods Properties:

Method `Create`

Property Name	Property Type	Description
`Rows`	integer	Number of rows.
`Columns`	integer	Number of columns.

Method `InitDense`

Property Name	Property Type	Description
`Rows`	integer	Number of rows.
`Columns`	integer	Number of columns.
`Elements`	Depend_Type	Array to initialize with.
`Element`	Depend_Type	Element to initialize with.

Method `InitSparse`

Property Name	Property Type	Description
`Rows`	integer	Number of rows.
`Columns`	integer	Number of columns.
`Starts`	integer	Array of starting points. Each element indicates where the particular major dimension starts in the array of indices. The size of `pStarts` is equal to major dimension.
`Indices`	integer	Array of indices. Each element is the number of minor (second) dimension.
`Elements`	Depend_Type	Array of particular elements values.
`Element`	Depend_Type	The element to fill with.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet20.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

class CStatistics

Class **CStatistics()** implements statistics for uncertain variable and uncertain function values across a set of trial points for a given simulation. Statistics objects are computed by all simulation and some optimization Solver Engines.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Kurtosis`	double	get	Retrieves the kurtosis of the variable or function over the simulation.
`Maximum`	double	get	Retrieves the maximum value of the variable or function during the simulation.
`Mean`	double	get	Retrieves the mean value of the variable or function over the simulation.
`MeanAbsDev`	Double	Get	Gets the average of the absolute deviations of trials from the mean of the uncertain variable or function.
`Minimum`	double	get	Retrieves the minimum value of the variable or function during the simulation.
`Mode`	double	get	Retrieves the mode of the variable or function over the simulation.
`NumErrors`	integer	get	Retrieves the number of errors encountered during the simulation.
`NumValues`	integer	get	Retrieves the number of values over which statistics are calculated.
`Skewness`	double	get	Retrieves the skewness of the variable or function over the simulation.
`StdDev`	double	get	Retrieves the standard deviation of the variable or function over the simulation.
`Variance`	double	get	Retrieves the variance of the variable or function over the simulation.

Methods:

Method Name	Description
`CITrials`	Returns the recommended number of trials given 0 < Alpha < 1 and tolerance (half the width of the confidence interval) for the specified uncertain variable or function.
`CondValueAtRisk`	Returns the expected value of the loss, given that a loss occurs, that

	has frequency 1 minus the specified percentile fro the uncertain variable or function.
MeanCI	Returns the confidence interval of the mean value given 0 < alpha < 1 for the specified uncertain function.
SemiDeviation	Returns the semi-deviation which is the square (or power) root of the semi-variance for the uncertain variable of function.
SemiVariance	Returns the semi-variance, or if the power argument is not equal to 2, the lower partial moment of the trials for the uncertain variable or function.
StdDevCI	Returns the confidence interval of the standard deviation value for the specified uncertain variable or function.
Target	Returns the cumulative probability of the target value in the distribution of trial values for the specified uncertain variable or function.
ValueAtRisk	Returns the loss amount such that greater losses occur with frequency less then 1 minus the specified percentile for the uncertain variable or function.

Examples:

C++

```
'statistics on uncertain variables
CStatistics VarStat = prob.VarUncertain.Statistics;
int i;
for (i = 0; i < nvars; i++)
{
 Console << "Variable " << i << Console.EndL();
 Console << "Minimum  = " << VarStat.Minimum[i] <<
   Console.EndL();
 Console << "Maximum  = " << VarStat.Maximum[i] <<
   Console.EndL();
 Console << "Mean     = " << VarStat.Mean[i] << Console.EndL();
 Console << "StdDev   = " << VarStat.StdDev[i] <<
    Console.EndL();
 Console << "Variance = " << VarStat.Variance[i] <<
    Console.EndL();
 Console << "Skewness = " << VarStat.Skewness[i] <<
    Console.EndL();
 Console << "MeanAbsDev = " << VarStat.MeanAbsDev[i] <<
    Console.EndL();
 Console << "Mode   = " << VarStat.Mode[i] <<
    Console.EndL();
 Console << "NumErrors = " << VarStat.NumErrors[i] <<
    Console.EndL();
 Console << "NumValues = " << VarStat.NumValues[i] <<
    Console.EndL();
 Console << "Kurtosis = " << VarStat.Kurtosis[i] <<
    Console.EndL();
 Console << "Mean CI = " << VarStat.MeanCI(0.95)[i] <<
 Console.EndL();
```

```
Console << "The Conditional-Value-at-Risk of the sampled
uncertain variable " << i+1 << " at the 5th percentile is: " <<
VarStat.CondValueAtRisk(0.95)[i] << Console.EndL();

//we now display the Lower Partial Moments (Semi-variance)
//values for the variables in 3 different ways --- first, we
//compute the LPM with a power of 5, then we compute the LPM
//with the default value of power of 2.0, and then we compute
//the LPM with power 2, with respect to the median value
CDoubleMatrix percentiles = prob.VarUncertain.Percentiles;
dbltemp = percentiles.Value[i][50];
Console<<"Semi-variance (Lower Partial Moment at power 5) of
the sampled uncertain variable " <<i+1<<" : " << prob.
VarUncertain.Statistics.SemiVariance(5)[i]<<Console.EndL();
Console<<"Semi-variance of the sampled uncertain variable "
<< i+1 <<" : " <<
prob.VarUncertain.Statistics.SemiVariance()[i] <<
Console.EndL();
Console<<"Semi-variance of the sampled uncertain variable with
respect to the median value " <<i+1<<" : " <<
prob.VarUncertain.Statistics.SemiVariance(2,
dbltemp)[i]<<Console.EndL();

//now compute the Semi-deviation corresponding to the 3 values
//of Semi-variance above;
dbltemp = percentiles.Value[i][50];
Console<<"Semi-deviation at power 5 of the sampled uncertain
variable " <<i+1<<" : " << prob.VarUncertain.Statistics.
SemiDeviation(5)[i]<<Console.EndL();
Console<<"Semi-deviation of the sampled uncertain variable "
<<i+1<<" : "<< myprob.VarUncertain.Statistics.
SemiDeviation()[i]<<Console.EndL();
Console<<"Semi-deviation of the sampled uncertain variable
with respect to the median value " <<i+1<<" : " <<
prob.VarUncertain.Statistics.SemiDeviation(2,dbltemp)[i]<<
Console.EndL();


//we now use the Target function to compute the "Inverse CDF";
dbltemp = percentiles.Value[i][50];
//this should always return 50%;
Console<<"Using the Target function, for the sampled uncertain
variable "<<i+1<<" the percentage of samples that are less than
or equal to " << dbltemp << "is: << prob.VarUncertain.
Statistics.Target(dbltemp)[i]<<Console.EndL();

//we now use the VaR function and check if it returns the
correct value;
dbltemp = percentiles.Value[i][5];
Console<<"The Value-at-Risk of the sampled uncertain variable
"<<i+1<<" at the 5th percentile is: "<<
prob.Variables[0].Statistics.ValueAtRisk(0.95)[i]<<Console.
EndL();
Console<<"This VaR value should be equal to "<<-
dbltemp<<Console.EndL();
```

```
 //we now use the CVaR function to compute the Conditional-VaR;
 Console<<"The Conditional-Value-at-Risk of the sampled
 uncertain variable "<<i+1<<" at the 5th percentile is: "<<
 myprob.Variables[0].Statistics.CondValueAtRisk(0.95)[i]<<
 Console.EndL();


      myprob.Variables[0].Statistics.SemiDeviation(2,
dbltemp)[i]<<Console.EndL();
 Console << "SemiDeviation = " << VarStat.SemiDeviation(5)[i] <<
 Console.EndL();
 Console << "semiVariance = " <<
 VarStat.CondValueAtRisk(0.95)[i] << Console.EndL();
 Console << "Mean CI = " << VarStat.MeanCI(0.95)[i]
 << Console.EndL();
 Console << "SemiDeviation = " << VarStat.SemiDeviation(5)[i] <<
 Console.EndL();









}

//obtain statistics on uncertain functions
CStatistics FcnStat = problem.Functions[0].Statistics;
CDoubleMatrix FcnPct = problem.Functions[0].Percentiles;
for (i = 0; i < nfcns; i++)
{
 Console << "Function " << i;
 Console << "Minimum  = " << FcnStat.Minimum[i] <<
    Console.Endl();
 Console << "Maximum  = " << FcnStat.Maximum[i] <<
    Console.Endl();
 Console << "Mean     = " << FcnStat.Mean[i] << Console.Endl();
 Console << "StdDev   = " << FcnStat.StdDev[i] <<
    Console.Endl();
 Console << "Variance = " << FcnStat.Variance[i] <<
    Console.EndL();
 Console << "Skewness = " << FcnStat.Skewness[i] <<
    Console.EndL();
 Console << "Kurtosis = " << FcnStat.Kurtosis[i] <<
    Console.EndL();
 Console << "10th Percentile = " << FcnPct.Value[i][10] <<
    Console.EndL();
 Console << "90th Percentile = " << FcnPct.Value[i][90] <<
    Console.EndL();
 Console << "CITrials = " << FcnStat.CITrials(0.95, 0.01)[0] <<
    Console.EndL();
 Console << "Conditional Value at Risk = " <<
    FcnStat.CondValueAtRisk(0.95)[0] << Console.EndL();
```

```
 Console << "Mean CI = " << FcnStat.MeanCI(0.95)[0] <<
 Console.EndL();
 Console << "Mean Absolute Deviation = " <<
 FcnStat.MeanAbsDev[0] << Console.EndL();
 Console << "Value at Risk = " << FcnStat. ValueAtRisk(0.95)[0]
 << Console.EndL();
 Console << "Target = " <<
 myprob.Variables[0].Statistics.Target(dbltemp)[0];
}
      }
```

class COptIIS

Class **COptIIS()** represents an Irreducibly Infeasible Subset of the constraints and variable bounds in an optimization problem. You may call IISFind() at any time after prob.Solver.Optimize() has reported the problem as infeasible. This routine computes an Irreducibly Infeasible Subset (IIS) of the constraints, such that your problem, with just those constraints, is still infeasible, but if any one constraint is dropped from the subset, the problem becomes feasible.

Parameters:

Property Name	Property Type	Property Attributes	Description
`BoundIndex`	integer address	get	Retrieves the bound index. (The bound index is a vector with `NumBounds` elements, indicating which bounds are included in IIS.)
`BoundStatus`	IIS_Status address	get	Retrieves the bound status. (The bound status is a vector with `NumBounds` elements, indicating the status of the bounds included in the IIS.) Refer to enum IIS_Status for description of each status.
`ConstraintIndex`	integer address	get	Retrieves the constraint index. (The constraint index is a vector with `NumConstraint` elements, indicating which constraints are included in the IIS.)
`ConstraintStatus`	IIS_Status address	get	Retrieves the constraint status. (The constraint index is a vector with `NumConstraint` elements, indicating the status of the constraints included in IIS.)
`NumBounds`	integer	get	Retrieves the number of variable bounds included in the IIS.
`NumConstraints`	integer	get	Retrieves the number of constraints included in the IIS.

Examples:

C++

```
//run an optimization
problem.Solver.Optimize();
_Optimize_Status status = problem.Solver.OptimizeStatus;
```

```
//check status of optimization model.  If infeasible, call
//IISFind to create an Irreducibly Infeasible Subset (IIS) of
//the constraints and variable bounds
if(status = Optimize_Status_Infeasible)
 {
  //call IISFind to compute the Irreducibly Infeasible Subset
  //(IIS)
  problem.Solver.IISFind();
  int i;
  //get the status of each variable bound
  for(i = 0; i < problem.Solver.OptIIS.NumBounds; i ++)
    Console << "Bound[" << problem.Solver.OptIIS.BoundIndex[i]
    <<"], status is " << problem.Solver.OptIIS.BoundStatus[i] <<
    Console.EndL();

  //get the status of each constraint
  for(i = 0; i < problem.Solver.OptIIS.NumConstraints; i ++)
    Console << "Constraint[" <<
    problem.Solver.OptIIS.ConstraintIndex[i] << "], status is "
    << problem.Solver.OptIIS.ConstraintStatus[i] <<
    Console.EndL();
  }
```

C#

```
//run an optimization
prob.Solver.Optimize();
Optimize_Status status = problem.Solver.OptimizeStatus;
//check status of optimization model.  If infeasible, call
//IISFind to create an Irreducibly Infeasible Subset (IIS) of
//the constraints and variable bounds
if (status == Optimize_Status.Unfeasible)
{
  //call IISFind to compute the Irreducibly Infeasible Subset
  //(IIS)
  prob.Solver.IISFind();
  //get the status of each variable bound
  for (int i = 0; i < prob.Solver.OptIIS.NumBounds; i++)
  MessageBox.Show("Bound[" +
  prob.Solver.OptIIS.BoundIndex[i] + "], status is "
  + prob.Solver.OptIIS.BoundStatus[i]);
  //get the status of each constraint
  for (int i = 0; i < prob.Solver.OptIIS.NumConstraints; i++)
    MessageBox.Show("Constraint[" +
    prob.Solver.OptIIS.ConstraintIndex[i] + "], status is "
    + prob.Solver.OptIIS.ConstraintStatus[i]);
}
```

VB.NET

```
'run an optimization
prob.Solver.Optimize()
'check status of optimization model.  If infeasible, call
'IISFind to create an Irreducibly Infeasible Subset (IIS) of
'the constraints and variable bounds
Dim status As Optimize_Status
status = prob.Solver.OptimizeStatus
If status = Optimize_Status.Unfeasible Then
'call IISFind to compute the Irreducibly Infeasible Subset
```

```
 '(IIS)
 prob.Solver.IISFind()
 'obtain the status of each variable bound
 For i = 0 To prob.Solver.OptIIS.NumBounds - 1
   MessageBox.Show("Bound[" & prob.Solver.OptIIS.BoundIndex(i) &
   _"], status is " & prob.Solver.OptIIS.BoundStatus(i))
 Next
 'obtain status of each constraint
 For i = 0 To prob.Solver.OptIIS.NumConstraints - 1
   MessageBox.Show("Constraint[" &
   prob.Solver.OptIIS.ConstraintIndex(i) & _"], status is " &
   prob.Solver.OptIIS.ConstraintStatus(i))
 Next
End If
```

VB6

```
'run an optimization
prob.Solver.Optimize
'check status of optimization model.  If infeasible, call
'IISFind to create an Irreducibly Infeasible Subset (IIS) of
'the constraints and variable bounds
Dim status As Optimize_Status
status = prob.Solver.OptimizeStatus
If status = Optimize_Status_Unfeasible Then
  'call IISFind to compute the Irreducibly Infeasible Subset
  '(IIS)
  prob.Solver.IISFind
  'obtain the status of each variable bound
  For i = 0 To prob.Solver.OptIIS.NumBounds - 1
    MsgBox("Bound[" & prob.Solver.OptIIS.BoundIndex(i) & _
    "], status is " & prob.Solver.OptIIS.BoundStatus(i))
  Next
 'obtain the status of each constraint
  For i = 0 To prob.Solver.OptIIS.NumConstraints - 1
    MsgBox("Bound[" & prob.Solver.OptIIS.ConstraintIndex(i) & _
    "], status is " & prob.Solver.OptIIS.ConstraintStatus(i))
  Next
End If
```

class CEngineStat

Class **CEngineStat()** implements the current status of an Engine object.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Milliseconds`	integer	get	Retrieves the total time in milliseconds used to perform optimization or simulation.
`Iterations`	integer	get	Returns the number of iterations the Engine has performed.
`Subproblems`	integer	get	Number of subproblems solved, when top-level algorithm[1] used; otherwise 1.
`LocalSolutions`	integer	get	Number of incumbents or locally optimal solutions found, when top-level algorithm used; otherwise 1.
`FunctionEvals`	integer	get	Retrieves the total number of function evalutions.
`JacobianEvals`	integer	get	Retrieves the total number of gradient evaluations.
`HessianEvals`	integer	get	Retrieves the total number of Hessian evaluations.
`Integer Gap`	integer	get	Retrieves the integer gap for the solved MIP.

Examples:

C++

```
//define variables
Double retSecs, retIter, retSubs, retSols;
//solve model
prob.Solver.Optimize();
//get the time taken to solve the optimization or simulation
retSecs = prob.Engine.Stat.Milliseconds;
```

[1] The top-level algorithm comprises the Branch and Bound algorithm and the Multistart methods.

```
//get the number of subproblems performed by the SDK
retSubs = prob.Engine.Stat.Subproblems;
//get the number of local solutions found by the SDK
retSols = prob.Engine.Stat.LocalSolutions;
//get the number of iterations performed by the SDK
retIter = prob.Engine.Stat.Iterations;
```

C#

```
//define variables
Double retSecs, retIter, retSubs, retSols;
//solve model
prob.Solver.Optimize();
//get the time taken to solve the optimization or simulation
retSecs = prob.Engine.Stat.Milliseconds;
//get the number of subproblems performed by the SDK
retSubs = prob.Engine.Stat.Subproblems;
//get the number of local solutions found by the SDK
retSols = prob.Engine.Stat.LocalSolutions;
//get the number of iterations performed by the SDK
retIter = prob.Engine.Stat.Iterations;
```

VB.NET

```
'define variables
Dim retSecs, retIter, retSubs, retSols as Double
'solve model
prob.Solver.Optimize()
'get the time taken to solve the optimization or simulation
retSecs = prob.Engine.Stat.Milliseconds
'get the number of subproblems performed by the SDK
retSubs = prob.Engine.Stat.Subproblems
'get the number of local solutions found by the SDK
retSols = prob.Engine.Stat.LocalSolutions
'get the number of iterations performed by the SDK
retIter = prob.Engine.Stat.Iterations
```

VB6

```
'define variables
Dim retSecs, retIter, retSubs, retSols as Double
'solve model
prob.Solver.Optimize()
'get the time taken to solve the optimization or simulation
retSecs = prob.Engine.Stat.Milliseconds
```

```
'get the number of subproblems performed by the SDK
retSubs = prob.Engine.Stat.Subproblems
'get the number of local solutions found by the SDK
retSols = prob.Engine.Stat.LocalSolutions
'get the number of iterations performed by the SDK
retIter = prob.Engine.Stat.Iterations
```

class CEngineLimit

Class **CEngineLimit()** returns the limits imposed by each Solver Engine according to the Problem_Type. (Currently, the Problem_Type argument is ignored so the engine will return it's limits regardless of what Problem Type is passed.)

Parameters:

Property Name	Property Type	Property Attributes	Description
VarUncertainLimit	integer	Get	Retrieves the maximum number of uncertain variables the current Engine supports.
FcnUncertainLimit	integer	Get	Retrieves the maximum number of uncertain functions the current Engine supports.
IterationLimit	integer	Get	Retrieves the maximum number of iterations the current Engine should perform.
CorrelationLimit	integer	Get	Retrieves the maximum number of correlations.
VarDecisionLimit	integer	Get	Retrieves the maximum number of decision variables the current Engine supports.
FcnConstraintLimit	integer	Get	Retrieves the maximum number of constraint functions the current Engine supports.
VarBoundLimit	integer	get	Retrieves the maximum number of bounds on the variables the current Engine supports.
VarIntegerLimit	integer	get	Retrieves the maximum number of integer variables the current Engine supports.

Examples:

C++ - Prints Engine Limits to C++ Console

```
//define variables
const wchar_t* engName;
int lpvars, lpconstraints, lpbounds, lpints;
int qpvars, qpconstraints, qpbounds, qpints;
```

```
int qcpvars, qcpconstraints, qcpbounds, qcpints;
int nlpvars, nlpconstraints, nlpbounds, nlpints;
int nspvars, nspconstraints, nspbounds, nspints;
int uncertainvars, uncertainfcns;

//set engine to LPQP engine (available only in SDK Platform)
//to solve a linear model. If using SDK Pro,
//use: [CEngine::LPName]
prob.Engine = prob.Engines[CEngine::LPQPName];
//print name of engine
engName = prob.Engine.Name;
wcout << engName << endl;
//get the decision variable limits:  lpvars = 8000
lpvars =
  prob.Engine.Limit[Problem_Type_OptLP].VarDecisionLimit;
//get the constraint limits:  lpconstraints = 8000
lpconstraints =
  prob.Engine.Limit[Problem_Type_OptLP].FcnConstraintLimit;
//get the variable bounds limit:  lpbounds = 16000
lpbounds =
  prob.Engine.Limit[Problem_Type_OptLP].VarBoundLimit;
//get the integer variable limits:  lpints = 2000
lpints =
  prob.Engine.Limit[Problem_Type_OptLP].VarIntegerLimit;
cout << "LP Limits: " << lpvars << " variables " <<
  lpconstraints << " constraints " << lpbounds << " bounds "
  << lpints << " integers" << endl;

//set engine to LPQP engine (available only in SDK Platform)
//to solve a quadratic model.  If using SDK Pro, use:
//[CEngine::GRGName]
prob.Engine = prob.Engines[CEngine::LPQPName];
//print name of engine
engName = prob.Engine.Name;
wcout << engName << endl;
//get the decision variable limits:  qpvars = 8000
qpvars =
  prob.Engine.Limit[Problem_Type_OptQP].VarDecisionLimit;
//get the constraint limits:  qpconstraints = 8000
qpconstraints =
  prob.Engine.Limit[Problem_Type_OptQP].FcnConstraintLimit;
//get the variable bounds limit:  qpbounds = 16000
qpbounds =
  prob.Engine.Limit[Problem_Type_OptQP].VarBoundLimit;
//get the integer variable limits:  qpints = 2000
qpints =
  prob.Engine.Limit[Problem_Type_OptQP].VarIntegerLimit;
cout << "QP Limits: " << qpvars << " variables " <<
      qpconstraints << " constraints " << qpbounds << " bounds "
      << qpints << " integers" << endl;

//set engine to SOCP engine (available only in SDK Platform)
//to solve a quadratically constrained or cone constrained model
//if using SDK Pro, use: [CEngine::GRGName]
prob.Engine = prob.Engines[CEngine::SOCPName];
```

```
//print name of engine
engName = prob.Engine.Name;
wcout << engName << endl;
//get the decision variable limits:  qcpvars = 2000
qcpvars =
  prob.Engine.Limit[Problem_Type_OptQCP].VarDecisionLimit;
//get the constraint limits:  qcpconstraints = 8000
qcpconstraints =
  prob.Engine.Limit[Problem_Type_OptQCP].FcnConstraintLimit;
//get the variable bounds limit:  qpbounds = 4000
qcpbounds =
  prob.Engine.Limit[Problem_Type_OptQCP].VarBoundLimit;
//get the integer variable limits:  qpints = 2000
qcpints =
  prob.Engine.Limit[Problem_Type_OptQCP].VarIntegerLimit;
cout << "QCP Limits: " << qcpvars << " variables " <<
  qcpconstraints << " constraints " << qcpbounds << " bounds "
  << qcpints << " integers" << endl;

//set engine to GRG Nonlinear engine in Solver SDK Pro
//or the Large Scale GRG Engine, in Solver SDK Platform
//to solve a nonlinear model.
prob.Engine = prob.Engines[CEngine::GRGName];
//print name of engine
engName = prob.Engine.Name;
wcout << engName << endl;
//get the decision variable limits:  qcpvars = 1000
nlpvars =
  prob.Engine.Limit[Problem_Type_OptNLP].VarDecisionLimit;
//get the constraint limits:  qcpconstraints = 1000
nlpconstraints =
  prob.Engine.Limit[Problem_Type_OptNLP].FcnConstraintLimit;
//get the variable bounds limit:  qpbounds = 2000
nlpbounds =
  prob.Engine.Limit[Problem_Type_OptNLP].VarBoundLimit;
//get the integer variable limits:  qpints = 1000
nlpints =
  prob.Engine.Limit[Problem_Type_OptNLP].VarIntegerLimit;
cout << "NLP Limits: " << nlpvars << " variables " <<
  nlpconstraints << " constraints " << nlpbounds << " bounds "
  << nlpints << " integers" << endl;

//set engine to Evolutionary in both Solver SDK Pro and Solver
//SDK Platform to solve a nonsmooth model.
prob.Engine = prob.Engines[CEngine::EVOName];
//print name of engine
engName = prob.Engine.Name;
wcout << engName << endl;
//get the decision variable limits:  qcpvars = 1000
nspvars =
  prob.Engine.Limit[Problem_Type_OptNSP].VarDecisionLimit;
//get the constraint limits:  qcpconstraints = 1000
nspconstraints =
  prob.Engine.Limit[Problem_Type_OptNSP].FcnConstraintLimit;
//get the variable bounds limit:  qpbounds = 2000
```

```
nspbounds =
  prob.Engine.Limit[Problem_Type_OptNSP].VarBoundLimit;
//get the integer variable limits:  qpints = 1000
nspints =
  prob.Engine.Limit[Problem_Type_OptNSP].VarIntegerLimit;
cout << "NSP Limits: " << nspvars << " variables " <<
  nspconstraints << " constraints " << nspbounds << " bounds "
  << nspints << " integers" << endl;

//set engine to Monte Carlo Simulation - available only in the
//Solver SDK Platform to run a simulation.
prob.Engine = prob.Engines[CEngine::MCSName];
//print name of engine
engName = prob.Engine.Name;
wcout << engName << endl;
//get the uncertain variable limit:  uncertainvars = 2000
uncertainvars =
  prob.Engine.Limit[Problem_Type_SimIndep].VarUncertainLimit;
//get the uncertain function limit:  uncertainfcns = 8000
uncertainfcns =
  prob.Engine.Limit[Problem_Type_SimIndep].FcnUncertainLimit;
cout << "Simulation Limits: " << uncertainvars << " uncertain
  variables " << uncertainfcns << " uncertain functions " <<
  endl;
```

C# - Prints Engine Limits to C# Message Boxes

```
//define variables
string engName;
int lpvars, lpconstraints, lpbounds, lpints;
int qpvars, qpconstraints, qpbounds, qpints;
int qcpvars, qcpconstraints, qcpbounds, qcpints;
int nlpvars, nlpconstraints, nlpbounds, nlpints;
int nspvars, nspconstraints, nspbounds, nspints;
int uncertainvars, uncertainfcns;

//set engine to LPQP engine (available only in SDK Platform)
//to solve a linear model. If using SDK Pro,
//use: [CEngine::LPName]
prob.Engine = prob.Engines[Engine.LPQPName];
//print name of engine
engName = prob.Engine.Name;
MessageBox.Show(engName);
//get the decision variable limits:  lpvars = 8000
lpvars =
  prob.Engine.Limit[Problem_Type.OptLP].VarDecisionLimit;
//get the constraint limits:  lpconstraints = 8000
lpconstraints =
  prob.Engine.Limit[Problem_Type.OptLP].FcnConstraintLimit;
//get the variable bounds limit:  lpbounds = 16000
lpbounds =
  prob.Engine.Limit[Problem_Type.OptLP].VarBoundLimit;
//get the integer variable limits:  lpints = 2000
lpints =
  prob.Engine.Limit[Problem_Type.OptLP].VarIntegerLimit;
```

```
MessageBox.Show("LP Limits: " + lpvars + " variables " +
lpconstraints + " constraints " + lpbounds + " bounds "   +
lpints + " integers");

//set engine to LPQP engine (available only in SDK Platform)
//to solve a quadratic model.  If using SDK Pro, use:
//[CEngine::GRGName]
prob.Engine = prob.Engines[Engine.LPQPName];
//print name of engine
engName = prob.Engine.Name;
MessageBox.Show(engName);
//get the decision variable limits:  qpvars = 8000
qpvars =
  prob.Engine.Limit[Problem_Type.OptQP].VarDecisionLimit;
//get the constraint limits:  qpconstraints = 8000
qpconstraints =
  prob.Engine.Limit[Problem_Type.OptQP].FcnConstraintLimit;
//get the variable bounds limit:  qpbounds = 16000
qpbounds =
  prob.Engine.Limit[Problem_Type.OptQP].VarBoundLimit;
//get the integer variable limits:  qpints = 2000
qpints =
  prob.Engine.Limit[Problem_Type.OptQP].VarIntegerLimit;
MessageBox.Show("QP Limits: " + qpvars + " variables " +
  qpconstraints + " constraints " + qpbounds + " bounds "
  + qpints + " integers");

//set engine to SOCP engine (available only in SDK Platform)
//to solve a quadratically constrained or cone constrained model
//If using SDK Pro, use: [CEngine::GRGName]
prob.Engine = prob.Engines[Engine.SOCPName];
//print name of engine
engName = prob.Engine.Name;
MessageBox.Show(engName);
//get the decision variable limits:  qcpvars = 2000
qcpvars =
  prob.Engine.Limit[Problem_Type.OptQCP].VarDecisionLimit;
//get the constraint limits:  qcpconstraints = 8000
qcpconstraints =
  prob.Engine.Limit[Problem_Type.OptQCP].FcnConstraintLimit;
//get the variable bounds limit:  qpbounds = 4000
qcpbounds =
  prob.Engine.Limit[Problem_Type.OptQCP].VarBoundLimit;
//get the integer variable limits:  qpints = 2000
qcpints =
  prob.Engine.Limit[Problem_Type.OptQCP].VarIntegerLimit;
MessageBox.Show("QCP Limits: " + qcpvars + " variables " +
  qcpconstraints + " constraints " + qcpbounds + " bounds "
  + qcpints + " integers");

//set engine to GRG Nonlinear engine in Solver SDK Pro
//or the Large Scale GRG Engine, in Solver SDK Platform
//to solve a nonlinear model.
prob.Engine = prob.Engines[Engine.GRGName];
//print name of engine
```

```
engName = prob.Engine.Name;
MessageBox.Show(engName);
//get the decision variable limits:  qcpvars = 1000
nlpvars =
  prob.Engine.Limit[Problem_Type.OptNLP].VarDecisionLimit;
//get the constraint limits:  qcpconstraints = 1000
nlpconstraints =
  prob.Engine.Limit[Problem_Type.OptNLP].FcnConstraintLimit;
//get the variable bounds limit:  qpbounds = 2000
nlpbounds =
  prob.Engine.Limit[Problem_Type.OptNLP].VarBoundLimit;
//get the integer variable limits:  qpints = 1000
nlpints =
  prob.Engine.Limit[Problem_Type.OptNLP].VarIntegerLimit;
MessageBox.Show("NLP Limits: " + nlpvars + " variables " +
  nlpconstraints + " constraints " + nlpbounds + " bounds "
  + nlpints + " integers");

//set engine to Evolutionary in both Solver SDK Pro and Solver
//SDK Platform to solve a nonsmooth problem
prob.Engine = prob.Engines[Engine.EVOName];
//print name of engine
engName = prob.Engine.Name;
MessageBox.Show(engName);
//get the decision variable limits:  qcpvars = 1000
nspvars =
  prob.Engine.Limit[Problem_Type.OptNSP].VarDecisionLimit;
//get the constraint limits:  qcpconstraints = 1000
nspconstraints =
  prob.Engine.Limit[Problem_Type.OptNSP].FcnConstraintLimit;
//get the variable bounds limit:  qpbounds = 2000
nspbounds =
  prob.Engine.Limit[Problem_Type.OptNSP].VarBoundLimit;
//get the integer variable limits:  qpints = 1000
nspints =
  prob.Engine.Limit[Problem_Type.OptNSP].VarIntegerLimit;
MessageBox.Show("NSP Limits: " + nspvars + " variables " +
  nspconstraints + " constraints " + nspbounds + " bounds "
  + nspints + " integers");

//set engine to Monte Carlo Simulation - available only in the
//Solver SDK Platform to run a simulation
prob.Engine = prob.Engines[Engine.MCSName];
//print name of engine
engName = prob.Engine.Name;
MessageBox.Show(engName);
//get the uncertain variable limit:  uncertainvars = 2000
uncertainvars =
  prob.Engine.Limit[Problem_Type.SimIndep].VarUncertainLimit;
//get the uncertain function limit:  uncertainfcns = 8000
uncertainfcns =
  prob.Engine.Limit[Problem_Type.SimIndep].FcnUncertainLimit;
MessageBox.Show("Simulation Limits: " + uncertainvars + "
uncertain variables " + uncertainfcns + " uncertain functions");
```

VB.NET - Prints Engine Limits to .NET Message Boxes

```
'define variables
Dim engName As String
Dim lpvars, lpconstraints, lpbounds, lpints As Integer
Dim qpvars, qpconstraints, qpbounds, qpints As Integer
Dim qcpvars, qcpconstraints, qcpbounds, qcpints As Integer
Dim nlpvars, nlpconstraints, nlpbounds, nlpints As Integer
Dim nspvars, nspconstraints, nspbounds, nspints As Integer
Dim uncertainvars, uncertainfcns As Integer

'Set engine to LPQP engine (available only in SDK Platform)
'to solve a linear model. If using SDK Pro,
'use: [CEngine::LPName]
prob.Engine = prob.Engines(Engine.LPQPName)
'print name of engine
engName = prob.Engine.Name
MessageBox.Show(engName)
'get the decision variable limits:  lpvars = 8000
lpvars = prob.Engine.Limit(Problem_Type.OptLP).VarDecisionLimit
'get the constraint limits:  lpconstraints = 8000
lpconstraints =
prob.Engine.Limit(Problem_Type.OptLP).FcnConstraintLimit
'get the variable bounds limit:  lpbounds = 16000
lpbounds = prob.Engine.Limit(Problem_Type.OptLP).VarBoundLimit
'get the integer variable limits:  lpints = 2000
lpints = prob.Engine.Limit(Problem_Type.OptLP).VarIntegerLimit
MessageBox.Show("LP Limits: " & lpvars & " variables " &
lpconstraints & " constraints " & lpbounds & " bounds " & lpints
& " integers")

'set engine to LPQP engine (available only in SDK Platform)
'to solve a quadratic model.  If using SDK Pro, use:
'[CEngine::GRGName]
prob.Engine = prob.Engines(Engine.LPQPName)
'print name of engine
engName = prob.Engine.Name
MessageBox.Show(engName)
'get the decision variable limits:  qpvars = 8000
qpvars = prob.Engine.Limit(Problem_Type.OptQP).VarDecisionLimit
'get the constraint limits:  qpconstraints = 8000
qpconstraints =
prob.Engine.Limit(Problem_Type.OptQP).FcnConstraintLimit
'get the variable bounds limit:  qpbounds = 16000
qpbounds = prob.Engine.Limit(Problem_Type.OptQP).VarBoundLimit
'get the integer variable limits:  qpints = 2000
qpints = prob.Engine.Limit(Problem_Type.OptQP).VarIntegerLimit
MessageBox.Show("QP Limits: " & qpvars & " variables " & _
qpconstraints & " constraints " & qpbounds & " bounds " & _
qpints & " integers")

'set engine to SOCP engine (available only in SDK Platform)
'to solve a quadratically constrained or cone constrained model
'if using SDK Pro, use: [CEngine::GRGName]
```

```
prob.Engine = prob.Engines(Engine.SOCPName)
'print name of engine
engName = prob.Engine.Name
MessageBox.Show(engName)
'get the decision variable limits:  qcpvars = 2000
qcpvars =
prob.Engine.Limit(Problem_Type.OptQCP).VarDecisionLimit
'get the constraint limits:  qcpconstraints = 8000
qcpconstraints =
prob.Engine.Limit(Problem_Type.OptQCP).FcnConstraintLimit
'get the variable bounds limit:  qpbounds = 4000
qcpbounds = prob.Engine.Limit(Problem_Type.OptQCP).VarBoundLimit
'get the integer variable limits:  qpints = 2000
qcpints = prob.Engine.Limit(Problem_Type.OptQCP).VarIntegerLimit
MessageBox.Show("QCP Limits: " & qcpvars & " variables " &
qcpconstraints & " constraints " & qcpbounds & " bounds " &
qcpints & " integers")

'set engine to GRG Nonlinear engine in Solver SDK Pro
'or the Large Scale GRG Engine, in Solver SDK Platform
'to solve a nonlinear model.
prob.Engine = prob.Engines(Engine.GRGName)
'print name of engine
engName = prob.Engine.Name
MessageBox.Show(engName)
'get the decision variable limits:  qcpvars = 1000
nlpvars =
prob.Engine.Limit(Problem_Type.OptNLP).VarDecisionLimit
'get the constraint limits:  qcpconstraints = 1000
nlpconstraints =
prob.Engine.Limit(Problem_Type.OptNLP).FcnConstraintLimit
'get the variable bounds limit:  qpbounds = 2000
nlpbounds = prob.Engine.Limit(Problem_Type.OptNLP).VarBoundLimit
'get the integer variable limits:  qpints = 1000
nlpints = prob.Engine.Limit(Problem_Type.OptNLP).VarIntegerLimit
MessageBox.Show("NLP Limits: " & nlpvars & " variables " & _
nlpconstraints & " constraints " & nlpbounds & " bounds " & _
nlpints & " integers")

'set engine to Evolutionary in both Solver SDK Pro and Solver
'SDK Platform to solve nonsmooth problem
prob.Engine = prob.Engines(Engine.EVOName)
'print name of engine
engName = prob.Engine.Name
MessageBox.Show(engName)
'get the decision variable limits:  qcpvars = 1000
nspvars =
prob.Engine.Limit(Problem_Type.OptNSP).VarDecisionLimit
'get the constraint limits:  qcpconstraints = 1000
nspconstraints =
prob.Engine.Limit(Problem_Type.OptNSP).FcnConstraintLimit
'get the variable bounds limit:  qpbounds = 2000
nspbounds = prob.Engine.Limit(Problem_Type.OptNSP).VarBoundLimit
'get the integer variable limits:  qpints = 1000
nspints = prob.Engine.Limit(Problem_Type.OptNSP).VarIntegerLimit
```

```
MessageBox.Show("NSP Limits: " & nspvars & " variables " & _
nspconstraints & " constraints " & nspbounds & " bounds " & _
nspints & " integers")

'set engine to Monte Carlo Simulation - available only in the
'Solver SDK Platform to run a simulation
prob.Engine = prob.Engines(Engine.MCSName)
'print name of engine
engName = prob.Engine.Name
MessageBox.Show(engName)
'get the uncertain variable limit:  uncertainvars = 2000
uncertainvars =
prob.Engine.Limit(Problem_Type.SimIndep).VarUncertainLimit
'get the uncertain function limit:  uncertainfcns = 8000
uncertainfcns =
prob.Engine.Limit(Problem_Type.SimIndep).FcnUncertainLimit
MessageBox.Show("Simulation Limits: " & uncertainvars & "
uncertain variables " & uncertainfcns & " uncertain functions")
```

VB 6 - Prints Engine Limits to VB6 Message Boxes

```
'define variables
Dim engName As String
Dim lpvars, lpconstraints, lpbounds, lpints As Integer
Dim qpvars, qpconstraints, qpbounds, qpints As Integer
Dim qcpvars, qcpconstraints, qcpbounds, qcpints As Integer
Dim nlpvars, nlpconstraints, nlpbounds, nlpints As Integer
Dim nspvars, nspconstraints, nspbounds, nspints As Integer
Dim uncertainvars, uncertainfcns As Integer

'Set engine to LPQP engine (available only in SDK Platform)
'to solve a linear model. If using SDK Pro,
'use: [CEngine::LPName]
prob.Engine = prob.Engines.Item("LP/Quadratic")
'print name of engine
engName = prob.Engine.Name
MsgBox (engName)
'get the decision variable limits:  lpvars = 8000
lpvars =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptLP).
  VarDecisionLimit
'get the constraint limits:  lpconstraints = 8000
lpconstraints =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptLP).
  FcnConstraintLimit
'get the variable bounds limit:  lpbounds = 16000
lpbounds =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptLP).
  VarBoundLimit
'get the integer variable limits:  lpints = 2000
lpints = prob.Engine.Limit(Problem_Type.Problem_Type_OptLP).
  VarIntegerLimit
MsgBox ("LP Limits: " & lpvars & " variables " & lpconstraints &
  " constraints " & lpbounds & " bounds " & lpints & "
  integers")
```

```
'set engine to LPQP engine (available only in SDK Platform)
'to solve a quadratic model.  If using SDK Pro, use:
'[CEngine::GRGName]
prob.Engine = prob.Engines.Item("LP/Quadratic")
'print name of engine
engName = prob.Engine.Name
MsgBox (engName)
'get the decision variable limits:  qpvars = 8000
qpvars =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptQP).
  VarDecisionLimit
'get the constraint limits:  qpconstraints = 8000
qpconstraints =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptQP).
  FcnConstraintLimit
'get the variable bounds limit:  qpbounds = 16000
qpbounds =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptQP).
  VarBoundLimit
'get the integer variable limits:  qpints = 2000
qpints = prob.Engine.Limit(Problem_Type.Problem_Type_OptQP).
  VarIntegerLimit
MsgBox ("QP Limits: " & qpvars & " variables " & _
  qpconstraints & " constraints " & qpbounds & " bounds " & _
  qpints & " integers")

'set engine to SOCP engine (available only in SDK Platform)
'to solve a quadratically constrained or cone constrained model
'if using SDK Pro, use: [CEngine::GRGName]
prob.Engine = prob.Engines.Item("SOCP Barrier")
'print name of engine
engName = prob.Engine.Name
MsgBox (engName)
'get the decision variable limits:  qcpvars = 2000
qcpvars =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptQCP).
  VarDecisionLimit
'get the constraint limits:  qcpconstraints = 8000
qcpconstraints =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptQCP).
  FcnConstraintLimit
'get the variable bounds limit:  qpbounds = 4000
qcpbounds =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptQCP).
  VarBoundLimit
'get the integer variable limits:  qpints = 2000
qcpints =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptQCP).
  VarIntegerLimit
MsgBox ("QCP Limits: " & qcpvars & " variables " &
  qcpconstraints & " constraints " & qcpbounds & " bounds " &
  qcpints & " integers")

'set engine to GRG Nonlinear engine in Solver SDK Pro
```

```
'or the Large Scale GRG Engine, in Solver SDK Platform
'to solve a nonlinear model.
prob.Engine = prob.Engines.Item("GRG Nonlinear")
'print name of engine
engName = prob.Engine.Name
MsgBox (engName)
'get the decision variable limits:  qcpvars = 1000
nlpvars =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptNLP).
  VarDecisionLimit
'get the constraint limits:  qcpconstraints = 1000
nlpconstraints =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptNLP).
  FcnConstraintLimit
'get the variable bounds limit:  qpbounds = 2000
nlpbounds =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptNLP).
  VarBoundLimit
'get the integer variable limits:  qpints = 1000
nlpints =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptNLP).
  VarIntegerLimit
MsgBox ("NLP Limits: " & nlpvars & " variables " & _
  nlpconstraints & " constraints " & nlpbounds & " bounds " & _
  nlpints & " integers")

'set engine to Evolutionary in both Solver SDK Pro and Solver
'SDK Platform to solve a nonsmooth problem
prob.Engine = prob.Engines.Item("Evolutionary")
'print name of engine
engName = prob.Engine.Name
MsgBox (engName)
'get the decision variable limits:  qcpvars = 1000
nspvars =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptNSP).
  VarDecisionLimit
'get the constraint limits:  qcpconstraints = 1000
nspconstraints =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptNSP).
  FcnConstraintLimit
'get the variable bounds limit:  qpbounds = 2000
nspbounds =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptNSP).
  VarBoundLimit
'get the integer variable limits:  qpints = 1000
nspints =
  prob.Engine.Limit(Problem_Type.Problem_Type_OptNSP).
  VarIntegerLimit
MsgBox ("NSP Limits: " & nspvars & " variables " & _
  nspconstraints & " constraints " & nspbounds & " bounds " & _
  nspints & " integers")

'set engine to Monte Carlo Simulation - available only in the
'Solver SDK Platform to run a simulation
prob.Engine = prob.Engines.Item("Monte Carlo Simulation")
```

```
'print name of engine
engName = prob.Engine.Name
MsgBox (engName)
'get the uncertain variable limit:  uncertainvars = 2000
uncertainvars =
  prob.Engine.Limit(Problem_Type.Problem_Type_SimIndep).
  VarUncertainLimit
'get the uncertain function limit:  uncertainfcns = 8000
uncertainfcns =
  prob.Engine.Limit(Problem_Type.Problem_Type_SimIndep).
  FcnUncertainLimit
MsgBox ("Simulation Limits: " & uncertainvars & " uncertain
  variables " & uncertainfcns & " uncertain functions")
```

class CEngineParam

Class **CEngineParam()** implements a single Engine parameter.

Usage:

C++	`prob.Model.Params[L"Parameter"].Default;`
C#	`prob.Model.Params["Parameter"].Default;`
VB .NET	`prob.Model.Params("Parameter").Default`
VB6	`Prob.Model.Params("Parameter").Default`
Description	Gets the Default setting for the parameter, "Parameter".

C++	`prob.Model.Params[0].Value;`
C#	`prob.Model.Params[0].Value;`
VB .NET	`prob.Model.Params(0).Value`
VB6	`prob.Model.Params(0).Value`
Description	Sets the value for the first model parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Name`	String	get	Retrieves the name of the Engine parameter.
`Value`	double	get/set	Gets/Sets the value of the parameter. The value must be within the minimum and the maximum.
`Default`	double	get	Retrieves the default value for the parameter.
`MaxValue`	double	get	Maximum value of the parameter `Value` property.
`MinValue`	double	get	Minimum value of the parameter `Value` property.

Common Engine Parameters:

Index	SDK Name	Default Value	Min Value	Max Value	Applicable Engine
0	`IISBounds`	0 – Include Bounds	0	1 – No Bounds	LP Simplex, LP/Quadratic, GRG Nonlinear, and SOCP
1	`Iterations`	2,147,483,647	0	2,147,483,647	All

2	MaxSubProblems	2,147,483,647	0	2,147,483,647	All Engines
3	MaxIntegerSols	2,147,483,647	0	2,147,483,647	All Engines
4	PopulationSize	0	0	200	GRG Nonlinear, LSGRG, Evolutionary
5	MaxTimeNoImp	30	0	2,147,483,647	Evolutionary
6	Scaling	0 – On	-1 – Off	1 – On	All Engines
8	UseDual	2 – Dual	1 – Primal	2	LP Simplex
9	SolveWithout	0 – Off	0	1 – On	All Engines
10	ProbingFeasibility	0 – Off	0	1 – On	LP Simplex
11	OptimalityFixing	0 – Off	0	1 – On	LP Simplex
13	BoundsImprovement	0 – Off	0	1 – On	LP Simplex
14	PrimalHeuristic	0 – Off	0	1 – On	LP Simplex
15	RequireBounds	1 – On	0	0 – Off	GRG Nonlinear, LSGRG, Evolutionary
17	LocalSearch	0	0	3	Evolutionary; 0 – Randomized Local, 1 – Deterministic Pattern, 2 – Gradient Local, 3 - Automatic
18	FixNonSmooth	0 – Off	0	1 – On	Evolutionary
19	RandomSeed	0	0	2,147,483,647	GRG Nonlinear, LSGRG, Evolutionary
20	MaxTime	0	0	2,147,483,647	All Engines
21	ReducedTol	1.0e-5	1.0e-9	1.0e-4	LP Simplex
22	PivotTol	1.0e-7	0	1	LP Simplex
23	SolutionTol	1.0e-5	1.0e-9	1.0e-4	LP Simplex
25	IntTolerance	0	0	1	All Engines
26	IntCutoff	2.0e+30	-2.0e+30	2.0e+30	All Engines
27	RecognizeLinear	0 – Do Not Recognize	0	1 – Recognize	GRG Nonlinear, LSGRG
28	Derivatives	3 – Jacobian	1 – Forward	4 – Jacobian with Error Checking	2 – Central; GRG Nonlinear, LSGRG
29	Estimates	0 – Tangent	0	1 – Quadratic	GRG Nonlinear, LSGRG

30	SearchOption	0 – Newton	0	1 – Conjugate	GRG Nonlinear, LSGRG
31	MultiStart	0 – Off	0	1 – On	GRG Nonlinear, LSGRG
32	TopoSearch	0 – Off	0	1 – On	GRG Nonlinear, LSGRG
33	Precision	1.0e-6	1.0e-9	1.0e-4	LP Simplex, GRG Nonlinear, Evolutionary
34	Convergence	1.0e-4	0	1	GRG Nonlinear, LSGRG, Evolutionary
36	MutationRate	0.075	0	1	Evolutionary
38	MaxKnapsackCuts	20	0	2,147,483,647	LP Simplex
39	KnapsackPasses	1	0	2,147,483,647	LP Simplex
40	MaxGomoryCuts	20	0	2,147,483,647	LP Simplex
41	GomoryPasses	1	0	2,147,483,647	LP Simplex
44	NumThreads	0 – Use Max	0	Number of cores on machine	All

LP/Quadratic Engine Parameters:

Index	SDK Name	Default Value	Min Value	Max Value	Settings
28	Derivatives	1	1	2	1 – Forward, 2 - Central
45	PrimalTolerance	1.0e-7	0	1	
46	DualTolerance	2,147,483,647	0	2,147,483,647	
50	Presolve	1	0	1	0 – Off, 1 - On
51	PreProcessing	1	1	3	1 – Automatic, 2 - None, 3 - Aggressive
52	Cuts	1	1	3	1 – Automatic, 2 – None, 3 0 Aggressive
53	Heuristics	1	1	3	1 – Automatic, 2 – None, 3 - Aggressive

GRG Nonlinear Engine Parameters:

Index	SDK Name	Default Value	Min Value	Max Value	Settings
48	RelaxBounds	0	0	1	0 – Do Not Relax, 1 - Relax

SOCP Engine Parameters:

Index	SDK Name	Default Value	Min Value	Max Value	Settings
45	GapTolerance	1.0e-6	0	1	
46	StepSizeFactor	.99	0	1	
47	FeasibilityTolerance	1.0e-6	0	1	
51	SearchDirection	3	0	3	0-Power Class, 1-Power Class with Predictor-Corrector, 2-Dual Scaling, and 3- Dual Scaling with Predictor-Corrector.
52	PowerIndex	1	0	2,147,483,647	

class CDistribution

Class **CDistribution** implements two functionalities: fitting a distribution (type and parameters) to sample data (1) and, inversely, constructing a distribution of known type and parameters (obtaining analytical moments, computing PDF, CDF etc., sampling data) (2).

Syntax:

```
CDistribution()
```

Parameters:

Property Name	Property Type	Property Attributes	Description
ArgsValid	boolean	get	Checks the validity of the arguments.
Problem	object	get	Gets Problem associated to this Distribution.
Distribution parameters			
ControlParams	double	get/set	Gets/Sets the control parameters (shift, lower cut, and upper cut).
DistType	distribution_type	get/set	Gets/Sets the distribution type (Normal, Weibull etc.).
Name	string	get	Retrieves the name of the distribution.
NumControlParams	integer	get	Retrieves the number of control parameters.
NumParams	integer	get/set	Gets/Sets the number of parameters.
Params	double	get	Retrieves the distribution parameters (location, shape, scale etc.).
Distribution analytic moments parameters			
Kurtosis	double	get	Retrieves the distribution kurtosis value.
LowerBound	double	get	Retrieves the lower bound.

Mean	double	get	Retrieves the distribution mean value.
Median	double	get	Retrieves the distribution median value.
Mode	double	get	Retrieves the distribution mode value.
Skewness	double	get	Retrieves the distribution skewness value.
UpperBound	double	get	Retrieves the upper bound.
Variance	double	get	Retrieves the distribution variance value.
Parameters of the Data to Fit			
Size	long	get	Retrieves the sample size.
Sample	double	get	Retrieves the sample one-dimensional array.
Parameters for Fitting Criteria			
FitLimit	double	get	Retrieves the goodness-of-fit test critical value.
FitStatistic	double	get	Retrieves the fitting statistic indicating the goodness-of-fit for Fit and AutoFit methods: Kolmogorov-Smirnof for continuous and Chi-squared for discrete distributions.
FitSuccess	boolean	get	True if distribution fit succeeded.
FitTest	double	get	Retrieves the fitting statistic independent on the sample's size. Generally FitSuccess is true if FitTest is less than FitLimit.
IndepTest	double	get	Retrieves a value indicating the appropriateness of the fitting (IndepTest). Fitting is appropriate if values to fit (Sample) are independent (uncorrelated). Small number of IndepTest indicates independent values.
SigLevel	double	get	Retrieves the fitting significance level, usually number between 0.9 and 0.99. In literature, it is known as "1 – α".

Methods:

Method Name	Description
AutoFit	Automatically selects the best fit distribution among qualifying candidates. Accepts three arguments: Type, Level_of_Significance, and FitAll. If `bFitAll = true` (see **Methods Usage)**, the best fit among all available distributions will be selected .
AutoFitAll	Returns the top 20 fitting distributions, includes three aguments: Type, SortType, and Allow_Shifted_Distributions.
CDF	Computes the cumulative distribution.
CDFInv	Computes the inverse of the cumulative distribution.
Fit	Fits the sample to the desired Type. If `bShift = true` (see **Methods Usage)**, the one-dimensional optimization will shift the data to obtain better fitting. **Note:** This method cannot fit continuous data to discrete distribution.
Init	**For use with COM API only** - The Init method accepts a string argument containing a PSI Distribution function call such as "PsiNormal(0,1)".
InitSample	Initializes the sample. Initialization creates one linear array as in `Sample` and `Size` properties as follows: • if `Occurs = NULL`, `Size` and `Sample` are the same as in `getSample`, `getSize` (see **Methods Usage)** • if `Sample = Occurs = NULL`, initialization generates random `Sample` from the current `DistType`
PDF	Computes the probability density function.

Methods Properties:

Method `AutoFit`

Property Name	Property Type	Description
bContinuous	Boolean	Indicates whether the data is continuous (= `true`) or discrete (=`false`).
SigLevel	double	Level of statistical significance of the autofit outcome to the resulting `Type` of distribution.
bFitAll	Boolean	*Optional* When FitAll is set to **True**, the default, the SDK internally tries to fit all continuous or discrete distributions (depending on the bContinuous

		argument) to the data. When FitAll is set to **False**, the SDK will internally restrict the distributions that are fit to the data depending on the computed skewness, median, mean, and standard deviation values for the data. **Only in very rare cases, should this argument be set to False.**

Method `AutoFitAll`

Property Name	Property Type	Description
`bContinuous`	Boolean	Indicates whether the data is continuous (= `true`) or discrete.
`SortType`	_Sort_Type	For continuous data, you can choose to rank the fitted distributions by the Chi Square (DistFit_SortCS), Kolmogorov-Smirnoff (DistFit_SoftKS) or Anderson-Darling (DistFit_SortAD) statistic. For discrete data, only the Chi Square statistic is meaningful.
`bShift`	Boolean	If true, the SDK will shift the center of analytic distribution to fit the sample data. False will disable the shift.

Method `PDF`

Property Name	Property Type	Description
`X`	double	Probability density value.

Method `CDF`

Property Name	Property Type	Description
`X`	double	Cumulative distribution value.

Method `CDFInv`

Property Name	Property Type	Description
`F`	double	Inversed cumulative distribution value.

Method `InitSample`

Property Name	Property Type	Description
`Size`	integer	Length of `Sample` and `Occurs`.

Sample	double	Sample one-dimensional array. if Sample = Occurs = NULL, initialization generates random Sample from the current DistType.
Occurs	integer	If NULL, Size and Sample are the same as in getSample and getSize.

Method Fit

Property Name	Property Type	Description
Type	enum	Distribution type to which the sample is fit. For more information, refer to enum DistFit_Type.
SigLevel	double	Level of statistical significance of the fit outcome to the specified Type of distribution.
bShift	Boolean	If true, the data will be shifted for better fitting by one-dimensional optimization.

Methods Usage:

C++ - AutoFit & InitSample

```
#include "C:/Program Files/Frontline Systems/Solver SDK
platform/Include/SolverSdk.h"

try
  {
   //create "sample" array of size 50 to hold sample values
   double sample[] = {-3.44,0.69,-0.11,1.25,0.73,-1.66,1.57,
                      -1.64,-0.51,-0.19,1.24,-0.96,1.40,-0.01,
                      -0.77,0.91,-0.72,0.85,-0.20,0.59,1.65,
                       1.40,0.19,-1.38,-0.45,-0.71,1.25,-1.18,
                      -0.33,2.33,0.73,-0.75,-0.63,-1.06,
                      -0.71,0.33,-0.82,0.43,0.10,0.88,0.11,
                      -0.88,0.02,-0.94,0.05,-0.06,-1.87,-2.03,
                      -0.89,1.14};
  //create new simulation problem
  CProblem problem(Solver_Type_Simulate, 1, 0);
  //create new distribution
  CDistribution dist;
  //pass "sample" array holding 50 sample values
  dist.InitSample(50, sample);
  //call AutoFit to automatically fit the best distribution
  dist.AutoFit(true, 0.95);
  //determine if AutoFit was successful
    if (dist.FitSuccess)
    {
     //if fit succesful, display "fit was 'successful' and
     //various fit stats
     Console << "Fit succeeded" <<Console.EndL();
     Console << "Distribution Type = " <<
```

```
      dist.Name << Console.EndL();
    for (int i=0; i< dist.get_NumParams();i++)
    Console << "Param[" << i << "] = " << dist.Params[i] <<
      Console.EndL();
   }
    else
   {
     //if not successful, indicate as such
     Console << "Fit unsuccesful" <<Console.EndL();
        return 1;
    }
```

C++ - AutoFitAll & InitSample

```
#include "C:/Program Files/Frontline Systems/Solver SDK
platform/Include/SolverSdk.h"

try
  {
   //create "sample" array of size 50 to hold sample values
   double sample[] = {-3.44,0.69,-0.11,1.25,0.73,-1.66,1.57,
                      -1.64,-0.51,-0.19,1.24,-0.96,1.40,-0.01,
                      -0.77,0.91,-0.72,0.85,-0.20,0.59,1.65,
                       1.40,0.19,-1.38,-0.45,-0.71,1.25,-1.18,
                      -0.33,2.33,0.73,-0.75,-0.63,-1.06,
                      -0.71,0.33,-0.82,0.43,0.10,0.88,0.11,
                      -0.88,0.02,-0.94,0.05,-0.06,-1.87,-2.03,
                      -0.89,1.14};
  //create new simulation problem
  CProblem problem(Solver_Type_Simulate, 1, 0);
  //create new distribution
  CDistribution dist;
  //pass "sample" array holding 50 sample values
  dist.InitSample(50, sample);
  //call AutoFitAll to return a collection of 20 of the most
  //closely fitted distributions
  dist.AutoFitAll (true, DistFit_SortKS,true);
  //variable to hold the collection of distributions returned by
  //AutoFitAll
  CDistributionCollection disttest;
  //display the name and FitTest value for the top 21
  //distributions.  For a complete list of distributions
  //included in the SDK, see enum Distribution_Type.
  for (int i = 0; i < 21; i++)
    Console << "Distribution: " << disttest[i].Name <<
    "          FitTest Statistic: " << disttest[i].FitTest <<
    Console.EndL();
  }
```

C++ - Fit & InitSample

```
#include "C:/Program Files/Frontline Systems/Solver SDK
platform/Include/SolverSdk.h"

try
  {
   //create "sample" array of size 50 to hold sample values
```

```
   double sample[] = {-3.44,0.69,-0.11,1.25,0.73,-1.66,1.57,
                      -1.64,-0.51,-0.19,1.24,-0.96,1.40,-0.01,
                      -0.77,0.91,-0.72,0.85,-0.20,0.59,1.65,
                       1.40,0.19,-1.38,-0.45,-0.71,1.25,-1.18,
                      -0.33,2.33,0.73,-0.75,-0.63,-1.06,
                      -0.71,0.33,-0.82,0.43,0.10,0.88,0.11,
                      -0.88,0.02,-0.94,0.05,-0.06,-1.87,-2.03,
                      -0.89,1.14};
  //create new simulation problem
  CProblem problem(Solver_Type_Simulate, 1, 0);
  //create new distribution
  CDistribution dist;
  //pass "sample" array holding 50 sample values
  dist.InitSample(50, sample);
  //call Fit to fit the sample values to a specific distribution
  //in this example, the Uniform distribution
  dist.Fit (DistFit_Uniform, 0.95, True);
  }
```

C++ - PDF, CDF, & CDFInv

```
#include "C:/Program Files/Frontline Systems/Solver SDK
platform/Include/SolverSdk.h"
try
 {
  //create new simulation problem with 1 uncertain variable and
  //0 uncertain functions
  int nvars = 1, nfcns = 0;
  CProblem problem(Solver_Type_Simulate, nvars, nfcns);
  //create new PsiNormal distribution
  CDistribution dist;
  problem.Model.VarDistribution[0] =
  CDistribution(L"PsiNormal(1,1)");
  dist = problem.Model.VarDistribution[0];
  //perform a simulation
  problem.Solver.Simulate();
  //create 5 "bins" and pass values of -2, -1, 0, 1 & 2
  int k = 0;
  double x[5];
  for (int i =0;i < 5; i++)
  {
    x[i] = -2 + k;
    k = k +1;
  }
  //obtain CDF, PDF, and CDFInv values
  double cdfvalue[5];
  double pdfvalue[5];
  double cdfinv[5];
  for (int i = 0; i <5 ; i++)
  {
   cdfvalue[i] = dist.CDF(x[i]);
   pdfvalue[i] = dist.PDF(x[i]);
   cdfinv[i] = dist.CDFInv(x[i]);
   Console << "i = " << i << " CDF: " << dist.CDF(x[i]) << ",
   PDF: " << dist.PDF(x[i]) << ", CDFInv: " <<
   dist.CDFInv(x[i])<<Console.EndL();
  }
```

```
 }
```

C## – AutoFit & InitSample

```
using SolverPlatform;

try
 {
  //create "sample" array of size 50 to hold sample values
  double[] sample = new double[] {-3.44,0.69,-0.11,1.25,0.73,-
                                   1.66,1.57,-1.64,-0.51,-0.19,
                                   1.24,-0.96,1.40,-0.01,
                                  -0.77,0.91,-0.72,0.85,
                                  -0.20,0.59,1.65,1.40,0.19,
                                  -1.38,-0.45,-0.71,1.25,-1.18,
                                  -0.33,2.33,0.73,-0.75,-0.63,
                                  -1.06,-0.71,0.33,
                                   0.82,0.43,0.10,0.88,0.11,
                                  -0.88,0.02,-0.94,0.05,-0.06,
                                  -1.87,-2.03,-0.89,1.14};

//create new problem with 1 uncertain variable and 0 uncertain
//functions
using (Problem problem = new Problem(Solver_Type.Simulate, 1,
0))
//create new distribution
Distribution dist = new Distribution();
//pass "sample" array holding 50 sample values
dist.InitSample(sample);
//call AutoFit to automatically fit the best distribution
dist.AutoFit(true, 0.95, true);
//determine if AutoFit was successful or not
if (dist.FitSuccess)
 {
  //if AutoFit is successful, display "fit was 'successful' and
  //various fit statistics.
  MessageBox.Show("Distribution Type = " + dist.Name);
  for (int i = 0; i < dist.NumParams; i++)
  MessageBox.Show("Param[" + i + "] = " + dist.Params[i])
  }
else
  {
   //if not successful, indicate as such
   MessageBox.Show("Fit unsuccesful");
    return;
   }
}
```

C## – AutoFitAll & InitSample

```
using SolverPlatform;

try
 {
  //create "sample" array of size 50 to hold sample values
  double[] sample = new double[] {-3.44,0.69,-0.11,1.25,0.73,-
                                   1.66,1.57,-1.64,-0.51,-0.19,
```

```
                                        1.24,-0.96,1.40,-0.01,
                                       -0.77,0.91,-0.72,0.85,
                                       -0.20,0.59,1.65,1.40,0.19,
                                       -1.38,-0.45,-0.71,1.25,-1.18,
                                       -0.33,2.33,0.73,-0.75,-0.63,
                                       -1.06,-0.71,0.33,
                                        0.82,0.43,0.10,0.88,0.11,
                                       -0.88,0.02,-0.94,0.05,-0.06,
                                       -1.87,-2.03,-0.89,1.14};

//create new problem with 1 uncertain variable and 0 uncertain
//functions
using (Problem problem = new Problem(Solver_Type.Simulate, 1,
0))
//create new distribution
Distribution dist = new Distribution();
//pass "sample" array holding 50 sample values
dist.InitSample(sample);
//call AutoFitAll to return a collection of 21 of the most
//closely fitted distributions
dist.AutoFitAll(True, DistFit_Sort.KS, True);
//display the distribution name and Fit Test statistic for all
//21 distributions.  For a complete list of distributions
//included in the SDK, see enum Distribution_Type.
DistributionCollection disttest = new DistributionCollection();
for (int i = 0; i < 21; i++)
 MessageBox.Show("Distribution: " + disttest[i].Name +
    "          FitTest Statistic: " + disttest[i].FitTest);
 }
```

C## – Fit & InitSample

```
using SolverPlatform;

try
 {
  //create "sample" array of size 50 to hold sample values
  double[] sample = new double[] {-3.44,0.69,-0.11,1.25,0.73,-
                                   1.66,1.57,-1.64,-0.51,-0.19,
                                   1.24,-0.96,1.40,-0.01,
                                  -0.77,0.91,-0.72,0.85,
                                  -0.20,0.59,1.65,1.40,0.19,
                                  -1.38,-0.45,-0.71,1.25,-1.18,
                                  -0.33,2.33,0.73,-0.75,-0.63,
                                  -1.06,-0.71,0.33,
                                   0.82,0.43,0.10,0.88,0.11,
                                  -0.88,0.02,-0.94,0.05,-0.06,
                                  -1.87,-2.03,-0.89,1.14};

//create new problem with 1 uncertain variable and 0 uncertain
//functions
using (Problem problem = new Problem(Solver_Type.Simulate, 1,
0))
//create new distribution
Distribution dist = new Distribution();
//pass "sample" array holding 50 sample values
```

```
dist.InitSample(sample);
//call Fit to fit the sample values to a specific distribution
//in this example, the Uniform distribution
dist.Fit(DistFit_Uniform, 0.95, True);
}
```

C## – PDF, CDF & CDFInv

```
using SolverPlatform;

try
 {
  //create new problem with 1 uncertain variable and 0 uncertain
  //functions
  using (Problem problem = new Problem(Solver_Type.Simulate,
  nvars, nfcns))
  //create 1 normal distribution
  Distribution distnormal = new Distribution();
  problem.Model.VarDistribution[0] = new
    Distribution("PsiNormal(1,1)");
  distnormal = problem.Model.VarDistribution[0];
  //run a simulation
  problem.Solver.Simulate();
  //create 5 "bins" and pass values of -2, -1, 0, 1, & 2
  double[] x = new double[5];
  int k = 0;
  for (int i = 0; i < 5; i++)
  {
   x[i] = -2 + k;
   k++;
  }
  double[] cdfvalue = new double[5];
  double[] pdfvalue = new double[5];
  double[] cdfinv = new double[5];
  for (int i = 0; i < 5; i++)
  {
   cdfvalue[i] = distnormal.CDF(x[i]);
   MessageBox.Show("CDFValue: " + cdfvalue[i]);
  }
  for (int i = 0; i < 5; i++)
  {
   pdfvalue[i] = distnormal.PDF(x[i]);
   MessageBox.Show("PDFValue: " + pdfvalue[i]);
  }
  for (int i = 0; i < 5; i++)
  {
   cdfinv[i] = distnormal.CDFInv(x[i]);
   MessageBox.Show("CDFInv: " + cdfinv[i]);
  }
}
```

VB .NET – AutoFit & InitSample

```
'create "sample" array of size 50 to hold sample values
Dim sample() As Double = New Double() {-3.44, 0.69, -0.11, _
0.73, -1.66, 1.57, -1.64, -0.51, -0.19, 1.24, -0.96, 1.4, _
-0.01, -0.77, 0.91, -0.72, 0.85, -0.2, 0.59, 1.65, 1.4, _
```

```
0.19, -1.38, -0.45, -0.71, 1.25, -1.18, -0.33, 2.33, _
0.73, -0.75, -0.63, -1.06, -0.71, 0.33, -0.82, 0.43, 0.1, _
0.88, 0.11, -0.88, 0.02, -0.94, 0.05, -0.06, -1.87, -2.03, _
-0.89, 1.14}

'create new Simulation Problem
Dim problem As New SolverPlatform.Problem
problem.Solver.SolverType = Solver_Type.Simulate
'create a new distribution
Dim dist As New Distribution
'pass "sample" array holding 50 sample values
dist.InitSample(sample)
'call AutoFit to automatically fit the best distribution
dist.AutoFit(True, 0.95, True)
Dim i As Integer
'determine if AutoFit was successful or not
If dist.FitSuccess Then
  MsgBox ("Fit was succesful. Distribution Type = " & dist.Name
  For i = 0 To dist.NumParams() - 1
     MsgBox("Param[" & i & "] = " & dist.Params(i) & vbCrLf)
  Next
Else
  'if not successful, indicate as such
  MsgBox ("Fit was not succesful.")
If Not problem Is Nothing Then
  problem.Dispose()
  End If
  Exit Sub
End If
```

VB .NET – AutoFitAll & InitSample

```
'create "sample" array of size 50 to hold sample values
Dim sample() As Double = New Double() {-3.44, 0.69, -0.11, _
0.73, -1.66, 1.57, -1.64, -0.51, -0.19, 1.24, -0.96, 1.4, _
-0.01, -0.77, 0.91, -0.72, 0.85, -0.2, 0.59, 1.65, 1.4, _
0.19, -1.38, -0.45, -0.71, 1.25, -1.18, -0.33, 2.33, _
0.73, -0.75, -0.63, -1.06, -0.71, 0.33, -0.82, 0.43, 0.1, _
0.88, 0.11, -0.88, 0.02, -0.94, 0.05, -0.06, -1.87, -2.03, _
-0.89, 1.14}

Dim problem As SolverPlatform.Problem = Nothing
Try
 'create new Simulation Problem
 Dim problem As New SolverPlatform.Problem
 problem.Solver.SolverType = Solver_Type.Simulate
 'create a new distribution
 Dim dist As New Distribution
  'pass "sample" array holding 50 sample values
 dist.InitSample(sample)
 'call AutoFitAll to return a collection of 21 of the most
 'closely fitted distributions
 dist.AutoFitAll(True, DistFit_SortType.KS, True)
 Dim disttest as New DistributionCollection
 For i = 0 to 20
 'display the distribution name and Fit Test statistic for the
```

```
'first 21 distributions.  For a complete list of
'distributions included in the SDK, see enum Distribution_Type.
'create a new DistributionCollection
     MsgBox "Distribution: " & disttest(i).Name & "    FitTest
     Statistic: " & disttest(i).FitTest
Next
```

VB .NET – Fit & InitSample

```
'create "sample" array of size 50 to hold sample values
Dim sample() As Double = New Double() {-3.44, 0.69, -0.11, _
0.73, -1.66, 1.57, -1.64, -0.51, -0.19, 1.24, -0.96, 1.4, _
-0.01, -0.77, 0.91, -0.72, 0.85, -0.2, 0.59, 1.65, 1.4, _
0.19, -1.38, -0.45, -0.71, 1.25, -1.18, -0.33, 2.33, _
0.73, -0.75, -0.63, -1.06, -0.71, 0.33, -0.82, 0.43, 0.1, _
0.88, 0.11, -0.88, 0.02, -0.94, 0.05, -0.06, -1.87, -2.03, _
-0.89, 1.14}

'create new Simulation Problem
Dim problem As New SolverPlatform.Problem
problem.Solver.SolverType = Solver_Type.Simulate
'create a new distribution
Dim dist As New Distribution
'pass "sample" array holding 50 sample values
dist.InitSample(sample)
//call Fit to fit the sample values to a specific distribution
//in this example, the Uniform distribution
dist.Fit(DistFit_Type.Uniform, 0.95, True)
```

VB .NET – PDF, CDF, & CDFInv

```
'create new Simulation Problem, problem, with 1 uncertain
'variable and 0 uncertain functions
problem = New Problem(Solver_Type.Simulate, 1, 0)
'create new PsiNormal distribution
Dim distnormal As New Distribution
problem.Model.VarDistribution(0) = New
Distribution("PsiNormal(1,1)")
distnormal = problem.Model.VarDistribution(0)
'run a simulation
problem.Solver.Simulate()
'create 5 "bins" and pass values of -2, -1, 0, 1 & 2
Dim x(0 To 4) As Double
Dim i, k As Integer
For i = 0 To 4
 x(i) = -2 + k
 k = k + 1
Next
'obtain CDF, PDF, and CDFInv values
Dim cdfvalue(0 To 4) As Double
Dim pdfvalue(0 To 4) As Doub
Dim cdfinv(0 To 4) As Double
For i = 0 To 4
 cdfvalue(i) = distnormal.CDF(x(i))
 pdfvalue(i) = distnormal.PDF(x(i))
 cdfinv(i) = distnormal.CDFInv(x(i))
```

```
 MsgBox ("CDF: " & cdfvalue(i) & ", PDF: " &
 pdfvalue(i) & ", CDFInv" & cdfinv(i))
Next
```

VB 6 – AutoFit & InitSample

```
'create "sample" array of size 50 to hold sample values
Dim sample(49) As Double
sample(0) = -3.44:   sample(1) = 0.69:   sample(2) = -0.11
sample(3) = 1.25:    sample(4) = 0.73:   sample(5) = -1.66
sample(6) = 1.57:    sample(7) = -1.64:   sample(8) = -0.51
sample(9) = -0.19:   sample(10) = 1.24:  sample(11) = -0.96
sample(12) = 1.4:    sample(13) = -0.01:    sample(14) = -0.77
sample(15) = 0.91:   sample(16) = -0.72:    sample(17) = 0.85
sample(18) = -0.2:   sample(19) = 0.59:     sample(20) = 1.65
sample(21) = 1.4:    sample(22) = 0.19:     sample(23) = -1.38
sample(24) = -0.45:  sample(25) = -0.71:    sample(26) = 1.25
sample(27) = -1.18:  sample(28) = -0.33:    sample(29) = 2.33
sample(30) = 0.73:   sample(31) = -0.75:    sample(32) = -0.63
sample(33) = -1.06:  sample(34) = -0.71:    sample(35) = 0.33
sample(36) = -0.82:  sample(37) = 0.43:     sample(38) = 0.1
sample(39) = 0.88:   sample(40) = 0.11:     sample(41) = -0.88
sample(42) = 0.02:   sample(43) = -0.94:    sample(44) = 0.05
sample(45) = -0.06:  sample(46) = -1.87:    sample(47) = -2.03
sample(48) = -0.89:  sample(49) = 1.14
'create new problem
Dim prob As SolverPlatform.Problem
Set prob = New SolverPlatform.Problem
'set problem type to Solver_Type_Simulate
prob.Solver.SolverType = Solver_Type_Simulate
'create new distribution
Dim dist As New Distribution
'pass "sample" array holding 50 sample values
dist.InitSample sample
'call AutoFit to automatically fit the best distribution
dist.AutoFit True, 0.95, True
Dim i As Integer
    'determine if AutoFit was successful or not
    If dist.FitSuccess = True Then
       'if fit successful, display "fit was 'successful' and
       'various fit statistics
       Msgbox "Fit was successful."
```

```
        Msgbox "Distribution Type = " & dist.Name
        MsgBox "FitStatistic = " & dist.FitStatistic
        MsgBox "FitLimit = " & dist.FitLimit
        MsgBox "FitTest = " & dist.FitTest
        MsgBox "IndepTest = " & dist.IndepTest
        For i = 0 To dist.NumParams() - 1
             Msgbox "Param[" & i & "] = " & dist.Params(i)
        Next
    Else
        'if not successful, indicate as such
        Msgbox "Fit was not successful."
        Set prob = Nothing
        Exit Sub
    End If
```

VB 6 – AutoFitAll & InitSample

```
 'create "sample" array of size 50 to hold sample values
Dim sample(49) As Double
sample(0) = -3.44:   sample(1) = 0.69:   sample(2) = -0.11
sample(3) = 1.25:    sample(4) = 0.73:   sample(5) = -1.66
sample(6) = 1.57:    sample(7) = -1.64:   sample(8) = -0.51
sample(9) = -0.19:   sample(10) = 1.24:  sample(11) = -0.96
sample(12) = 1.4:    sample(13) = -0.01:    sample(14) = -0.77
sample(15) = 0.91:   sample(16) = -0.72:    sample(17) = 0.85
sample(18) = -0.2:   sample(19) = 0.59:     sample(20) = 1.65
sample(21) = 1.4:    sample(22) = 0.19:     sample(23) = -1.38
sample(24) = -0.45:  sample(25) = -0.71:    sample(26) = 1.25
sample(27) = -1.18:  sample(28) = -0.33:    sample(29) = 2.33
sample(30) = 0.73:   sample(31) = -0.75:    sample(32) = -0.63
sample(33) = -1.06:  sample(34) = -0.71:    sample(35) = 0.33
sample(36) = -0.82:  sample(37) = 0.43:     sample(38) = 0.1
sample(39) = 0.88:   sample(40) = 0.11:     sample(41) = -0.88
sample(42) = 0.02:   sample(43) = -0.94:    sample(44) = 0.05
sample(45) = -0.06:  sample(46) = -1.87:    sample(47) = -2.03
sample(48) = -0.89:  sample(49) = 1.14
'create new problem
Dim prob As SolverPlatform.Problem
Set prob = New SolverPlatform.Problem
'set problem type to Solver_Type_Simulate
prob.Solver.SolverType = Solver_Type_Simulate
```

```
'create new distribution
Dim dist As New Distribution
'pass "sample" array holding 50 sample values
dist.InitSample sample
'call AutoFitAll to return a collection of 20 of the most
'closely fitted distributions
dist.AutoFitAll True, DistFit_SortKS, True
'display the distribution name and Fit Test statistic for the
'first 21 distributions. Use test(x).DistType to return the
'distribution's corresponding value.  For a complete list of
'Distributions included in the SDK, see enum Distribution_Type.
Dim disttest As DistributionCollection
Dim i As Integer
For i = 0 to 20
      MsgBox "Distribution: " & disttest(i).Name & "     FitTest
      Statistic: " & disttest(i).FitTest
Next
```

VB 6 – Fit & InitSample

```
'create "sample" array of size 50 to hold sample values
Dim sample(49) As Double
sample(0) = -3.44:   sample(1) = 0.69:   sample(2) = -0.11
sample(3) = 1.25:    sample(4) = 0.73:   sample(5) = -1.66
sample(6) = 1.57:    sample(7) = -1.64:   sample(8) = -0.51
sample(9) = -0.19:   sample(10) = 1.24:  sample(11) = -0.96
sample(12) = 1.4:    sample(13) = -0.01:    sample(14) = -0.77
sample(15) = 0.91:   sample(16) = -0.72:    sample(17) = 0.85
sample(18) = -0.2:   sample(19) = 0.59:     sample(20) = 1.65
sample(21) = 1.4:    sample(22) = 0.19:     sample(23) = -1.38
sample(24) = -0.45:  sample(25) = -0.71:    sample(26) = 1.25
sample(27) = -1.18:  sample(28) = -0.33:    sample(29) = 2.33
sample(30) = 0.73:   sample(31) = -0.75:    sample(32) = -0.63
sample(33) = -1.06:  sample(34) = -0.71:    sample(35) = 0.33
sample(36) = -0.82:  sample(37) = 0.43:     sample(38) = 0.1
sample(39) = 0.88:   sample(40) = 0.11:     sample(41) = -0.88
sample(42) = 0.02:   sample(43) = -0.94:    sample(44) = 0.05
sample(45) = -0.06:  sample(46) = -1.87:    sample(47) = -2.03
sample(48) = -0.89:  sample(49) = 1.14
'create new problem
Dim prob As SolverPlatform.Problem
Set prob = New SolverPlatform.Problem
'set problem type to Solver_Type_Simulate
```

```
prob.Solver.SolverType = Solver_Type_Simulate
'create new distribution
Dim dist As New Distribution
'pass "sample" array holding 50 sample values
dist.InitSample sample
//call Fit to fit the sample values to a specific distribution
//in this example, the Uniform distribution
dist.Fit DistFit_Uniform, 0.95, True
```

VB 6 – Init – COM API only

```
'create new Normal Distribution
Dim dist1 As New SolverPlatform.Distribution
dist1.Init "PsiNormal(2000, 500)"
```

VB 6 – PDF, CDF, & CDFInv

```
'create new problem
Dim prob As SolverPlatform.Problem
Set prob = New SolverPlatform.Problem
'set problem type to Solver_Type_Simulate
prob.Solver.SolverType = Solver_Type_Simulate
'create 1 uncertain variables
prob.VarUncertain.Size = 1
'create 1 normal distribution
Dim distNormal As New Distribution
distNormal.Init "PsiNormal(1,1)"
'assign distributions to uncertain variables
prob.Model.VarDistribution(0) = distNormal
'perform a simulation
prob.Solver.Simulate
'create 6 "bins" and pass values of -2, -1, 0, 1, & 2
Dim x(0 To 6) As Double
Dim i,k as Integer
For i = 0 To 6
   x(i) = -2 + k
   k = k + 1
Next
'obtain CDF, PDF, and CDFInv values
Dim cdfvalue(0 To 6) As Double
Dim pdfvalue(0 To 6) As Double
Dim cdfinv(0 To 6) As Double
For i = 0 To 6
```

```
   cdfvalue(i) = distNormal.CDF(x(i))
   MsgBox cdfvalue(i)
Next
For i = 0 To 6
   pdfvalue(i) = distNormal.PDF(x(i))
   MsgBox pdfvalue(i)
Next
For i = 0 To 6
   cdfinv(i) = distNormal.cdfinv(x(i))
   MsgBox cdfinv(i)
Next
```

class CSolverException

Class **CSolverException()** implements the exception class for the Solver SDK Platform.

When an error occurs when using C++/C#, or VB.NET, either in the system or in the currently executing application, the Solver Platform reports it by throwing an exception containing information about the error. Once thrown, an exception could be handled by the application or by the default exception handler. If using VB6, the OnError statement should be used.

Syntax:

```
CSolverException()
```

Parameters:

Property Name	Property Type	Property Attributes	Description
ErrorCode	_APIerr	get	Retrieves the the error code of the generated exception. For more information, refer to enum APIerr.
Description	string	get	Retrieves a descriptive text about the generated exception.

Examples:

C++ - Try/Catch

```
catch (CSolverException (e))
 {
      cout << "Throws Exception" << endl;
      cout << "Code  = " << e.ErrorCode << endl;
      cout << "Text = " << e.Description << endl;
      return 1;
 }
```

C# - Try/Catch

```
catch (SolverException ex)
 {
      MessageBox.Show("Exception " + ex.ExceptionType + "\r\n" +
      "Exception " + ex.Message,"SDK Exception Occurred");
 }
```

VB.NET - Try/Catch

```
Catch ex As SolverException
      MsgBox("Exception " & ex.ExceptionType & vbCrLf &
            ex.Message, MsgBoxStyle.Critical, "Exception")}
```

class CInputData

Class **CInputData()**

Class CInputData() represents a two-dimensional Excel range of problem inputs. (This object cannot be created directly and can only be used when loading a Solver Excel model that contains at least one PsiInput() parameter.) The PsiInput function provides a way to mark certain cells as containing (changeable) input data for an optimization or simulation model. This is especially useful if you create a model in Excel, but you want to run it outside Excel – possibly on an Intranet or Web Server – using Frontline System's Solver SDK Platform. When you use this function to mark certain cell ranges as Inputs, save your workbook, and load it into Solver SDK Platform under the control of your application program, these cell ranges are made available through SDK InputData objects. You can then get or set values in these cell ranges, before you run your optimization or simulation model via the SDK.

For an example of how to implement and use a PsiInput parameter in Excel and the SDK, please see the SDK Product Mix example which is available in C/C++/C#, Java, Matlab, VB.NET/VB6 at C:\Program Files\Frontline Systems\Solver SDK Platform\Examples. This example loads an Excel Solver model that contains three PsiInput parameters. Users are then allowed to change the values of these three parameters and solve the model.

Syntax:

```
CInputData()
```

Usage: If =PsiInput(B1) exists in the Excel worksheet where B1 is an Excel cell address.

C++	`CInputData d;` `d = prob.InputData[L"B1"];`
C#	`InputData d;` `d = prob.InputData["B1"];`
VB .NET	`Dim d as InputData` `d = prob.InputData("B1");`
VB 6	`Dim d as InputData` `Set d = prob.InputData("B1");`

Usage: If =PsiInput(Demand) exists in the Excel worksheet, where Demand is a defined name.

C++	`CInputData d;` `d = prob.InputData[L"Demand"];`
C#	`InputData d;` `d = prob.InputData["Demand"];`
VB .NET	`Dim d as InputData` `d = prob.InputData("Demand");`
VB 6	`Dim d as InputData` `Set d = prob.InputData("Demand");`
Description	Represents an Excel range of problem inputs.

Usage: If a PsiInput() parameter exists in an Excel workbook, the InputData vector can be accessed by index.

C++	`CInputData d;` `d = prob.InputData[0];`
C#	`InputData d;` `d = prob.InputData[0];`
VB .NET	`Dim d as InputData` `d = prob.InputData(0);`
VB 6	`Dim d as InputData` `Set d = prob.InputData(0);`

Parameters:

Property Name	Property Type	Property Attributes	Description
`Columns`	Integer	get	Gets the number of columns in the input range

Element	Double	get	Returns a single element of a 1-dimensional array i.e. if the Excel workbook contains the function =PsiInput(A1:C3), the Element property would return a 1-dimensional array with 9 elements.
Index	Integer	get	Returns the index of the input range.
Name	String	get	Returns the name (cell address) of the input range.
NumElements	Integer	Get	Returns the number of elements in the input range.
Problem	Problem	Get	Gets the Problem associated to this Problem input.
Rows	Integer	Get	Returns the number of rows in the input range.
Value	Double	Get	Returns the value of a single element in the input range – indexed by row and column.

Property Usage:

C+

```
//creates new problem
CProblem prob;
//loads the Excel Workbook into Solver SDK
prob.Load(L"C:\\ProductMix.xlsx",File_Format_XLStd);

CInputData d;
//Index of 0  = the first Input range on the Excel worksheet
d = prob.InputData[0];
//for use with the Excel worksheet defined name, "Input"
//d = prob.InputData[L"Input"];
//for use with the Excel function =PsiInput(D17:F17)
//d = prob.InputData[L"D17:F17"];

//returns the name (cell address) of the input range
//strName is now equal to "D17:F17"
const wchar_t* strName = d.Name;
//returns the index of the input range
//index is now equal to 0 (1st input range on worksheet)
int index = d.Index;
//returns the number of columns in the input range
//intColumns is now equal to 3 (3 columns in the range D17:F17)
int intColumns = d.Columns;
//returns the number of rows in the input range
//intRows is now equal to 1 (1 row in the range D17:F17)
```

```
int intRows =d.Rows;
//loops through the input range for input values
//dblValue will be equal to the value in each cell, i.e. the
//first time through the loop dblValue will equal the value in
//cell D17, the second time through the loop dblValue will
//equal the value in cell E17, and so on.
for (int i=0; i<intColumns; i++)
      for (int j=0; j<intRows; j++)
      double dblValue = d.Value[j,i];
```

C#

```
//creates new problem
Problem prob = new Problem();
//loads the Excel Workbook into Solver SDK
prob.Load("C:\\ProductMix.xlsx",File_Format.XLStd);

InputData d;
//for use with the Excel function =PsiInput(D17:F17)
d = prob.InputData["D17:F17"];
//Index of 0  = the first Input range on the Excel worksheet
//d = prob.InputData[0];
//for use with the Excel worksheet defined name, "Input"
//d = prob.InputData["Input"];

//returns the name (cell address) of the input range
//strName is now equal to "D17:F17"
string strName = d.Name;
//returns the index of the input range
//index is now equal to 0 (1st input range on worksheet)
int index = d.Index;
//returns the number of columns in the input range
//intColumns is now equal to 3 (3 column2 in the range D17:F17)
int intColumns = d.Columns;
//returns the number of rows in the input range
//intRows is now equal to 1 (1 rows in the range D17:F17)
int intRows =d.Rows;
//loops through input range for input values
//dblValue will be equal to the value in each cell, i.e. the
//first time through the loop dblValue will equal the value in
//cell D17, the second time through the loop dblValue will
//equal the value in cell E17, and so on.
double dblValue;
for (int i=0; i<intColumns; i++)
      for (int j=0; j<intRows; j++)
      dblValue = d.Value[j,i];
```

VB .NET

```
  'creates new problem
  Dim prob As New SolverPlatform.Problem
  'loads the Excel Workbook into Solver SDK
  prob.Load("C:\\ProductMix.xlsx",
SolverPlatform.File_Format.XLStd)

  Dim d As SolverPlatform.InputData
```

```
  'for use when the Excel function =PsiInput(D17:F17) is present
in workbook
  d = prob.InputData("D17:F17")
  'Index of 0  = the first Input range on the Excel worksheet
  'd = prob.InputData(0)
  'for use with the Excel worksheet defined name, "Input"
  'd = prob.InputData("Input");

  'returns the name (cell address) of the input range
  'strName is now equal to "D17:F17"
  Dim strName As String
  strName = d.Name
  'returns the index of the input range
  'index is now equal to 0 (1st input range on worksheet)
  Dim index As Integer
  index = d.Index
  'returns the number of columns in the input range
  'intColumns is now equal to 3 (3 columns in the range D17:F17)
  Dim intColumns As Integer
  intColumns = d.Columns
  'returns the number of rows in the input range
  'intRows is now equal to 1 (1 row in the range D17:F17)
  Dim intRows As Integer
  intRows = d.Rows
  'loops through input range for input values
  'dblValue will be equal to the value in each cell, i.e. the
  'first time through the loop dblValue will equal the value in
  'cell D17, the second time through the loop dblValue will
  'equal the value in cell E17, and so on.
  Dim dblValue As Double
  Dim i, j As Integer
  For i = 0 To intColumns - 1
     For j = 0 To intRows - 1
         dblValue = d.Value(j, i)
     Next
  Next
```

VB 6

```
'creates new problem
Dim prob As New SolverPlatform.Problem
'loads the Excel Workbook into Solver SDK
prob.Load "C:\ProductMix.xlsx", File_Format_XLStd
Dim d As SolverPlatform.InputData

'for use when the Excel function =PsiInput(D17:F17) is present in
workbook
Set d = prob.InputData("D17:F17")
'Index of 0  = the first Input range on the Excel worksheet
'd = prob.InputData(0)
'for use with the Excel worksheet defined name, "Input"
'd = prob.InputData("Input");

'returns the name (cell address) of the input range
'strName now equals "D17:F17"
Dim strName As String
```

```
strName = d.Name
'returns the index of the input range
'index now equals 0 (1st input range on worksheet)
Dim index As Integer
index = d.index
'returns the number of columns in the input range
'intColumns now equals 3 (3 columns in the range D17:F17)
Dim intColumns As Integer
intColumns = d.Columns
'returns the number of rows in the input range
'intRows now equals 1 (1 row in the range D17:F17)
Dim intRows As Integer
intRows = d.Rows
'loops through input range for input values
'dblValue will be equal to the value in each cell, i.e. the first
'time through the loop dblValue will equal the value in cell D17,
'the second time through the loop dblValue will equal the value in
'cell E17, and so on
Dim dblValue As Double
Dim i, j As Integer
For i = 0 To intColumns - 1
      For j = 0 To intRows - 1
            dblValue = d.Value(j, i)
      Next
Next
```

class CModelParam

Class **CModelParam()** sets the value for a single Model parameter. These parameters are available for use when loading a model created with the Risk Solver Platform (or any of our Excel Solvers) in Excel. It is possible to set the values for these parameters, however, most will have no effect once the model is loaded into the SDK as their values must be set in Excel before the model is loaded. Model parameters can be found in Excel on the Model Task Pane. This class also retrieves the default parameter setting and the name of the parameter, along with the minimum and maximum values that this parameter will accept.

Usage:

C++	`prob.Model.Params[L"Parameter"].Default;`
C#	`prob.Model.Params["Parameter"].Default;`
VB .NET	`prob.Model.Params("Parameter").Default`
VB6	`Prob.Model.Params("Parameter").Default`
Description	Gets the Default setting for the parameter, "Parameter".

C++	`prob.Model.Params[0].Value;`
C#	`prob.Model.Params[0].Value;`
VB .NET	`prob.Model.Params(0).Value`
VB6	`prob.Model.Params(0).Value`
Description	Sets the value for the first model parameter.

Parameters:

Property Name	Property Type	Property Attributes	Description
`Name`	String	get	Retrieves the name of the Model parameter.
`Value`	Double	get/set	Gets/Sets the value of the parameter. The value must be within the minimum and the maximum values. There are only three parameters whose values will have an effect on the Solver SDK Platform. *Only three parameter values will have any effect in the Solver SDK Platform, ChanceAutoAdjust, NumTrials, and NumOptimizations.*

Default	Double	get	Retrieves the default value for the parameter.
MaxValue	Double	get	Retrieves the maximum value of the parameter Value property.
MinValue	Double	get	Retrieves the minimum value of the parameter Value property.

Model Parameters:

Index	SDK Name	Default Value	Min Value	Max Value	Excel Name
0	DesiredModel*	4	1	4	Intended Model Type
5	Sparse*	0	0	1	User Internal Sparse
6	ActiveOnly*		0	1	Only Parse Active Sheet
9	TransformNonSmooth*	0	0	2	Nonsmooth Model Transformation
11	InteractiveOptimization*	0	0	1	Use Interactive Optimization
13	PsiOptimizationFunctions*	0	0	1	User Psi Functions
14	LoadMacros*	1	0	1	Load 2020 VBA Macros – Backward Compatibility Only, not implemented in V2020
15	WrapText*	1	0	1	Wrap Text in Output Pane
16	DlgStyle*	0	0	2	Solver Parameters Dialog
17	GuidedMode*	1	0	1	Guided Mode
18	OptInterpreter*	0	0	2	Interpreter (in Optimization Section)
19	UseofUncertainty*	1	1	4	Intended Use of Uncertainty
20	TransformStochastic*	0	0	2	Stochastic Transformation
21	ChanceConstraintNorm*	4	1	4	Chance Constraints Use
22	ChanceAutoAdjust	0	0	1	Auto Adjust Chance Constraints
23	NumTrials	1000	1	2,147,483,647	Trials per SimNumulation
30	NumOptimizations	1	1	2,147,483,647	Optimizations to Run
31	BigM*	1000000	-1E+30	1E+30	BigM Value

32	SolveMode*	0	0	3	Solve Mode
33	SolveUncertain*	0	0	3	Solve Uncertain Models
34	IncrementalParsing*	0	0	1	Incremental Parsing
35	DefaultLowerBound*	-1e+30	-1e+30	1e+30	Decision Vars Lower
36	DefaultUpperbound*	1e+30	-1e+30	1e+30	Decision Vars Upper
37	LogLevel*	1	-1	2	Log Level
39	ScanForBounds*	1	0	2	Scan for Bounds
40	LowerCensor	-1e+30	-1e+30	1e+30	Lower Censor
41	UpperCensor	-1e+30	-1e+30	1e+30	Upper Censor
42	CensorType	0	1	2	Censor Measure
43	LowerCutoff	-1e+30	-1e+30	1e+30	Lower Cutoff
44	UpperCutoff	-1e+30	-1e+30	1e+30	Upper Cutoff
45	CutoffType	0	0	2	Cutoff Measure
46	RunSpecificSim	-1	-1	2,147,483,647	Run Specific Simulation
47	RunSpecificOpt	-1	-1	2,147,483,647	Run Specific Optimization

*Parameter value settings will not influence the Solver SDK during the optimization or simulation process.

Parameter Descriptions: For more details please see the Risk Solver Platform User Guide

SDK Property Name	Description
DesiredModel	1 – Linear, 2 - Quadratic, 3 or 4 – Nonlinear
Sparse	1 – PsiInterpreter runs in Sparse model 0 – Dense Mode
ActiveOnly	1 – PsiInterpreter analyzes active worksheet only, 0 – Analyzes referenced cells throughout workbook (across all worksheets)
TransformNonSmooth	0 – Automatic, 1 – Always, 2 – Never
InteractiveOptimization	1 – Use Interactive Optimization, 0 – Do Not Use Interactive Optimization
PsiOptimizationFunctions	1 – Use Psi Optimization Functions, 0 – Do not use Psi Optimization Functions
LoadMacros	1 – True, 0 – False

WrapText	1 – True, 0 – False
DlgStyle	0 – Automatic, 1 – Old Excel Style, 2 – New Excel Style
GuidedMode	1 – True, 0 – False
OptInterpreter	0 – Automatic, 1 – Psi Interpreter, 2 – Excel Interpreter
UseOfUncertainty	1 – No Uncertainties, 2 – With Recourse Variables, 3 – In Chance Constraints, 4 – in Psi Stat Functions
TransformStochastic	0 – Automatic, 1 – Deterministic Equivalent Transformation, 2- Robust Counterpart Transformation
ChanceConstraintNorm	1 – L1 Norm, 2 – L2 Norm, 3 – L-Inf Norm, 4 – D-Norm
ChanceAutoAdjust	1 – Auto Adjust Chance Constraints, 0 – Do not Auto Adjust Chance Constraints
NumTrials	Number of trials to perform in each simulation.
NumOptimizations	Number of optimizations to perform.
BigM	A constant for use with model transformation in the Risk Solver Platform Solver for Excel. Its value should be larger than the upper bound of the nonsmoothly occurring variables. However, a large value can result in a poorly scaled model.
SolveMode	0 – Solve Complete Model, 1 – Analyze without Solving, 2 – Solve without Integer Constraints, 3 – Solve for Recourse Variables
SolveUncertain	0 – Automatic, 1 – Simulation Optimization, 2 – Stochastic Transformation, 3 – Stochastic Solver Engine
IncrementalParsing	1 – True, 0 – False
DefaultLowerBound	This sets a default lower bound on all decision variables in the Excel Solver model. Use -1E+30 to not set a default. This option will override Assume Nonnegative if the value is other than the default of -1E+30.
DefaultUpperBound	This sets a default upper bound on all decision variables in the Excel Solver model. Use 1E+30 to not set a default.
LogLevel	-1 – None, 0 – Minimal, 1- Normal, 2 – Verbose
ScanForBounds	1 – True, 0 – False
LowerCensor	Used to restrict the values of samples from the uncertain

	variable's distribution to lie above the value specified. If a value is sampled that is lower than the value specified, the trial value is set to the value of the Lower Censor.
UpperCensor	Used to restrict the values of samples from the uncertain variable's distribution to lie below the value specified. If a value is sampled that is larger than the value specified, the trial value is set to the value of the Upper Censor.
CensorType	Use this parameter only if you want to set global default bounds on the probability distributions of all uncertain variables. This option specifies the "units of measure" for the values you enter for the Lower Censor and Upper Censor parameters. Set this parameter to 0 to choose **None** (the default) if you do not want to set these global bounds. If you set this parameter to 1 (**Percentile)**, then the Lower Censor and Upper Censor values must be between 0.01 and 0.99, and they specify percentiles of each uncertain variable's probability distribution. If you set this parameter to 2(**Std Deviation)**, then the Lower Censor and Upper Censor can be any positive or negative value, and they specify the number of standard deviations away from the mean for each uncertain variable.
LowerCutoff	Used to restrict the values of samples from the uncertain variable's distribution to lie above the value specified. If a value is sampled that is lower than this value, then another sample is taken.
UpperCutoff	Used to restrict the values of samples from the uncertain variable's distribution to lie below the value specified. If a value is sampled that is larger than this value, than another sample is taken.
CutoffType	Use this parameter only if you want to set global default bounds on the probability distributions of all uncertain variables. This option specifies the "units of measure" for the values you enter for the Lower Cutoff and Upper Cutoff parameters. Set this parameter to 0 (**None)** (the default) if you do not want to set these global bounds. If you set this parameter to 1 (**Percentile)**, then the Lower Cutoff and Upper Cutoff values must be between 0.01 and 0.99, and they specify percentiles of each uncertain variable's probability distribution. If you set this parameter to 2 (**Std Deviation)**, then the Lower Cutoff and Upper Cutoff can be any positive or negative value, and they specify the number of standard deviations away from the mean for each uncertain variable.
RunSpecificSim	When performing multiple simulations, use this parameter to run a specific simulation (i.e. 3rd simulation of 10). A value of "-1" turns this parameter off.

`RunSpecificOpt`	When performing multiple optimizations, use this parameter to run a specific optimization (i.e. 3rd optimization of 10). A value of "-1" turns this parameter off.

Class CParameters()

Class CParameters() (CProblemParameter if using the COM API) is a class that represents a set of parameters in an Excel worksheet, such as PsiOptParam, PsiSimParam, and PsiSenParam. In the Excel worksheet, you can manually define an input cell as a parameter by simply selecting the cell, choosing **Parameters** and **Sensitivity**, **Optimization** or **Simulation** from the Risk Solver Platform Ribbon, and entering a lower and upper limit on values for the parameter, or a list of explicit values for the parameter.

In an Excel Solver Model, the function PsiOptParam accepts a list of different *values* that an objective or constraint coefficient should equal in different optimizations. *Values* is either a cell range, such as PsiOptParam(A1:A3), or an array of numbers, such as PsOptParam({6.0, 7.5, 9.0}). In the latter case, if the number of optimizations is set to 5, PsiOptParam will return 6.0 for the first optimization, 7.5 for the second optimization, 9.0 for the third optimization, 6.0 for the fourth optimization, and 7.5 for the fifth optimization.

In an Excel Solver Model, the function PsiSimParam accepts a list of different *values* that a variable should have in different simulations. (The variable value is the same for all trials in one simulation.) *Values* is either a cell range, such as PsiSimParam(A1:A3), or an array of numbers, such as PsiSimParam({6.0, 7.5, 9.0}). In the latter case, if the number of simulations is set to 5, PsiSimParam will return 6.0 for all trials of the first simulation, 7.5 for all trials of the second simulation, 9.0 for all trials of the third simulation, 6.0 for all trials of the fourth simulation, and 7.5 for all trials of the fifth simulation.

The values of PsiSenParam() function calls are automatically varied when you create a Sensitivity Analysis Report or Chart in the Excel Solver model. PsiSenParam() has limited use in the Solver SDK as neither a Sensitivity report nor chart is available outside of Excel.

Syntax:

```
CParameter()
```

Usage: If the Excel cell B1=PsiOptParam(1,2), =PsiSimParam(1,2) or =PsiSenParam(1,2) exists in the Excel worksheet.

C++	`CParameters d;` `d = prob.Parameters[L"B1"];`
C#	`Parmeters d;` `d = prob.Parameters["B1"];`
VB .NET	`Dim d as Parameters` `d = prob.InputData("B1");`
VB 6	`Dim d as ProblemParameter` `Set d = prob.InputData("B1");`

Usage: If an Excel cell with a defined name of "Demand" contains the formula =PsiOptParam(1,2), =PsiSimParam(1,2) or =PsiSenparam(1,2) exists in the Excel worksheet.

C++	`CParameters d;` `d = prob.Parameters[L"Demand"];`
C#	`Parameters d;` `d = prob.Parameters["Demand"];`
VB .NET	`Dim d as Parameters` `d = prob.Parameters("Demand");`
VB 6	`Dim d as Parameters` `Set d = prob.Parameters("Demand");`
Description	Represents an Excel range of problem inputs.

Usage: If an Excel cell contains the formula, =PsiOptParam(1,2), =PsiSimParam(1,2) or =PsiSenParam(1,2), the InputData vector can be accessed by index.

C++	`CParameters d;` `d = prob.Parameters[0];`
C#	`InputData d;` `d = prob.Parameters[0];`
VB .NET	`Dim d as Parameters` `d = prob.Parameters(0);`
VB 6	`Dim d as Parameters` `Set d = prob.Parameters(0);`

Parameters:

Property Name	Property Type	Property Attributes	Description
AllValues	DoubleMatrix	gets/set	Gets/Sets the matrix of AllValues.
Index	Integer	Get	Gets the index of the Parameter collection.
LowerBound	DoubleBuffer	Get	Gets the lower bound of the Parameter vector.
Name	String	gets/set	Gets/Sets the name of the Parameter vector.
NumTicks	Integer	gets/set	Gets/Sets the number of ticks in the Parameter vector. This parameter can only be set for a Sensitivity Parameter.
ParameterType	ParameterType	get/set	Gets/Sets the type of the Parameter Vector.
Problem	Problem	Get	Gets the Problem associated to this Problem parameter.
ReturnValue	DoubleBufferRO	Get	Returns the average of the minimum and maximum values of the individual parameter. If the Base Case exists, then returns the Base Case.
Size	Integer	get/set	Gets/Sets the size of the parameter vector.
UpperBound	DoubleBuffer	Get	Gets the upper bound of the Parameter vector.
Value	DoubleBufferRO	Get	Gets current values of the Parameter vector

Property Usage:

C+ - Inspect Parameters from Existing Excel Solver Model

```
//creates new problem
CProblem prob;
//loads the Excel Workbook, SDKTest.xlsx, into the Solver SDK
prob.Load(L"C:\\SDKTest.xlsx",File_Format_XLStd);

CParameter d;
//Index of 0  = the first Parameter range on the Excel worksheet
d = prob.Parameters[0];
//for use with the Excel worksheet defined name, "PsiParameters"
//d = prob.Parameters[L"PsiParameters"];
```

```
//for use with Excel workbook containing PsiOptParam,
//PsiSimParam or PsiSenparam in cells A1:A2
//d = prob.Parameters[L"A1:A2"];

//returns the name (cell address) of the input range
//strName is now equal to "A1:A2"
const wchar_t* strName = d.Name;
//returns the index of the input range
//index is now equal to 0 (1st input range on worksheet)
int index = d.Index;
//returns the parameter type, Parameter_Type_Optimization,
//Parameter_Type_Simulation, or Parameter_Type_Sensitivity
_Parameter_Type ptype = d.ParameterType;
//returns the number of simulations or optimizations
int intNumTicks= d.NumTicks;
//loops through the two Parameters, d.Size  = 2
for (int i=0; i<d.Size; i++)
{
  //returns the value of each parameter
  double dblValue = d.Value[i];
  //returns the average of the minimum and maximum values
  double dblReturnValue = d.ReturnValue[i];
  //returns the lower bound of the input values
  //if PsiOptParam(1,2), PsiSimParam(1,2) or PsiSenParam(1,2)
  //d.LowerBound[0] = 1
  double dblLowerBound = d.LowerBound[i];
  //returns the upper bound of the input values
  //if PsiOptParam(1,2), PsiSimParam(1,2) or PsiSenParam(1,2)
  //d.UpperBound[0] = 2
  double dblUpperBound = d.UpperBound[i];

  //create DoubleMatrix to inspect individual Paramter values
  CDoubleMatrix pMatrix = d.AllValues;
  //number of columns in Parameter vector
  numCols = pMatrix.Columns;
  //number of rows in Parameter vector
  numRows = pMatrix.Rows;
  //values of each Parameter
  value1 = pMatrix.Value[0][0];
  value2 = pMatrix.Value[0][1];
  value3 = pMatrix.Value[0][2];
  value4 = pMatrix.Value[0][3];
  value5 = pMatrix.Value[0][4];
}
```

C+ - Create a New Parameter Passing Values Explicitly

```
int nvars = 2, nfcns = 2;
//creates a new simulation problem with 2 uncertain variables
//and 2 uncertain functions.  Solver Type could be
//Solver_Type_Simulate, Solver_Type_Maximize,
//Solver_Type_Minimize, or Solver_Type_FindFeas
CProblem prob(Solver_Type_Simulate, nvars, nfcns);

//creates new parameter
CParameter p;
```

```
//names parameter SimParam1
p.Name = L"SimParam1";
//sets the type of parameter to Simulation - could also be
//Parameter_Type_Optimization or Parameter_Type_Sensitivity
p.ParameterType = Parameter_Type_Simulation;
//creates one parameter
p.Size = 1;
//sets the number of simulations to 5
prob.Solver.NumSimulations = 5;
//add the new parameter, SimParam1, to the parameter collection
prob.Parameters.Add(p);

//create a double matrix to hold parameter values
CDoubleMatrix newMatrix(p.Size, problem.Solver.NumSimulations);
newMatrix.Value[0][0] = 1;
newMatrix.Value[0][1] = 2;
newMatrix.Value[0][2] = 3;
newMatrix.Value[0][3] = 4;
newMatrix.Value[0][4] = 5;

//pass the values to the new Parameter, SimParam1.
prob.Parameters[L"SimParam1"].AllValues = newMatrix;

//create evaluator
CSimEvaluator evalSim;
prob.Evaluators[Eval_Type_Sample].Evaluator = & evalSim;
//passes disribituion for 2 uncertain variables
problem.Model.VarDistribution[0] =
      CDistribution(L"PsiUniform(0,2)");
problem.Model.VarDistribution[1] =
      CDistribution(L"PsiNormal(1,1)");
//runs a simulation which calls the evaluator evalSim of type
//Eval_Type_Sample
prob.Solver.Simulate();


class CSimEvaluator : public IEvaluator
{
public:
  virtual _Engine_Action Evaluate(CEvaluator& evaluator)
  {
    //create a pointer to prob
    CProblem& prob = evaluator.Problem;
    //First, we ask for the current uncertain variable and
    //constraint values
    const double* pVar = prob.VarUncertain.Value;
    double* pFcn = prob.FcnUncertain.Value;

    //calculate the uncertain functions using the SimParam1
    //parameter in the first uncertain function, pFcn[0].
    pFcn[0] =
      prob.Parameters[L"SimParam1"].AllValues.Value[0][0] *
      pVar[0] + 6 * pVar[1];
    pFcn[1] = 7 * pVar[0] * 8 * pVar[1];   // X*Y
```

```
    //tells the SDK to continue solving
    return Engine_Action_Continue;
      }
};
```

C+ - Create a New Parameter Passing Min and Max Values

```
int nvars = 2, nfcns = 2;
//creates a new simulation problem with 2 uncertain variables
//and 2 uncertain functions.  Solver Type could be
//Solver_Type_Simulate, Solver_Type_Maximize,
//Solver_Type_Minimize, or Solver_Type_FindFeas
CProblem prob(Solver_Type_Simulate, nvars, nfcns);

//creates new parameter
CParameter p;
//names parameter SimParam1
p.Name = L"SimParam1";
//sets the type of parameter to Simulation - could also be
//Parameter_Type_Optimization or Parameter_Type_Sensitivity
p.ParameterType = Parameter_Type_Simulation;
//creates one parameter
p.Size = 1;
//sets the number of simulations to 5
prob.Solver.NumSimulations = 5;
//sets the lower bound for the parameters to 1 and the upper
//bound to 5.  The SDK will subtract the lower bound from the
//upper bound and divide by the number of simulations ((upper -
//lower/prob.Solver.NumSimulations) to get the incremental
//parameter values for each simulation.  In this example, the
//parameter for the first simulation will equal 1, the 2nd, 2;
//the 3rd, 3; the 4th, 4; and the 5th, 5
//((5 - 1)/5) = 1 - the Solver will increment the parameter by 1
//for each simulation.
p.LowerBound[0] = 1;
p.UpperBound[0] = 5;
//add the new parameter, SimParam1, to the parameter collection
prob.Parameters.Add(p);

//create evaluator
CSimEvaluator evalSim;
prob.Evaluators[Eval_Type_Sample].Evaluator = & evalSim;
//passes disribituion for 2 uncertain variables
problem.Model.VarDistribution[0] =
      CDistribution(L"PsiUniform(0,2)");
problem.Model.VarDistribution[1] =
      CDistribution(L"PsiNormal(1,1)");

//runs a simulation which calls the evaluator evalSim of type
//Eval_Type_Sample
prob.Solver.Simulate();


class CSimEvaluator : public IEvaluator
{
public:
```

```
  virtual _Engine_Action Evaluate(CEvaluator& evaluator)
  {
    //create a pointer to prob
    CProblem& prob = evaluator.Problem;
    //First, we ask for the current uncertain variable and
    //constraint values
    const double* pVar = prob.VarUncertain.Value;
    double* pFcn = prob.FcnUncertain.Value;

    //calculate the uncertain functions using the SimParam1
    //parameter in the first uncertain function, pFcn[0].
    pFcn[0] =
      prob.Parameters[L"SimParam1"].AllValues.Value[0][0] *
      pVar[0] + 6 * pVar[1];
    pFcn[1] = 7 * pVar[0] * 8 * pVar[1];   // X*Y

    //tells the SDK to continue solving
    return Engine_Action_Continue;
      }
};
```

C# - Inspect Parameters from Existing Excel Solver Model

```
//creates new problem
using (Problem prob = new Problem())
{
  //loads the Excel workbook, SDKTest.xlsx, into the Solver SDK
  prob.Load("C:\\SDKTest.xlsx",File_Format.XLStd);
  int numCols, numRows;
  double value1, value2, value3, value4, value5;

  Parameter d;
  //Index of 0 = the first Parameter range on the Excel
  //worksheet
  d = problem.Parameters[0];
  //for use with Excel worksheet defined name, "PsiParameters"
  //d = problem.Parameters["PsiParameters"];
  //for use with Excel workbook containing PsiOptParam,
  //PsiSimParm or PsiSenParm in cells A1:A2
  //d = problem.Parameters["A1:A2"];

  //returns the name (cell address) of the input range
  //strName is now equal to "A1:A2"
  string strName = d.Name;
  //returns the index of the input range
  //index is now equal to 0 (1st input range on worksheet)
  int index = d.Index;
  //returns the parameter type, Parameter_Type.Optimization,
  //Parameter_Type.Simulation or Parameter_Type.Sensitivity
  Parameter_Type ptype = d.ParameterType;
  //returns the number of simulations or optimizations
  int intNumTicks = d.NumTicks;
  //loops through the two Parameters, d.Size = 2
  for (int i = 0; i < d.Size; i++)
  {
    //returns the value of each parameter
```

```
    double dblValue = d.Value[i];
    //returns the average of the minimum and maximum values
    double dblReturnValue = d.ReturnValue[i];
    //returns the lower bound of the input values
    //if PsiOptParam(1,2), PsiSimParam(1,2) or PsiSenParam(1,2)
    //d.LowerBound = 1 and d.UpperBound = 2
    double dblLowerBound = d.LowerBound[i];
    double dblUpperBound = d.UpperBound[i];
  }

//create DoubleMatrix to inspect individual Parameter values
DoubleMatrix pMatrix = new DoubleMatrix();
pMatrix = d.AllValues;
//number of columns in Parameter
numCols = pMatrix.Columns;
//number of rows in Parameter
numRows = pMatrix.Rows;
//return values for individual parameters
value1 = pMatrix[0,0];
value2 = pMatrix[0,1];
value3 = pMatrix[0,2];
value4 = pMatrix[0,3];
value5 = pMatrix[0,4];
}
```

C# - Create a New Parameter Passing Values Explicitly

```
//create new problem with 2 uncertain variables and 2 uncertain
//functions - this problem could also be an optimization model
//with 2 decision variables and 2 constraints.
//using (Problem prob = new Problem(Solver_Type.Maximize, nvars,
//    nfcns))
int nvars = 2, nfcns = 2;
using (Problem problem = new Problem(Solver_Type.Simulate,
      nvars, nfcns))
{
//create new parameter
Parameter p = new SolverPlatform.Parameter();
//name the parameter "SimParam1"
p.Name = "SimParam1";
//set the ParameterType, in this example we are creating a
//Simulation parameter.  The parameter type could also be
//Parameter_Type.Optimazation or Parameter_Type.Sensitivity
p.ParameterType = Parameter_Type.Simulation;
//setting the size of the parameter vector to 1
p.Size = 1;
//setting the number of simulations to 5
problem.Solver.NumSimulations = 5;
//add the new Parameter, p
problem.Parameters.Add(p);
//create double matrix to hold parameter values
DoubleMatrix newMatrix = new DoubleMatrix(p.Size,
p.Solver.NumSimulations);
newMatrix[0,0] = 1;
newMatrix[0,1] = 2;
newMatrix[0,2] = 3;
newMatrix[0,3] = 4;
```

```
newMatrix[0,4] = 5;
//pass the values from the DoubleMatrix to the Parameter
problem.Parameters["SimParam1"].AllValues = newMatrix;
//create evaluator of type Eval_Type.Sample to compute uncertain
//functions
problem.Evaluators[Eval_Type.Sample].OnEvaluate += new
EvaluateEventHandler(SimEvaluator_OnEvaluate);

//pass distributions for the two uncertain variables.
problem.Model.VarDistribution[0] = new
      Distribution("PsiUniform(0,2)");
problem.Model.VarDistribution[1] = new
      Distribution("PsiInvNormal(1,1)");

//run a simulation
problem.Solver.Simulate();
}

private Engine_Action SimEvaluator_OnEvaluate(Evaluator
evaluator)
{
  double ret;
  //create a pointer to the Problem
  Problem problem = evaluator.Problem;
  //create a pointer to the current variables
  double[] Var = evaluator.Problem.VarUncertain.Value.Array;
  //calculate the uncertain functions using the Simparam1
  //parameter in the first uncertain function, pFcn[0]
  evaluator.Problem.FcnUncertain.Value[0] =
      problem.Parameters["SimParam1"].Value[0] * Var[0] +
      Var[1];   // X+Y
  evaluator.Problem.FcnUncertain.Value[1] = Var[0] * Var[1];
      // X*Y
  //tells the SDK to continue solving
  return Engine_Action.Continue;
       }
```

C# - Create a New Parameter Passing Min and Max Values

```
//create new problem with 2 uncertain variables and 2 uncertain
//functions - this problem could also be an optimization model
//with 2 decision variables and 2 constraints.
//using (Problem prob = new Problem(Solver_Type.Maximize, nvars,
//    nfcns))
int nvars = 2, nfcns = 2;
using (Problem problem = new Problem(Solver_Type.Simulate,
      nvars, nfcns))
{
//create new parameter
Parameter p = new SolverPlatform.Parameter();
//name the parameter "SimParam1"
p.Name = "SimParam1";
//set the ParameterType, in this example we are creating a
//Simulation parameter.  The parameter type could also be
//Parameter_Type.Optimazation or Parameter_Type.Sensitivity
p.ParameterType = Parameter_Type.Simulation;
//setting the size of the parameter vector to 1
```

```
p.Size = 1;
//setting the number of simulations to 5
problem.Solver.NumSimulations = 5;
//sets the lower bound for all parameters to 1 and the upper
//bound to 5.  The SDK will subtract the lower bound from the
//upper bound and divide by the number of simulations ((upper -
//lower/prob.Solver.NumSimulations) to get the incremental
//parameter values for each simulation.  In this example, the
//parameter for the first simulation will equal 1, the 2nd, 2;
//the 3rd, 3; the 4th, 4; and the 5th, 5
//((5 - 1)/5) = 1 - the Solver will increment the parameter by 1
//for each simulation.
p.LowerBound[0] = 1;
p.UpperBound[0] = 5;

//add the new Parameter, p
problem.Parameters.Add(p);
//create evaluator of type Eval_Type.Sample to compute uncertain
//functions
problem.Evaluators[Eval_Type.Sample].OnEvaluate += new
EvaluateEventHandler(SimEvaluator_OnEvaluate);

//pass distributions for the two uncertain variables.
problem.Model.VarDistribution[0] = new
      Distribution("PsiUniform(0,2)");
problem.Model.VarDistribution[1] = new
      Distribution("PsiInvNormal(1,1)");

//run a simulation
problem.Solver.Simulate();
}

private Engine_Action SimEvaluator_OnEvaluate(Evaluator
evaluator)
{
  double ret;
  //create a pointer to the Problem
  Problem problem = evaluator.Problem;
  //create a pointer to the current variables
  double[] Var = evaluator.Problem.VarUncertain.Value.Array;
  //calculate the uncertain functions using the Simparam1
  //parameter in the first uncertain function, pFcn[0]
  evaluator.Problem.FcnUncertain.Value[0] =
      problem.Parameters["SimParam1"].Value[0] * Var[0] +
      Var[1];   // X+Y
  evaluator.Problem.FcnUncertain.Value[1] = Var[0] * Var[1];
      // X*Y
  //tells the SDK to continue solving
  return Engine_Action.Continue;
}
```

VB .NET – Inspect Parameters from Existing Excel Solver Model

```
//creates new problem
Dim prob As SolverPlatform.Problem = Nothing
//loads the Excel workbook, SDKTest.xlsx, into the Solver SDK
```

```
prob.Load("C:\\SDKTest.xlsx", File_Format.XLStd)


Dim d As New SolverPlatform.Parameter
//Index of 0 = the first Parameter range on the Excel worksheet
d = prob.Parameters(0)
//for use with the Excel worksheet defined name, "PsiParameters"
d = prob.Parameters("PsiParameters")
//for use with Excel workbook containing PsiOptParm, PsiSimParam
or PsiSenParam in cells A1:A2
d = prob.Parameters("A1:A2")

//returns the name (cell address) of the input range
//strName is now equal to "A1:A2"
Dim strName As String
strName = d.Name
//returns the index of the input range
//index is now equal to 0 (1st input range on worksheet)
Dim index As Integer
index = d.Index
//returns the paramter type, Parameter_Type.Optimization,
//Parameter_Type.Simulation, or Parameter_Type.Sensitivity
Dim ptype As SolverPlatform.Parameter_Type
ptype = prob.Parameter.ParameterType
//returns the number of simulations or optimizations
Dim intNumticks As Integer
intNumticks = d.NumTicks
//loops through the two Parameters, d.Size = 2
Dim i As Integer
Dim dblValue, dblReturnValue, dblLowerBound, dblUpperBound As
      Double
For i = 0 To d.Size - 1
  //returns the value of each parameter
  dblValue = d.Value(i)
  //returns the average of the minimum and maximum values
  dblReturnValue = d.ReturnValue(i)
  //returns the lower and upper bounds of the input values
  //ifPsiOptparam(1,2), PsiSimParam(1,2), or PsiSenParam(1,2)
  //d.LowerBound(i) = 1 and d.UpperBound(i) = 2
  dblLowerBound = d.LowerBound(i)
  dblUpperBound = d.UpperBound(i)
Next

Dim numCols, numRows As Integer
Dim value1, value2, value3, value4, value5 As Double
//creates DoubleMatrix to hold parameter vector values
Dim pMatrix As New DoubleMatrix
pMatrix = d.AllValues
//returns number of columns in Parameter vector
numCols = pMatrix.Columns
//returns number of rows in Parameter vector
numRows = pMatrix.Rows
//individual values of Parameter
value1 = pMatrix.Value(0, 0)
value2 = pMatrix.Value(0, 1)
```

```
value3 = pMatrix.Value(0, 2)
value4 = pMatrix.Value(0, 3)
value5 = pMatrix.Value(0, 4)
```

VB .NET – Create a New Parameter Passing Values Explicitly

```
'creates a new simulation problem with 2 uncertain variables and
2 uncertain functions.
Dim nvars As Integer = 2
Dim nfcns As Integer = 2
Dim prob As SolverPlatform.Problem
'Solver_Type could be Solver_Type_Simulate,
'Solver_Type_Maximize, Solver_Type_Minimize or
'Solver_Type_FindFeas
prob = New Problem(Solver_Type.Simulate, nvars, nfcns)

'creates new parameter
Dim p As New SolverPlatform.Parameter
'names parameter "SimParam1"
p.Name = "SimParam1"
'sets the type of parameter to Simulation - could also be
'Parameter_Type_Optimization or Parameter_Type_Sensitivity
p.ParameterType = Parameter_Type.Simulation
'creates one parameter
p.Size = 1
'sets the number of simulations to 5
prob.Solver.NumSimulations = 5
'adds the parameter to the parameter collection
prob.Parameters.Add(p)

'creates a double matrix to hold parameter values
Dim newMatrix As New DoubleMatrix
With newMatrix
.Create(Rows:=p.Size, Columns:=prob.Solver.NumSimulations)
  .Value(0, 0) = 1
  .Value(0, 1) = 2
  .Value(0, 2) = 3
  .Value(0, 3) = 4
  .Value(0, 4) = 5
End With
'passes the values of the new parameter to the SDK
prob.Parameters("SimParam1").AllValues = newMatrix

'creates evaluator
SimEvaluator = New Evaluator
prob.Evaluators(Eval_Type.Sample) = SimEvaluator

'passes uncertain vars distributions
prob.Model.VarDistribution(0) = New
      Distribution("PsiUniform(0,2)")
prob.Model.VarDistribution(1) = New
      Distribution("PsiNormal(1,1)")
prob.Solver.Simulate()

'Simulation Evaluator
```

```
Private Function SimEvaluator_OnEvaluate(ByVal evaluator As
      SolverPlatform.Evaluator) As SolverPlatform.Engine_Action
      Handles SimEvaluator.OnEvaluate
  'create new Problem
  Dim prob As New Problem
  'create pointer to evaluator.Problem
  prob = evaluator.Problem
  'create pointer to current variables
  Dim Var() As Double =
      evaluator.Problem.VarUncertain.Value.Array
  'calculate the uncertain functions using the SimParam1
  'parameter in the first uncertain function, pFcn[0]
  evaluator.Problem.FcnUncertain.Value(0) =
      evaluator.Problem.Parameters("SimParam1").Value(0) *
      Var(0) + Var(1)
  evaluator.Problem.FcnUncertain.Value(1) = Var(0) * Var(1)
  'tells the SDK to continue solving
  SimEvaluator_OnEvaluate = Engine_Action.[Continue]
End Function
```

VB .NET – Create a New Parameter Passing Min and Max Values

```
'create new simulation problem with 2 uncertain variables and 2
uncertain functions
Dim nvars As Integer = 2
Dim nfcns As Integer = 2
Dim prob As SolverPlatform.Problem
'Solver_Type could be Solver_Type_Simulate,
'Solver_Type_Maximize, Solver_Type_Minimize or
'Solver_Type_FindFeas
prob = New Problem(Solver_Type.Simulate, nvars, nfcns)

'creates new parameter
Dim p As New SolverPlatform.Parameter
'names parameter "SimParam1"
p.Name = "SimParam1"
'sets the type of parameter to Simulation - could also be
'Parameter_Type_Optimization or Parameter_Type_Sensitivity
p.ParameterType = Parameter_Type.Simulation
'creates one parameter
p.Size = 1
'sets the number of simulations to 5
prob.Solver.NumSimulations = 5
'sets the lower bound for all parameters to 1 and the upper
'bound to 5.  The SDK will subtract the lower bound from the
'upper bound and divide by the number of simulations ((upper -
'lower/prob.Solver.NumSimulations) to get the incremental
'parameter values for each simulation.  In this example, the
'parameter for the first simulation will equal 1, the 2nd, 2;
'the 3rd, 3; the 4th, 4; and the 5th, 5 ((5 - 1)/5) = 1 - the
'Solver will increment the parameter by 1 for each simulation.
p.LowerBound(0) = 1
p.UpperBound(0) = 5
'adds the parameter to the parameter collection
prob.Parameters.Add(p)
```

```
'creates evaluator
SimEvaluator = New Evaluator
prob.Evaluators(Eval_Type.Sample) = SimEvaluator

'passes uncertain vars distributions
prob.Model.VarDistribution(0) = New
      Distribution("PsiUniform(0,2)")
prob.Model.VarDistribution(1) = New
      Distribution("PsiNormal(1,1)")
prob.Solver.Simulate()

'Simulation Evaluator
Private Function SimEvaluator_OnEvaluate(ByVal evaluator As
      SolverPlatform.Evaluator) As SolverPlatform.Engine_Action
      Handles SimEvaluator.OnEvaluate
  'create new Problem
  Dim prob As New Problem
  'create pointer to evaluator.Problem
  prob = evaluator.Problem
  'create pointer to current variables
  Dim Var() As Double =
      evaluator.Problem.VarUncertain.Value.Array
  'calculate the uncertain functions using the SimParam1
  'parameter in the first uncertain function, pFcn[0]
  evaluator.Problem.FcnUncertain.Value(0) =
      evaluator.Problem.Parameters("SimParam1").Value(0) *
      Var(0) + Var(1)
  evaluator.Problem.FcnUncertain.Value(1) = Var(0) * Var(1)
  'tells the SDK to continue solving
  SimEvaluator_OnEvaluate = Engine_Action.[Continue]
End Function
```

VB 6 – Inspect Parameters from Existing Excel Solver Model

```
'creates new problem
Dim prob As New SolverPlatform.Problem
'loads the Excel Workbook into Solver SDK
prob.Load "C:\SDKTest.xlsx", File_Format_XLStd

Dim d As SolverPlatform.InputData
'for use when the Excel function =PsiInput(H1:H4) is present in
workbook
Set d = prob.InputData("H1:H4")
'Index of 0  = the first Input range on the Excel worksheet
'Set d = prob.InputData(0)
'for use with the Excel worksheet defined name, "Input"
'Set d = prob.InputData("Input");

'returns the name (cell address) of the input range
'strName now equals "H1:H4"
Dim strName As String
strName = d.Name
'returns the index of the input range
'index now equals 0 (1st input range on worksheet)
Dim index As Integer
index = d.index
```

```
'returns the number of columns in the input range
'intColumns now equals 1 (1 column in the range H1:H4)
Dim intColumns As Integer
intColumns = d.Columns
'returns the number of rows in the input range
'intRows now equals 4 (4 rows in the range H1:H4)
Dim intRows As Integer
intRows = d.Rows
'loops through input range for input values
'dblValue will be equal to the value in each cell, i.e. the first
time       'through the loop dblValue will equal the value in cell
H1, the second time
'through the loop dblValue will equal the value in cell H2, and so
on
Dim dblValue As Double
Dim i, j As Integer
For i = 0 To intColumns - 1
      For j = 0 To intRows - 1
            dblValue = d.Value(j, i)
      Next
Next
```

VB 6 – Create a New Parameter Passing Values Explicitly

```
Dim nvars, nfcns As Integer
nvars = 2
nfcns = 2
'creates new simulation problem with 2 uncertain variables and 2
'uncertain functions. Solver_Type could be
'Solver_Type_Simulate, Solver_Type_Maximize,
'Solver_Type_Minimize, or Solver_Type_FindFeas
Dim prob As New Problem
prob.VarUncertain.Size = nvars
prob.FcnUncertain.Size = nfcns
prob.Solver.SolverType = Solver_Type_Simulate

'creates new parameter
Dim p As New SolverPlatform.ProblemParameter
p.Name = "SimParam1"
'names the parameter "SimParam1"
'sets the type of parameter to Simulation - could also be
'Parameter_Type_Optimization or Parameter_Type_Sensitivity
p.ParameterType = Parameter_Type_Simulation
'creates one parameter
p.Size = 1
'sets the number of simulations to 5
prob.Solver.NumSimulations = 5
'adds the new parameter to the collection
prob.Parameters.Add p

'creates a double matrix to hold parameter values
Dim newMatrix As New DoubleMatrix
With newMatrix
 .Create Rows:=p.Size, Columns:=prob.Solver.NumTrials
 .Value(0, 0) = 1
 .Value(0, 1) = 2
```

```
 .Value(0, 2) = 3
 .Value(0, 3) = 4
 .Value(0, 4) = 5
End With
'pass the parameter values to the SDK
prob.Parameters("SimParam1").AllValues = newMatrix

'create evaluator
Set SimEvaluator = New Evaluator
prob.Evaluators(Eval_Type_Sample) = SimEvaluator

'passes distributions for 2 uncertain variables
Dim distUniform As New Distribution
distUniform.Init "PsiUniform(0,2)"
prob.Model.VarDistribution(0) = distUniform
Dim distNormal As New Distribution
distNormal.Init "PsiNormal(1,1)"
prob.Model.VarDistribution(1) = distNormal
'runs a simulation which calls the evaluator evalSim of type
'Eval_Type_Sample
prob.Solver.Simulate

'Simulation Evaluator
Private Function SimEvaluator_Evaluate(ByVal Evaluator As
SolverPlatform.IEvaluator) As SolverPlatform.Engine_Action
  'creates new Problem
  Dim prob As New Problem
  Dim Var() As Double
  'creates pointer to the current variables
  Var = Evaluator.Problem.VarUncertain.Value.Array
  'calculates the uncertain functions using the SimParam1
  'parameter in the first uncertain function, pFcn[0]
  Evaluator.Problem.FcnUncertain.Value(0)=
      Evaluator.Problem.Parameters("SimParam1").Value(0) *
      Var(0) + Var(1)
  Evaluator.Problem.FcnUncertain.Value(1) = Var(0) * Var(1)
  'tells the SDK to continue solving
  SimEvaluator_Evaluate = Engine_Action_Continue
End Function
```

VB 6 – Create a New Parameter Passing Min and Max Values

```
Dim nvars, nfcns As Integer
nvars = 2
nfcns = 2
'creates new simulation problem with 2 uncertain variables and 2
'uncertain functions. Solver_Type could be
'Solver_Type_Simulate, Solver_Type_Maximize,
'Solver_Type_Minimize, or Solver_Type_FindFeas
Dim prob As New Problem
prob.VarUncertain.Size = nvars
prob.FcnUncertain.Size = nfcns
prob.Solver.SolverType = Solver_Type_Simulate

'creates new parameter
Dim p As New SolverPlatform.ProblemParameter
```

```
p.Name = "SimParam1"
'names the parameter "SimParam1"
'sets the type of parameter to Simulation - could also be
'Parameter_Type_Optimization or Parameter_Type_Sensitivity
p.ParameterType = Parameter_Type_Simulation
'creates one parameter
p.Size = 1
'sets the number of simulations to 5
prob.Solver.NumSimulations = 5

'sets the lower bound for all parameters to 1 and the upper
'bound to 5.  The SDK will subtract the lower bound from the
'upper bound and divide by the number of simulations ((upper -
'lower/prob.Solver.NumSimulations) to get the incremental
'parameter values for each simulation.  In this example, the
'parameter for the first simulation will equal 1, the 2nd, 2;
'the 3rd, 3; the 4th, 4; and the 5th, 5 ((5 - 1)/5) = 1 - the
'Solver will increment the parameter by 1 for each simulation.
p.LowerBound(0) = 1
p.UpperBound(0) = 5

'adds the new parameter to the collection
prob.Parameters.Add p

'creates a double matrix to hold parameter values
Dim newMatrix As New DoubleMatrix
With newMatrix
 .Create Rows:=p.Size, Columns:=prob.Solver.NumTrials
 .Value(0, 0) = 1
 .Value(0, 1) = 2
 .Value(0, 2) = 3
 .Value(0, 3) = 4
 .Value(0, 4) = 5
End With
'pass the parameter values to the SDK
prob.Parameters("SimParam1").AllValues = newMatrix

'create evaluator
Set SimEvaluator = New Evaluator
prob.Evaluators(Eval_Type_Sample) = SimEvaluator

'passes distributions for 2 uncertain variables
Dim distUniform As New Distribution
distUniform.Init "PsiUniform(0,2)"
prob.Model.VarDistribution(0) = distUniform
Dim distNormal As New Distribution
distNormal.Init "PsiNormal(1,1)"
prob.Model.VarDistribution(1) = distNormal
'runs a simulation which calls the evaluator evalSim of type
'Eval_Type_Sample
prob.Solver.Simulate

'Simulation Evaluator
Private Function SimEvaluator_Evaluate(ByVal Evaluator As
SolverPlatform.IEvaluator) As SolverPlatform.Engine_Action
  'creates new Problem
```

```
  Dim prob As New Problem
  Dim Var() As Double
  'creates pointer to the current variables
  Var = Evaluator.Problem.VarUncertain.Value.Array
  'calculates the uncertain functions using the SimParam1
  'parameter in the first uncertain function, pFcn[0]
  Evaluator.Problem.FcnUncertain.Value(0)=
      Evaluator.Problem.Parameters("SimParam1").Value(0) *
      Var(0) + Var(1)
  Evaluator.Problem.FcnUncertain.Value(1) = Var(0) * Var(1)
  'tells the SDK to continue solving
  SimEvaluator_Evaluate = Engine_Action_Continue
End Function
```

Enumerations

Click to navigate to the desired topic:

- enum APIerr
- enum License_Loc
- enum Solver_Type
- enum Depend_Type
- enum Problem_Type
- enum Variable_Type
- enum Function_Type
- enum Eval_Type
- enum Engine_Action
- enum Array_Order
- enum File_Format
- enum Distribution_Type
- enum DistFit_Type
- enum Frequency_Type
- enum Simulate_Status
- enum Optimize_Status
- enum Integer_Type
- enum Cone_Type
- enum IIS_Status
- enum License_SelectorCode
- enum DistParam_Type
- enum Dist_VarFcn
- enum DistFit_SortType
- enum Check_Type
- enum Convex_Type
- enum Stochastic_Type
- enum Solve_Type
- enum Simulate_Type
- enum Chance_Type

- enum Parameter_Type

enum APIerr

Enumeration for specifying different types of exceptions.

Syntax:

C / C++

```
public enum _APIerr
```

VB .NET / C#

```
public enum APIerr
```

Members:

Member Name	Error Code	Description
APIerr_Success	0	In the Solver SDK Platform, return code 0 always indicates the method has been successful.
APIerr_OutOfMemory	1	Out of memory.
APIerr_NotYetImplemented	2	This feature not implemented in this version.
APIerr_ProbPtr_Invalid	3	Invalid problem handle. A Problem must first be defined.
APIerr_Problem_Undefined	4	Problem definition is incomplete.
APIerr_License_None	5	No license found.
APIerr_License_Invalid	6	No valid license found.
APIerr_License_Expired	7	License has expired.
APIerr_License_Remain	8	License usage exhausted.
APIerr_Problem_Invalid	9	Problem is invalid.
APIerr_Problem_Objsense	10	Problem type must be either Solver_Type_Maximize, or Solver_Type_Minimize, or Solver_Type_FindFeas.
APIerr_SPConLinear_Missing	11	Matrix is invalid. Majorstart or

		Minorindex is missing.
APIerr_SPConLinear_BegInvalid	12	Matrix is invalid. Majorstart values must be increasing, and less than the total number "L" of elements in the matrix.
APIerr_SPConLinear_IdxInvalid	13	Matrix is invalid. Minorindex element is out of range.
APIerr_ObjJacDep_TypeInvalid	14	Objective Type is invalid.
APIerr_Engine_DLLNotFound	15	Engine Library could not be found.
APIerr_Engine_EntryNotFound	16	Engine entry point could not be found.
APIerr_Engine_LoadFailed	17	Engine could not be loaded.
APIerr_Engine_ProbMismatch	18	The selected Engine cannot solve a problem of this type. Please select another Engine.
APIerr_Engine_NotSupported	19	Engine could not be found.
APIerr_Optimize_NoMultiThread	20	Multiple threads are not supported in this version.
APIerr_Optimize_ResultMissing	21	Result is missing.
APIerr_Optimize_AllDiffIdx	22	The group index for an alldifferent constraint must be 1 for the first constraint, 2 for the second, etc.
APIerr_Optimize_AllDiffOne	23	An alldifferent constraint must include at least 2 variables.
APIerr_Optimize_AllDiffBnd	24	The lower bounds on the variables in an alldifferent constraint must be 1; the upper bounds must be equal to the number of variables included in the constraint.
APIerr_Param_Missing	25	Parameter does not exist.
APIerr_Param_Value	26	Parameter value is out of range.
APIerr_Collection_Index	27	Invalid index into the Collection.
APIerr_Optimize_ConeIdx	28	The group index for a cone constraint must be 1 for the first constraint, 2 for the second, etc.
APIerr_Optimize_ConeOne	29	A cone constraint must include at least 2 variables.
APIerr_Function_NotDefined	30	Function is not a member of the Function Collection.
APIerr_Variable_NotDefined	31	Variable is not a member of the Variable

		Collection.
APIerr_Variable_ProbInUse	32	Variable is already in use by another problem.
APIerr_Function_ProbInUse	33	Function is already in use by another problem.
APIerr_Function_NameExist	34	Function name is already in use.
APIerr_Function_NameInvalid	35	Function name is invalid.
APIerr_Variable_NameExist	36	Variable name is already in use.
APIerr_Variable_NameInvalid	37	Variable name is invalid.
APIerr_Case_NotSupported	38	Case is not a member of the Case Collection.
APIerr_Model_DistInvalid	39	Invalid distribution.
APIerr_Save_Invalid	40	Problem could not be saved.
APIerr_Engine_Standard	41	You cannot unload one of the standard Engines.
APIerr_Load_Invalid	42	Problem could not be loaded.
APIerr_Array_Bounds	43	Array is being accessed outside its bounds.
APIerr_Invalid_Value	44	Invalid value.
APIerr_Collection_Readonly	45	Collection cannot be changed at this time. (This code is thrown when a change made to a Collection could easily render a Problem invalid. For example, you set a matrix and try to remove a variable.
APIerr_Feature_Readonly	46	Feature becomes read-only after being assigned to a Problem.
APIerr_Assignment_Error	47	Objects are assigned to different Problems.
APIerr_DifferentObjectTypes	48	Assignment between objects of different types.
APIerr_ObjectReinitialized	49	Previously initialized object cannot be reinitialized.
APIerr_WrongArraySize	50	Input array size does not match the calculated size.
APIerr_Unknown	51	Unexpected error. Please contact Technical Support.
APIerr_MatrixInFastUpdateMode	52	Operation cannot be performed because the matrix is in fast update mode.
APIerr_Model_StatInvalid	53	Invalid Statistics

APIerr_WeightMatrix_Invalid	54	Weight matrix has invalid values.
APIerr_Model_DistAltParamsInvalid	55	Invalid Distribution
APIerr_Model_DistCompBoundsInvalid	56	Invalid distribution computed bounds.
APIerr_Duplicate_Name	57	Duplicate Name
APIerr_Not_Associated_Array	58	Not associated array
APIerr_Not_Associated_Excel_Object_Model	59	Not associated Excel object model.
APIerr_Invalid_Excel_Range_Name	60	Invalid Excel range name.
APIerr_Input_NotDefine	61	Input is not a member of the Inputs collection.
APIerr_PropOrFunction_NotImplemented	62	Property or function not implemented.
APIerr_MatrixCellNotExist	63	Property or function not implemented.
APIerr_Parameter_NameExist	64	Parameter name is already in use.
APIerr_Parameter_NameInvalid	65	Parameter name is invalid.
APIerr_Parameter_NotDefined	66	Parameter is not a member of the Parameters collection.
APIerr_Duplicate_Indices	67	Duplicated indices

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum License_Loc

Enumeration for specifying the meaning of the licensing string.

Syntax:

C / C++

```
public enum _License_Loc
```

VB .NET / C#

```
public enum License_Loc
```

Members:

Member Name	Code	Description
License_Loc_Str	0	The string contains the licensing information.
License_Loc_Variable	1	The string is the environment variable name.
License_Loc_File	2	The string is the name/path of the license file.
License_Loc_Registry	3	The string is the name/path of the Windows Registry entry.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Solver_Type

Enumeration for specifying either optimization objective or simulation.

Syntax:

C/C++

```
public enum _Solver_Type
```

VB .NET / C#

```
public enum Solver_Type
```

Members:

Member Name	Code	Description
Solver_Type_Simulate	0	Specifies simulation.
Solver_Type_Minimize	1	Specifies minimization objective.
Solver_Type_Maximize	2	Specifies maximization objective.
Solver_Type_FindFeas	3	Specifies finding a feasible solution for the constraints, without an objective.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Depend_Type

Enumeration for specifying the nature of an element in the matrix of function values.

A matrix of `Depend_Type` values specifies the nature of each dependence of a function on a variable (see the Members table below). The Solver will use a corresponding matrix of double values (`CDoubleMatrix`) that holds the first derivatives of functions with respect to variables.
In a linear programming problem, all of the first matrix values are `Depend_Type_None` or `Depend_Type_Linear`, and all of the second matrix values (doubles) are zero or constant numbers.
In a nonlinear problem, the `CDoubleMatrix` (Jacobian matrix) values are not constant – they can change at each new trial point explored by the Solver.

Syntax:

C / C++

```
public enum _Depend_Type
```

VB .NET / C#

```
public enum Depend_Type
```

Members:

Member Name	Code	Description
Depend_Type_None	0	No matrix element present for this constraint and variable.
Depend_Type_Linear	1	The variable behaves linear for this constraint. (So the matrix element is constant)
Depend_Type_Quadratic	2	The variable is quadratic with respect to this constraint.
Depend_Type_QuadCon	4	The variable is quadratic with respect to this constraint.
Depend_Type_Convex	8	The variable is convex with respect to this constraint.
Depend_Type_Smooth	16	The variable is smooth nonlinear with respect to this constraint.
Depend_Type_Nonsmooth	32	The variable is non-smooth with respect to this constraint.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Problem_Type

Enumeration for specifying the Problem Type.

Syntax:

C / C++

```
public enum _Problem_Type
```

VB .NET / C#

```
public enum Problem_Type
```

Members:

Member Name	Code	Description
Problem_Type_NA	0	Problem type not available.
Problem_Type_OptLP	1	Linear optimization problem.
Problem_Type_OptQP	2	Quadratic optimization problem.
Problem_Type_OptQCP	4	Quadratically constrained optimization problem.
Problem_Type_OptCVX	8	Convex optimization problem.
Problem_Type_OptNLP	16	Non-linear optimization problem.
Problem_Type_OptNSP	32	Non-smooth optimization problem.
Problem_Type_SimIndep	256	Independent simulation problem.
Problem_Type_SimCorrel	512	Correlational simulation problem.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Variable_Type

Enumeration for specifying the variable type.

Syntax:

C / C++

```
public enum _Variable_Type
```

VB .NET / C#

```
public enum Variable_Type
```

Members:

Member Name	Code	Description
Variable_Type_All	0	Specifies variables of all types.
Variable_Type_Case	1	Case variable.
Variable_Type_Uncertain	2	Uncertain variable.
Variable_Type_Decision	4	Decision variable.
Variable_Type_Recourse	8	Recourse variable. A recourse variable is a type of decision variable whose value is determined at a later stage in time, whereas a 'regular' decision variable value must be determined today.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Function_Type

Enumeration for specifying the function type.

Syntax:

C / C++

```
public enum _Function_Type
```

VB .NET / C#

```
public enum Function_Type
```

Members:

Member Name	Code	Description
Function_Type_All	0	Specifies functions of all types.
Function_Type_Case	1	Case function. (Not yet implemented)
Function_Type_Uncertain	2	Uncertain function.
Function_Type_Objective	4	Objective function.
Function_Type_Constraint	8	Constraint function.
Function_Type_Chance	16	Soft constraint function. While a regular (“hard”) constraint must always be satisfied, a soft constraint may be violated by a small amount, for example $A \leq 1$ where A turns out to be 1.0001 at the solution. Soft constraint violations are penalized in the objective function.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Eval_Type

Enumeration for specifying different kind of evaluators or callback routines.

Syntax:

C / C++

```
public enum _Eval_Type
```

VB .NET / C#

```
public enum Eval_Type
```

Members:

Member Name	Code	Description
Eval_Type_Simulation	1	Simulation type. The routine is called after each simulation.
Eval_Type_Subproblem	2	Evaluation of a subproblem. The routine is called at every new subproblem.
Eval_Type_Iteration	4	Evaluation of iteration. The routine is called at every iteration.
Eval_Type_NewSolution	8	Routine called when a new solution (incumbent) is found.
Eval_Type_Function	256	Routine called to compute the objective and the function values.
Eval_Type_Gradient	512	Routine called to compute the objective and function gradients. The Evaluator makes sure the computing is done with the proper gradients. The Solver will compute numerical approximations of derivatives, and compare against values supplied by the user's evaluator. When the values are different, the GradError evaluator is called.
Eval_Type_GradError	1024	Routine called when Derivatives option = 4, and the user wants to test his own derivatives.
Eval_Type_Hessian	2048	Routine called to compute Hessian values (= 2nd order derivatives) for a specific function.

Eval_Type_Sample	65536	Routine called to compute the uncertain function values in a simulation problem.
Eval_Type_SimBegin	262144	This routine is called at the beginning of a simulation run (of two or more simulations).
Eval_Type_SimComplete	524288	This routine is called at the end of a simulation run (of two or more simulations).
Eval_Type_Optimization	1048576	This routine is called at the end of an optimization.
Eval_Type_Custom	2097152	This routine is called when Solver "pauses".
Eval_Type_OptBegin	4194304	This routine is called at the beginning of an optimization run (of two or more solves).
Eval_Type_OptEnd	8388608	This routine is called at the end of an optimization run (of two or more solves).

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Engine_Action

Enumeration for specifying the Engine action.

The possible return values (listed in the Members table) can be passed back to the SDK when one the evaluators is called.

Syntax:

C / C++

```
public enum _Engine_Action
```

VB .NET / C#

```
public enum Engine_Action
```

Members:

Member Name	Code	Description
Engine_Action_Continue	0	Continuing evaluation. The Solver continues the optimization; this is the "normal" return code.
Engine_Action_Evaluate	1	Starting evaluation. The Solver should call the "Function Evaluator" next, in order to pass the current set of variable values. This can be used when you want to know the exact variable values at a specific point. You would also have to set some kind of flag to let your own function evaluator know that it is being called for this reason.
Engine_Action_Stop	2	Halting evaluation. (Aborts optimization.)

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Array_Order

Enumeration for specifying the arrangement of elements in an array.

Syntax:

C / C++

```
public enum _Array_Order
```

VB .NET / C#

```
public enum Array_Order
```

Members:

Member Name	Code	Description
Array_Order_ByRow	1	Arrangement by row. (The elements of a matrix are either passed or expected in row-wise order.)
Array_Order_ByCol	-1	Arrangement by column. (The elements of a matrix either are passed or expected in column-wise order.)

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum File_Format

Enumeration for specifying file format.

Syntax:

C / C++

```
public enum _File_Format
```

VB .NET / C#

```
public enum File_Format
```

Members:

Member Name	Code	Description
File_Format_All	0	Specifies all file formats.
File_Format_LP	1	LP file format. (Text-only file format for linear and quadratic problems.)
File_Format_MPS	2	MPS file format (for linear and quadratic problems).
File_Format_LPFML	3	LPFML file format (XML-based) for linear problems.
File_Format_OSIL	3	OSIL file format (XML – based) for LP, QP, and QCP problems.
File_Format_XLStd	4	Frontline Standard Excel format.
File_Format_XLPSI	5	Frontline PSI Excel format.
File_Format_XLWB	6	Not for use with the Solver SDK Platform.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Distribution_Type

Enumeration for specifying the distribution type for an uncertain variable.

Syntax:

C / C++

```
public enum _Distribution_Type
```

VB .NET / C#

```
public enum Distribution_Type
```

Members:

Member Name	Code	Description
Distribution_Type_None	0	Distribution type is not defined.
Distribution_Type_Bernoulli	1	Bernoulli distribution of the uncertain variable.
Distribution_Type_Beta	2	Beta distribution of the uncertain variable.
Distribution_Type_BetaGeneral	4	Beta Generalized distribution of the uncertain variable.
Distribution_Type_BetaSubj	5	Beta Subjective distribution of the uncertain variable.
Distribution_Type_Binomial	6	Binomial distribution of the uncertain variable.
Distribution_Type_Cauchy	7	Cauchy distribution of the uncertain variable. (Not available in the current SDK version.)
Distribution_Type_Chisquared	8	Chisquared distribution of the uncertain variable.
Distribution_Type_Cumulative	9	Cumulative distribution of the uncertain variable.
Distribution_Type_Discrete	13	Discrete distribution of the uncertain variable.
Distribution_Type_DiscreteUniform	14	Discrete uniform distribution of the uncertain variable. (Not available in the current SDK version.)

Distribution_Type_Erf	15	Erf distribution of the uncertain variable.
Distribution_Type_Erlang	16	Erlang distribution of the uncertain variable.
Distribution_Type_Exponential	17	Exponential distribution of the uncertain variable.
Distribution_Type_Gamma	18	Gamma distribution of the uncertain variable.
Distribution_Type_General	19	General distribution of the uncertain variable.
Distribution_Type_Geometric	20	Geometric distribution of the uncertain variable.
Distribution_Type_Histogram	21	Histogram distribution of the uncertain variable.
Distribution_Type_Hypergeometric	22	Hypergeometric distribution of the uncertain variable.
Distribution_Type_IntegerUniform	24	Integer Uniform distribution of the uncertain variable.
Distribution_Type_InverseGaussian	25	Inverse Gaussian distribution of the uncertain variable.
Distribution_Type_Laplace	26	Laplace distribution of the uncertain variable. (Not available in the current SDK version.)
Distribution_Type_Logarithmic	27	Logarithmic distribution of the uncertain variable. (Not available in the current SDK version.)
Distribution_Type_Logistic	28	Logistic distribution of the uncertain variable.
Distribution_Type_Loglogistic	29	Loglogistic distribution of the uncertain variable.
Distribution_Type_Lognormal	30	Lognormal distribution of the uncertain variable.
Distribution_Type_Lognorm2	31	Lognorm2 distribution of the uncertain variable.
Distribution_Type_MaxExtreme	32	MaxExtreme distribution of the uncertain variable.
Distribution_Type_MinExtreme	33	MinExtreme distribution of the

		uncertain variable.
Distribution_Type_Myerson	34	Myerson distribution of the uncertain variable.
Distribution_Type_NegativeBinomial	35	Negative binomial distribution of the uncertain variable.
Distribution_Type_Normal	36	Normal distribution of the uncertain variable.
Distribution_Type_Pareto	37	Pareto distribution of the uncertain variable.
Distribution_Type_Pareto2	38	Pareto Second Kind distribution of the uncertain variable.
Distribution_Type_Pearson5	39	Pearson type 5 distribution of the uncertain variable. (Not available in the current SDK version.)
Distribution_Type_Pearson6	40	Pearson type 6 distribution of the uncertain variable. (Not available in the current SDK version.)
Distribution_Type_Pert	41	Pert distribution of the uncertain variable.
Distribution_Type_Poisson	42	Poisson distribution of the uncertain variable.
Distribution_Type_Rayleigh	43	Rayleigh distribution of the uncertain variable.
Distribution_Type_Student	47	Student distribution of the uncertain variable.
Distribution_Type_Triangular	48	Triangular distribution of the uncertain variable.
Distribution_Type_TriGeneral	49	Trigeneral distribution of the uncertain variable.
Distribution_Type_Uniform	50	Uniform distribution of the uncertain variable.
Distribution_Type_Weibull	52	Weibull distribution of the uncertain variable.
Distribution_Type_NormalSkew	53	Generalized Normal based on Myerson
Distribution_Type_Sip	55	Stochastic Information Packet
Distribution_Type_Last	55	The last index in the distribution

enumerator.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum DistFit_Type

Enumeration for specifying the distribution types fitted by methods `Fit` and `Autofit`.

Syntax:

C / C++

```
public enum _DistFit_Type
```

VB .NET / C#

```
public enum DistFit_Type
```

Members:

Member Name	Code	Description
DistFit_None	0	None
DistFit_Bernoulli	1	Bernoulli distribution type.
DistFit_Beta	2	Beta distribution type.
DistFit_BetaGeneral	4	BetaGeneral distribution type.
DistFit_Binomial	6	Binomial distribution type.
DistFit_Cauchy	7	Cauchy distribution type.
DistFit_Chisq	8	ChiSquare distribution type.
DistFit_Erf	15	Erf distribution type.
DistFit_Erlang	16	Erlang distribution type.
DistFit_Exponential	17	Exponential distribution type.
DistFit_Gamma	18	Gamma distribution type.
DistFit_Geometric	20	Geometric distribution type.
DistFit_IntUniform	24	IntUniform distribution type.
DistFit_InvGauss	25	Inverse Gaussian distribution type.
DistFit_Laplace	26	Laplace distribution type.
DistFit_Logarithmic	27	Logarithmic distribution type.
DistFit_Logistic	28	Logistic distribution type.

DistFit_LogLogistic	29	LogLogistic distribution type.
DistFit_LogNormal	30	LogNormal distribution type.
DistFit_MaxExtreme	32	MaxExtreme distribution type.
DistFit_MinExtreme	33	MinExtreme distribution type.
DistFit_NegBinomial	35	NegBinomial distribution type.
DistFit_Normal	36	Normal distribution type.
DistFit_Pareto	37	Pareto distribution type.
DistFit_Pearson5	39	Pearson5 distribution type.
DistFit_Poisson	42	Poisson distribution type.
DistFit_Rayleigh	43	Rayleigh distribution type.
DistFit_Student	47	Student distribution type.
DistFit_Uniform	50	Uniform distribution type.
DistFit_Weibull	52	Weibull distribution type.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Frequency_Type

Enumeration for specifying the frequency type to be returned.

The `Frequency_Type` is passed as an argument to the `GetFrequency` method on CFunction objects of type `Function_Type_Uncertain`. Such functions have a computed ("output") probability distribution, sampled during a Monte Carlo simulation. The `GetFrequency` method returns an array whose elements count the frequency of occurrence of different values computed for the function during the simulation.

Syntax:

C / C++

```
public enum _Frequency_Type
```

VB .NET / C#

```
public enum Frequency_Type
```

Members:

Member Name	Code	Description
`Frequency_Type_Density`	0	Probability density function values are returned.
`Frequency_Type_Cumulative`	1	Cumulative (low to high) frequency values are returned.
`Frequency_Type_RevCumulative`	2	Reverse cumulative (high to low) frequency values are returned.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Simulate_Status

Enumeration for specifying the status of a simulation.

Syntax:

C / C++

```
public enum _Simulate_Status
```

VB .NET / C#

```
public enum Simulate_Status
```

Members:

Member Name	Code	Description
Simulate_Status_NotExecuted	-10	Simulation not executed.
Simulate_Status_Complete	0	Completed status.
Simulate_Status_Lic_Problem	-1	Missing or invalid simulation license problem.
Simulate_Status_User_Abort	6	Simulation aborted by user.
Simulate_Status_Memory_Dearth	11	Memory dearth status. Returned when the system cannot allocate enough memory to perform the simulation.
Simulate_Status_Interpret_Err	12	Interpretation error (Parse, Diagnostics, or Executor error.
Simulate_Status_Exception	999	Exception occurred status. Returned when an exception has been caught by try/catch.
Simulate_Status_Custom_Base	1000	Custom base status. (Base for Solver engine custom results.) Intended for future use.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Optimize_Status

Enumeration for specifying the status of an optimization.

Syntax:

C / C++

```
public enum _Optimize_Status
```

VB .NET / C#

```
public enum Optimize_Status
```

Members:

Member Name	Code	Description
Optimize_Status_NotExecuted	-10	Optimization is not executed
Optimize_Status_Invalid	-3	Invalid status. Can be returned for API_QQ_RESTART_SOLVE.
Optimize_Status_Ignore	-2	Ignore status. Used when dummy result code needs to be overridden.
Optimize_Status_Lic_Problem	-1	Invalid license status. (License expired, missing, invalid, etc.)
Optimize_Status_Optimal	0	Optimal solution has been found.
Optimize_Status_Converged	1	The Solver has converged to the current solution.
Optimize_Status_No_Remedies	2	"No remedies" status. (All remedies failed to find better point.)
Optimize_Status_Iterate_Limit	3	Iterates limit reached. Indicates an early exit of the algorithm.
Optimize_Status_Unbounded	4	Optimizing an unbounded objective function.
Optimize_Status_Unfeasible	5	Feasible solution could not be found.
Optimize_Status_User_Abort	6	Optimization aborted by user. Indicates an early exit of the algorithm.

Optimize_Status_Linear_Invalid	7	Invalid linear model. Returned when a linearity assumption renders incorrect.
Optimize_Status_Bad_Dataset	8	Bad data set status. Returned when a problem data set renders inconsistent.
Optimize_Status_Float_Err	9	Float error status. (Internal float error.)
Optimize_Status_Time_Out	10	Time out status. Returned when the maximum allowed time has been exceeded. Indicates an early exit of the algorithm.
Optimize_Status_Memory_Dearth	11	Memory dearth status. Returned when the system cannot allocate enough memory to perform the optimization.
Optimize_Status_Interpret_Err	12	Interpretation error. (Parser, Diagnostics, or Executor error.)
Optimize_Status_Api_Err	13	Fatal API error. (API not responding.)
Optimize_Status_Incumb_Cand	14	The Solver has found an integer solution within integer tolerance.
Optimize_Status_Bbnode_Limit	15	Branching and bounding node limit reached. Indicates an early exit of the algorithm.
Optimize_Status_Bbmips_Limit	16	Branching and bounding maximum number of incumbent points reached. Indicates an early exit of the algorithm.
Optimize_Status_Probable	17	Probable global optimum reached. Returned when MSL (Bayesian) global optimality test has been satisfied.
Optimize_Status_Bounds_Missing	18	Missing bounds status. Returned for EV/MSL Require Bounds when bounds are missing.
Optimize_Status_Bounds_Conflict	19	Bounds conflict status. Indicates ≤ = > = bounds conflict with existing binary or alldifferent constraints.
Optimize_Status_Bounds_Inconsist	20	Bounds inconsistency status.

		Returned when the lower bound value of a variable is grater than the upper bound value, i.e. *lb[i]* > *ub[i]* for some variable bound *i*.
`Optimize_Status_Derivative_Err`	21	Derivative error. Returned when `API_Jacobian` has not been able to compute gradients.
`Optimize_Status_Cone_Overlap`	22	Cone overlap status. Returned when a variable appears in more than one cone.
`Optimize_Status_Exception`	999	Exception occurred status. Returned when an exception has been caught by try/catch top-level.
`Optimize_Status_Custom_Base`	1000	Custom base status. (Base for Solver Engine custom results.)

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Integer_Type

Enumeration for specifying the type of an integer variable.

Syntax:

C / C++

```
public enum _Integer_Type
```

VB .NET / C#

```
public enum Integer_Type
```

Members:

Member Name	Code	Description
Integer_Type_None	0	No integer type specified (Default.) The variable is continuous, i.e. *3.6*, etc.
Integer_Type_Integer	1	Standard integer type. The Variable is integer, i.e. *-5*, *-3*, *2*, *12*, etc..
Integer_Type_Binary	2	Integer of binary type. Variable is binary, i.e. exactly *0* or *1*.
Integer_Type_Alldifferent	3	Integer of alldifferent type. The variable is part of an alldifferent group. If N variables are part of the same alldifferent group, it means that their value will be from 1 though N, or any permutation thereof. For example, if *N=3*, the three variables will be *1, 2, 3* or *1, 3, 2* or *2, 1, 3* or *2, 3, 1* or *3, 1, 2* or *3, 2, 1*.
Integer_Type_Mask	0xFFF	Mask for integer types.
Integer_Type_Semicont	0x8000	Flag that marks variable is semicontinuous.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Cone_Type

Enumeration for specifying the cone type.

Syntax:

C / C++

```
public enum _Cone_Type
```

VB .NET / C#

```
public enum Cone_Type
```

Members:

Member Name	Code	Description
Cone_Type_None	0	The variable is not part of a cone constraint.
Cone_Type_SOC	1	The variable is member of a cone.
Cone_Type_SOC_Start	2	The variable is head of the cone.
Cone_Type_SRC	3	The variable is part of a rotated cone.
Cone_Type_SRC_Start	4	The variable is head of rotated cone.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Cons_Rel

Enumeration for specifying the relation between the left-hand side and the right-hand side of the constraints.

There are two ways to specify constraints:

1. By setting the upper and lower bound on the constraint functions, or
2. By passing only the right-hand side and the relation, where relation can be: ≤ , = or ≥ .

Syntax:

C / C++

```
public enum _Cons_Rel
```

VB .NET / C#

```
public enum Cons_Rel
```

Members:

Member Name	Code	Description
Cons_Rel_LE	0	Less than or Equal to.
Cons_Rel_EQ	1	Equal to.
Cons_Rel_GE	2	Greater than or Equal to.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum IIS_Status

Enumeration for specifying the status of the Irreducibly Infeasible Subset (IIS) of constraints.

Syntax:

C / C++

```
public enum _IIS_Status
```

VB .NET / C#

```
public enum IIS_Status
```

Members:

Member Name	Code	Description
IIS_Status_LowerBound	0	Lower bound status. Indicates that the lower bound of a variable contributes to the infeasibility of the optimization problem.
IIS_Status_Fixed	1	Fixed status. Indicates that both the upper and lower bounds of a variable contribute to the infeasibility of the optimization problem.
IIS_Status_UpperBound	2	Upper bound status. Indicates that the upper bound of a variable contributes to the infeasibility of the optimization problem.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum License_SelectorCode

Enumeration for specifying the selector code for the license.

Syntax:

C / C++

```
public enum _License_SelectorCode
```

VB .NET / C#

```
public enum License_SelectorCode
```

Members:

Member Name	Code	Description
License_SelectorCode_PROM	1	The license is locked to the host ID PROM.
License_SelectorCode_IPAddr	2	The license is locked to the host IP address.
License_SelectorCode_DiskID	4	The license is locked to the host Disk ID.
License_SelectorCode_HostName	8	The license is locked to the host name.
License_SelectorCode_Ethernet	16	The license is locked to the host Ethernet Address.
License_SelectorCode_NV_IPX	32	The license is locked to the host Netware IPX.
License_SelectorCode_NV_Serial	64	The license is locked to the host Netware Serial.
License_SelectorCode_PortableServ	128	The license is locked to the Computer ID key.
License_SelectorCode_Custom	256	Custom code locking criteria. Not currently used.
License_SelectorCode_CPU	512	The license is locked to Processor ID.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum DistParam_Type

Enumeration for specifying the type of the distribution parameters.

Syntax:

C / C++

```
public enum _DistParam_Type
```

VB .NET / C#

```
public enum DistParam_Type
```

Members:

Member Name	Code	Description
DistParam_None	0	None.
DistParam_LowerCut	1	LowerCut distribution parameter.
DistParam_UpperCut	2	UpperCut distribution parameter.
DistParam_Shift	3	Shift distribution parameter.
DistParam_Location	4	Location distribution parameter.
DistParam_Scale	5	Scale distribution parameter.
DistParam_Shape	6	Shape distribution parameter.
DistParam_Alpha	7	Alpha distribution parameter.
DistParam_Alpha1	8	Alpha1 distribution parameter.
DistParam_Alpha2	9	Alpha2 distribution parameter.
DistParam_Beta	10	Beta distribution parameter.
DistParam_Gamma	11	Gamma distribution parameter.
DistParam_Lambda	12	Lambda distribution parameter.
DistParam_Mu	13	Mu distribution parameter.
DistParam_Nu	14	Nu distribution parameter.
DistParam_Sigma	15	Sigma distribution parameter.
DistParam_Theta	16	Theta distribution parameter.
DistParam_A	17	"A" distribution parameter.

DistParam_B	18	B distribution parameter.
DistParam_C	19	C distribution parameter.
DistParam_D	20	D distribution parameter.
DistParam_M	21	M distribution parameter.
DistParam_N	22	N distribution parameter.
DistParam_Q	23	Q distribution parameter.
DistParam_S	24	S distribution parameter.
DistParam_Lower	25	Lower distribution parameter.
DistParam_Upper	26	Upper distribution parameter.
DistParam_Bottom	27	Bottom distribution parameter.
DistParam_Top	28	Top distribution parameter.
DistParam_BotPerc	29	Bottom percentile distribution parameter.
DistParam_TopPerc	30	Top percentile distribution parameter.
DistParam_Min	31	Minimum distribution parameter.
DistParam_Max	32	Maximum distribution parameter.
DistParam_Likely	33	Likely distribution parameter.
DistParam_Mean	34	Mean distribution parameter.
Stdev Distribution Parameter	35	Standard deviation distribution parameter.
DistParam_Var	36	Var distribution parameter.
DistParam_Rate	37	Rate distribution parameter.
DistParam_DegFr	38	DegFr distribution parameter.
DistParam_Data	39	Data distribution parameter.
DistParam_Value	40	Value distribution parameter.
DistParam_Prob	41	Prob distribution parameter.
DistParam_Trials	42	Trials distribution parameter.
DistParam_SamSize	43	Sample size distribution parameter.

`DistParam_PopSize`	44	Population size distribution parameter.
`DistParam_Items`	45	Items distribution parameter.
`DistParam_Successes`	46	Successes distribution parameter.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Dist_VarFcn

Enumeration for specifying whether the call applies to an uncertain variable or to an uncertain function.

Syntax:

C / C++

```
public enum _Dist_VarFcn
```

VB .NET / C#

```
public enum Dist_VarFcn
```

Members:

Member Name	Code	Description
Dist_Var	0	Specifies an uncertain variable.
Dist_Fcn	1	Specifies an uncertain function.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum DistFit_SortType

Specifies the sorting criteria used in AutoFitAll method of CDistribution class.

Syntax:

C / C++

```
public enum _DistFit_SortType
```

VB .NET / C#

```
public enum DistFit_SortType
```

Members:

Member Name	Code	Description
DistFit_SortCS	0	Distribution fitting Chi-Square statistics.
DistFit_SortKS	1	Distribution fitting Komogorov-Smirnov statistics.
DistFit_SortAD	2	Distribution fitting Anderson-Darling statistics.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Check_Type

Used in method DependCheck() of Model class.

Syntax:

C / C++

```
public enum _Check_Type
```

VB .NET / C#

```
public enum Check_Type
```

Members:

Member Name	Code	Description
CheckType_Gradients	1	Check for Gradients
CheckType_Structure	2	Check for Structure
CheckType_Convexity	3	Check for Convexity
CheckType_Default	4	If using a linear engine, the default will be CheckType_Structure. If using a nonlinear engine, the default will be CheckType_Convexity.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Convex_Type

Returns the Problem's convexity type. Used only when loading an Excel workbook.

Syntax:

C / C++

```
public enum _Convex_Type
```

VB .NET / C#

```
public enum Convex_Type
```

Members:

Member Name	Code	Description
Convex_Type_NA	1	The initial state, before analysis is performed.
Convex_Type_Unknown	2	Convexity type unknown
Convex_Type_Convex	3	Problem is convex.
Convex_Type_NonConvex	4	Problem is nonconvex.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Stochastic_Type

Specifies if the Problem is stochastic or not. Used only when loading an Excel workbook.

Syntax:

C / C++

```
public enum _Stochastic_Type
```

VB .NET / C#

```
public enum Stochastic_Type
```

Members:

Member Name	Code	Description
Stochastic_Type_NA	0	The initial state, before analysis is performed.
Stochastic_Type_NonStochastic	1	Problem is not stochastic.
Stochastic_Type_Stochastic	2	Problem is stochastic.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Solve_Type

Specifies the solve type in the Optimize() call.

Syntax:

C / C++

```
public enum _Solve_Type
```

VB .NET / C#

```
public enum Solve_Type
```

Members:

Member Name	Code	Description
Solve_Type_Solve	0	Solve the complete problem.
Solve_Type_Analyse	1	Analyse without solving.
Solve_Type_NoIntegers	2	Solve without integer constraints.
Solve_Type_SolveForRecourse	3	Solve for recourse variables.
Solve_Type_SimOpt	4	Solve with simulation optimization.
Solve_Type_Cluster	5	Not implemented in the Solver SDK Platform.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Simulate_Type

Specifies the simulation type in the Simulate() call.

Syntax:

C / C++

```
public enum _Simulate_Type
```

VB .NET / C#

```
public enum Simulate_Type
```

Members:

Member Name	Code	Description
Simulate_Type_Simulate	0	Run a simulation.
Simulate_Type_Cluster	1	Not implemented in the Solver SDK Platform.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Chance_Type

The chance type of a function of type chance. Can only be changed/used if loading an Excel workbook.

Syntax:

C / C++

```
public enum _Chance_Type
```

VB .NET / C#

```
public enum Chance_Type
```

Members:

Member Name	Code	Description
Chance_Type_None	0	Normal/deterministic objective/constraint.
Chance_Type_Expval	1	Expected value – used only for objective.
Chance_Type_VaR	2	Value at Risk (VaR) criterion.
Chance_Type_CVar	3	Conditional Value at Risk (CVaR) criterion.
Chance_Type_USet	4	Uncertainty Set (USet) criterion

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

enum Parameter_Type

Specifies the type of the parameter.

Syntax:

C / C++

```
public enum _Parameter_Type
```

VB .NET / C#

```
public enum Parameter_Type
```

Members:

Member Name	Code	Description
Parameter_Type_Sensitivity	0	Sensitivity parameter.
Parameter_Type_Optimization	1	Optimization parameter.
Parameter_Type_Simulation	2	Simulation parameter.

Requirements:

Namespace: SolverPlatform

C, C++, Fortran, Pascal

Library: SolverSDK.lib

.NET 1.1

Assembly: SolverSDKNet11.dll

.NET 2.0

Assembly: SolverSDKNet20.dll

.NET 4.0

Assembly: SolverSDKNet40.dll

Java

Class library: SolverSDK.jar

Linux/Unix

Shared library: libSolverSDK.so

Solver SDK Platform Procedural API Reference

Click to navigate to the desired section:

- Method and Property Functions Overview
- Problem Method Functions
- Problem Property Functions
- Engine Method Functions
- Engine Property Functions
- Evaluator Method Functions
- Model Method Functions
- Model Property Functions
- Variable Method Functions
- Variable Property Functions
- Function Method Functions
- Function Property Functions
- Engine Method Functions
- Functions Error Codes

Method and Property Functions Overview

Problem Method Functions

Function Name	Description
SolverProblem	Creates new Problem.
SolverProbInit	Creates and initializes a new Problem.
SolverProbFree	Destroys Problem and frees memory.
SolverProbLoad	Creates a Problem by reading it in from a file.
SolverProbSave	Saves Problem to file.
SolverProbTrialPeriod	Returns the number of days remaining before the expiration date of the Solver Platform license.

Problem Property Functions

Function Name	Description
SolverProbTypeGet	Returns current Problem type and/or current Solver type.
SolverProbTypeSet	Sets current Problem type.
SolverProbNameSet	Sets current Problem name.
SolverProbNameGet	Returns current Problem name.
SolverProbEngineGet	Returns the current Engine. ("Current Engine" the currently selected Engine to be used in the optimization or simulation.)
SolverProbEngineSet	Sets Engine as current. ("Current Engine" the currently selected Engine to be used in the optimization or simulation.)

Solver Method Functions

Function Name	Description
SolverSimulate	Performs simulation.
SolverOptimize	Performs optimization.
SolverOptIISFind	Returns an Irreducibly Infeasible Set for an infeasible optimization problem. If there is more than one IIS (which is the most common scenario), the method function finds

	and reports the first one found.
SolverReport	Creates report file for a Problem.

Solver Property Functions

Function Name	Description
SolverNumSimulationsGet	Returns the number of simulations run.
SolverNumSimulationsSet	Sets the number of simulations.
SolverNumTrialsGet	Returns the number of trials per simulation.
SolverNumTrialsSet	Sets the number of trials per simulation.
SolverSimulateStatus	Returns the status of a simulation.
SolverOptimizeStatus	Returns the status of an optimization.
SolverOptIIS	Returns the parameters of the Irreducibly Infeasible Set.
SolverNumSolutionsGet	Returns the number of solutions found.
SolverNumSolutionsSet	Sets the number of solutions to be found.
SolverSolutionsIndexGet	Retrieves the specified solution.
SolverSolutionIndexSet	Sets the solution as current.
SolverFunctionCurrent	Returns the name of the current function vector.
SolverFunctionType	Returns the type of the current function vector.
SolverFunctionIndex	Returns the index of the current function inside the current function vector.

Engine Method Functions

Function Name	Description
SolverEngineAdd	Adds Engine to the Engine Collection.
SolverEngineRemove	Removes Engine from the Engine Collection.
SolverEngParamLoad	Loads Engine parameters from file.
SolverEngParamSave	Saves Engine parameters to file.
SolverEngParamReset	Resets Engine parameters to their default values.

Engine Property Functions

Function Name	Description
SolverEngCount	Returns the number of Solver Engines in the Engines Collection.
SolverEngName	Returns the string name of a Solver Engine.
SolverEngType	Returns the Solver type and/or Problem type for the current Engine.
SolverEngParamCount	Returns the number of parameters in the parameters Collection of an Engine.
SolverEngParamName	Returns the name of a parameter from the parameters Collection of an Engine.
SolverEngParamGet	Returns the current value of a Solver Engine parameter.
SolverEngParamSet	Sets the value of a Solver Engine parameter.
SolverEngParamInfo	Returns the default / minimum / maximum value of a Solver Engine parameter.
SolverEngLimit	Returns the limits of a Solver Engine.
SolverEngStat	Retrieves performance statistics for a Solver Engine

Evaluator Method Functions

Function Name	Description
SolverEvaluatorAdd	Adds Evaluator.
SolverEvaluatorRemove	Removes Evaluator.

Model Method Functions

Function Name	Description
SolverModArrayOrder	Sets the array ordering for the matrices in the model (by rows or by columns).
SolverModDependTest	Determines the _Problem_Type of a model, and fills in the linear matrix coefficients when appropriate.

Model Property Functions

Function Name	Description
SolverModNumVariables	Retrieves the number of variables of specified type.

SolverModNumFunctions	Retrieves the number of functions of specified type.
SolverModNumAllGrad	Retrieves the number of nonzero gradient elements in functions.
SolverModNumFcnHess	Retrieves the number of nonzero Hessian elements in a function.
SolverModNumDistArguments	Retrieves the number of nonzero elements in DistArguments matrix.
SolverModNumCorrelations	Retrieves the number of nonzero elements in DistCorrelations matrix.
SolverModAllLinearGet	Returns the constant gradient elements (LP coefficients) of functions with respect to variables of specified types _Array_Order).
SolverModAllLinearSet	Sets the constant gradient elements (LP coefficients) of functions with respect to variables of specified types _Array_Order).
SolverModAllSpLinearGet	Returns constant gradient non-zero elements (LP coefficients) of functions with respect to variables of specified types.
SolverModAllSpLinearSet	Sets constant gradient elements (LP coefficients) of functions with respect to variables of specified types.
SolverModAllGradDependGet	Returns the nature of dependence of functions with respect to the variables of specified types.
SolverModAllGradDependSet	Sets the nature of dependence on variables of selected types.
SolverModAllGradientGet	Returns all gradient values, dense.
SolverModAllGradientSet	Sets all gradient values, dense.
SolverModAllSpGradientGet	Returns all gradient values, sparse.
SolverModAllSpGradientSet	Sets all gradient values, sparse.
SolverModFcnHessianGet	Returns Hessian values for a function, dense.
SolverModFcnHessianSet	Sets Hessian values for a function, dense.
SolverModFcnSpHessianGet	Returns Hessian values for a function, sparse.
SolverModFcnSpHessianSet	Sets Hessian values for a function, sparse.
SolverModFcnQuadraticGet	Returns the constant Hessian matrix elements, with respect to variables of specified types for a given function.

SolverModFcnQuadraticSet	Sets the constant Hessian matrix elements, with respect to variables of specified types for a given function.
SolverModFcnSpQuadraticGet	Returns the Constant Hessian matrix elements, with respect to variables of specified types.
SolverModFcnSpQuadraticSet	Sets the Constant Hessian matrix elements, with respect to variables of specified types.
SolverModFcnHessDependGet	Returns the Hessian dependence type.
SolverModFcnHessDependSet	Sets the Hessian dependence type.
SolverModDistTypeGet	Returns the type of distribution for uncertain variables.
SolverModDistTypeSet	Sets the type of distribution for uncertain variables.
SolverModDistArgumentsGet	Returns the argument values for distributions of uncertain variables.
SolverModDistArgumentsSet	Sets the argument values for distributions of uncertain variables.
SolverModDistSpArgumentsGet	Returns argument values for distributions of uncertain variables.
SolverModDistSpArgumentsSet	Sets argument values for distributions of uncertain variables.
SolverModDistCorrelationsGet	Returns the correlation coefficient between uncertain variables.
SolverModDistCorrelationsSet	Sets the correlation coefficient between uncertain variables
SolverModVarIndexName	Returns the name of a variable range. (Not implemented yet.)
SolverModFcnIndexName	Returns the name of a function range. (Not implemented yet.)
SolverModNameVarIndex	Returns the index of a variable and the number of variables in a range. (Not implemented yet.)
SolverModNameFcnIndex	For future use - not implemented yet.

Variable Method Functions

Function Name	Description
SolverVariableAdd	Adds a vector of variables of a given type to the Variables Collection.
SolverVariableRemove	Removes a vector of variables from the Variables

	Collection.
SolverVarGetCorrelation	Returns the correlation coefficients between elements of a variable vector and other uncertain variable.
SolverVarGetFrequency	Returns the frequency of observations in each bin.
SolverVarNonNegative	Declares a lower bound of 0 for variables that do not have a lower bound.

Variable Property Functions

Function Name	Description
SolverVarCount	Returns the number of variable vectors in the Variables Collection.
SolverVarNameGet	Returns the name of a vector of variables.
SolverVarNameSet	Sets the name of a vector of variables.
SolverVarTypeGet	Returns variable type for a vector of variables.
SolverVarTypeSet	Sets variable type for a vector of variables.
SolverVarValue	Returns the values of a vector of variables.
SolverVarSizeGet	Returns the number of elements in variable vector.
SolverVarSizeSet	Sets the number of elements in variable vector.
SolverVarPosition	Returns the position of the first element in Model flat index space.
SolverVarStatistics	Retrieves various statistics for a vector of variables.
SolverVarPercentiles	Returns the 5th, 10th, ..., 95th percentiles for a vector of variables.
SolverVarAllTrialsGet	Returns all trial values of variables for a vector of variables.
SolverVarAllTrialsSet	Sets all trial values of variables for a vector of variables.
SolverVarLowerBoundGet	Returns the lower bounds on variables for a vector of variables.
SolverVarLowerBoundSet	Sets the lower bounds on variables for a vector of variables.
SolverVarUpperBoundGet	Returns the upper bounds on variables for a vector of variables.
SolverVarUpperBoundSet	Sets the upper bounds on variables for a vector of variables.
SolverVarIntegerTypeGet	Returns the type of the integer variables in a vector of

	variables.
SolverVarIntegerTypeSet	Sets the type of the integer variables in a vector of variables.
SolverVarGroupIndexGet	Returns the index number of the alldifferent group that a variable belongs to.
SolverVarGroupIndexSet	Sets the index number of the alldifferent group that a variable belongs to.
SolverVarConeTypeGet	Returns the cone type for a vector of variables.
SolverVarConeTypeSet	Sets the cone type for a vector of variables.
SolverVarConeIndexGet	Returns the index number of the cone that a variable belongs to.
SolverVarConeIndexSet	Sets the index number of the cone that a variable belongs to.
SolverVarInitialValueGet	Returns the initial values of variables.
SolverVarInitialValueSet	Sets the initial values of variables for NLP.
SolverVarFinalValue	Returns the final values for variables for a vector of variables.
SolverVarDualValue	Returns the final dual values for variables for a vector of variables.
SolverVarDualLower	Returns, for a vector of variables, the lower bound of the range over which variable dual values are valid.
SolverVarDualUpper	Returns, for a vector of variables, the upper bound of the range over which variable dual values are valid.

Mathematical Method Functions

Function Name	Description
SolverFunctionAdd	Adds a vector of functions of a given type to the Functions Collection.
SolverFunctionRemove	Removes a vector of functions from the Functions Collection.
SolverFcnGetCorrelation	Returns the correlation coefficients between uncertain function and elements of a specified function vector.
SolverFcnGetFrequency	Returns the boundary and frequency of observations of solution trial points in each bin.
SolverFcnNonNegative	Sets the lower bound of functions to 0, if not already given.
SolverFcnRelation	Returns an array with the relation types of the constraints

for the function vector.

Mathematical Property Functions

Function Name	Description
SolverFcnCount	Returns the number of function vectors in the Functions Collection.
SolverFcnNameGet	Returns the name of vector of functions.
SolverFcnNameSet	Sets the name of vector of functions.
SolverFcnTypeGet	Returns the type of vector of functions.
SolverFcnTypeSet	Sets the type of vector of functions.
SolverFcnValueGet	Returns the values of vector of functions.
SolverFcnValueSet	Sets the value of vector of functions.
SolverFcnSizeGet	Returns the number of elements in function vector.
SolverFcnSizeSet	Sets the number of elements in function vector.
SolverFcnPosition	Retrieves the position of the first element in Model flat index space.
SolverFcnStatistics	Retrieves various statistics for a given function vector. sample.
SolverFcnPercentiles	Returns the 5th, 10th, ..., 95th percentiles for a vector of functions.
SolverFcnAllTrials	Returns the trial values for functions of vector of functions.
SolverFcnLowerBoundGet	Returns the lower bounds on functions for a vector of functions.
SolverFcnLowerBoundSet	Sets the lower bounds on functions for a vector of functions.
SolverFcnUpperBoundGet	Returns the upper bounds on functions for a vector of functions.
SolverFcnUpperBoundSet	Sets the upper bounds on functions for a vector of functions.
SolverFcnFinalValue	Returns the final primal values for functions of function vector.
SolverFcnDualValue	Returns the final dual values for constraint functions of function vector.
SolverFcnDualLower	Returns, for a vector of functions, the lower bound of range over which constraint dual values are valid.

SolverFcnUpperLower	Returns, for a vector of functions, the upper bound of range over which constraint dual values are valid.

Environment Method Functions

Function Name	Description
SolverEnvInit	Initializes the environment variable.
SolverEnvFree	Frees a previously created environment variable.
SolverEnvRuntimeIDGet	Retrieves the runtime ID of the environment variable.
SolverEnvRuntimeIDSet	Sets the runtime ID of the environment variable.
SolverEnvLockCodeGet	Retrieves the lock code of the machine running the SDK.
SolverEnvLicenseGet	Retrieves the Solver Platform usage license.
SolverEnvLicenseSet	Sets the Solver Platform usage license by adding a license string to the license file.
SolverEngTrialPeriod	Returns the number of days remaining before the expiration date of the license for a particular Engine.

Probability Distribution Method Functions

Function Name	Description
SolverModDistAutoFit	Automatically fits a function or a variable sample to the most suitable distribution.
SolverModDistCDF	Returns the Cumulative Distribution Function value for a function or variable.
SolverModDistCDFInv	Returns the Inverse Cumulative Distribution function value for a function or variable.
SolverModDistPDF	Returns the Probability Density Function value for a function or variable.

Simulation Distribution Arguments Functions

Function Name	Description
SolverModDistControlParamsGet	Returns the control parameters for the distribution of a function or variable.
SolverModDistControlParamsSet	Sets the control parameters for the distribution of a function or variable.

Fitting Method Functions

Function Name	Description
SolverModDistFit	Fits a function or a variable sample to a given distribution.
SolverModDistInitSample	Initializes a sample for the distribution of a function or a variable.

Fitting Quality Criteria Functions

Function Name	Description
SolverModDistFitLimit	Retrieves the critical fitting value adjusted to the sample size. If not exceeded by the computed test value, fitting is successful.
SolverModDistFitSigLevel	Retrieves the significance level achieved to a particular distribution type as a result of a fitting operation.
SolverModDistFitStatistic	Retrieves the fitting statistic dependent on the sample's size.
SolverModDistFitSuccess	Returns a Boolean value indicating if the distribution fit was successful.
SolverModDistFitTest	Retrieves statistics adjusted to the sample’s size.
SolverModDistIndepTestGet	Retrieves a value indicating the appropriateness of the fitting.

Initialization Method Function

Function Name	Description
SolverModDistInit	Initializes the distribution for a function or a variable.

Distribution Parameter Functions

Function Name	Description
SolverModDistKurtosis	Returns the analytic kurtosis for the distribution of a function or variable.
SolverModDistMean	Returns the analytic mean value for the distribution of a function or variable.
SolverModDistMedian	Returns the analytic median value for the distribution of a function or variable.
SolverModDistMode	Returns the analytic mode value for the distribution of a

	function or variable.
SolverModDistSkewness	Returns the analytic skewness for the distribution of a function or a variable.
SolverModDistVariance	Returns the analytic variance for the distribution of a function or a variable.

Distribution Arguments Functions

Function Name	Description
SolverModDistNumControlParams	Returns the number of control parameters for the distribution of a function or variable.
SolverModDistNumParams	Returns the number of parameters for the distribution of a function or variable.
SolverModDistParamsGet	Returns the parameters for the distribution of a function or variable.
SolverModDistParamsSet	Sets the parameters for the distribution of a function or variable.

Sample Management Functions

Function Name	Description
SolverModDistSampleGet	Retrieves the sample for a distribution of a function or variable.
SolverModDistSampleSet	Sets the sample for a distribution of a function or variable.
SolverModDistSizeGet	Returns the size of the sample for a distribution of a function or a variable.
SolverModDistSizeSet	Sets the size of the sample for a distribution of a function or a variable.

Problem Method Functions

Click to navigate to the desired topic:

- SolverProblem
- SolverProbInit
- SolverProbFree
- SolverProbLoad
- SolverProbSave
- SolverProbTrialPeriod

SolverProblem

SolverProblem() is a method function that creates new Problem.

Syntax:

```
SolverProblem(HProblem* pprob)
```

Parameters:

Parameter Name	Parameter Description
pprob	Returns the problem instance handle.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverProbInit

SolverProbInit() is a method function that creates and initializes a new Problem.

Syntax:

```
SolverProbInit (HProblem* pprob, _Solver_Type solvertype, int
               numvars, int numfcns)
```

Parameters:

Parameter Name	Parameter Description
pprob	Returns the problem instance handle.
solvertype	Specifies either optimization or simulation via the _Solver_Type enum.
numvars	Number of variables.
numfcns	Number of functions.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverProbFree

SolverProbFree() is a method function that destroys a Problem and frees memory.

Syntax:

```
SolverProbFree (HProblem prob)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverProbLoad

SolverProbLoad() is a method function that creates a Problem by reading it in from a file.

Syntax:

```
SolverProbLoad (HProblem* pprob, string problemspecfile,
                _File_Format fileformat)
```

Parameters:

Parameter Name	Parameter Description
`pprob`	Returns the problem instance handle.
`problemspecfile`	Filename/path (or worksheet name string) from which problem info will be read.
`fileformat`	Specifies the file format via the `_File_Format` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverProbSave

SolverProbSave() is a method function that saves a Problem to a file.

Syntax:

```
SolverProbSave (HProblem prob, string problemspecfile,
                _File_Format fileformat)
```

Parameters:

Parameter Name	Parameter Description
`prob`	The problem instance handle.
`problemspecfile`	Filename/path (or worksheet name string) to which problem info will be written.
`fileformat`	Specifies the file format via the `_File_Format` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverProbTrialPeriod

SolverProbTrialPeriod() is a function that returns the number of days remaining before the expiration date of the Solver Platform license.

Syntax:

```
SolverProbTrialPeriod(void* prob, int* trialdaysleft)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`trialdaysleft`	Number of seconds until the Solver SDK Platform trial license expires.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Problem Property Functions

Click to navigate to the desired topic:

- SolverProbTypeGet
- SolverProbTypeSet
- SolverProbNameGet
- SolverProbNameSet
- SolverProbEngineGet
- SolverProbEngineSet

SolverProbTypeGet

SolverProbTypeGet() is a property function that returns the current Problem type and/or the current Solver type.

Syntax:

```
SolverProbTypeGet (HProblem prob, _Solver_Type* psolvertype,
                   _Problem_Type* pproblemtype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`psolvertype`	Specifies, via the `_Solver_Type` enum, the Solver type.
`pproblemtype`	Specifies, via the `_Problem_Type` enum, the Problem type.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverProbTypeSet

SolverProbTypeSet() is a property function that sets the current Problem type.

Syntax:

```
SolverProbTypeGet (HProblem prob, _Solver_Type solvertype,
                   _Problem_Type problemtype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`solvertype`	Specifies, via the `_Solver_Type` enum, Solver the type.
`problemtype`	Specifies, via the `_Problem_Type` enum, the Problem type.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverProbNameGet

SolverProbNameGet() is a property function that returns the current Problem name.

Syntax:

```
SolverProbNameGet (HProblem prob, string* pname)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
pname	Name of the Problem.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverProbNameSet

SolverProbNameSet() is a property function that sets the current Problem name.

Syntax:

```
SolverProbNameSet (HProblem prob, string name)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the Problem.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverProbEngineGet

SolverProbEngineGet() is a method function that returns the current Engine.

Syntax:

```
SolverEngineGet (HProblem prob, string *enginename)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	Returns the current Solver Engine name.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverProbEngineSet

SolverProbEngineSet() is a method function that sets an Engine as current.

Syntax:

```
SolverEngineEngineSet (HProblem prob, string enginename)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	New current Solver Engine name.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

Solver Method Functions

Click to navigate to the desired topic:

- SolverSimulate
- SolverOptimize
- SolverOptIISFind
- SolverReport
- SolverAnalyze
- SolverSolveRecourse
- SolverSimOpt

SolverSimulate

SolverSimulate() is a method function that performs simulation and returns its status.

Syntax:

```
SolverSimulate (HProblem prob, _Simulate_Status* psimstatus)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`psimstatus`	Status of the simulation presented via the `_Simulate_Status` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverOptimize

SolverOptimize() is a method function that performs optimization and returns its result and the final value of the objective function.

Syntax:

```
SolverOptimize (HProblem prob, _Optimize_Status* poptstatus,
                double* pobjfinalvalue)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`poptstatus`	Result of the optimization presented via the `_Optimize_Status` enum.
`pobjfinalvalue`	Final value of the objective function.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverOptIISFind

SolverOptIISFind() is a method function that returns an Irreducibly Infeasible Set for an infeasible optimization problem.

Syntax:

```
SolverOptIISFind (HProblem prob, int* piisnumcons, int*
                  piisnumbnds)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
piisnumcons	Pointer to a variable which will hold the number of constraints included in the IIS.
piisnumbnds	Pointer to a variable which will hold the number of variable bounds included in IIS.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverReport

SolverReport() is a method function that creates a report file for a Problem.

Syntax:

```
SolverReport (HProblem prob, string reportspecfile, string
              reportname, int reportoutline)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
reportspecfile	Filename/path (or worksheet name string) to contain report information.
reportname	Report name (e.g. "Answer", "Sensitivity", "Limits", "Population", "Feasibility", "Feasibility-Bounds").
reportoutline	Specifies whether the report will be outlined. The value is either 0 or 1: 0 = don't outline; 1 = outline this report.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverAnalyze

SolverAnalyze() is a method function that diagnoses the problem type of the Problem.

Syntax:

```
SolverAnalyze(HProblem prob, _Problem_Type* problemtype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`problemtype`	Problem type, returned via the `_Problem_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverSolveRecourse

SolverSolveRecourse() is a method function that solves for the recourse variables in a stochastic optimization problem.

Syntax:

```
SolverSolveRecourse(HProblem prob, _Optimize_Status* poptstatus,
                    double* pobjfinalvalue)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`poptstatus`	Result of optimization returned via the `_Optimize_Status` enum.
`pobjfinalvalue`	Final value of the objective function.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverSimOpt

SolverSimOpt() is a method function that solves a simulation optimization Problem.

Syntax:

```
SolverSimOpt(HProblem prob, _Optimize_Status* poptstatus,
              double* pobjfinalvalue)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`poptstatus`	Result of optimization returned via the `_Optimize_Status` enum.
`pobjfinalvalue`	Final value of the objective function.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Solver Property Functions

Click to navigate to the desired topic:

- SolverNumSimulationsGet
- SolverNumSimulationsSet
- SolverNumTrialsGet
- SolverNumTrialsSet
- SolverSimulateStatus
- SolverOptimizeStatus
- SolverOptIIS
- SolverNumSolutionsGet
- SolverNumSolutionsSet
- SolverSolutionsIndexGet
- SolverSolutionsIndexSet
- SolverFunctionCurrent
- SolverFunctionType
- SolverFunctionIndex
- SolverOptIndexGet
- SolverOptIndexSet
- SolverNumOptimizationsGet
- SolverNumOptimizationsSet

SolverNumSimulationsGet

SolverNumSimulationsGet() is a property function that returns the number of simulations run.

Syntax:

```
SolverNumSimulationsGet (HProblem prob, int* pnumsimulations)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`pnumsimulations`	Number of simulations (if several being run).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverNumSimulationsSet

SolverNumSimulationsSet() is a property function that sets the number of simulations.

Syntax:

```
SolverNumSimulationsSet (HProblem prob, int numsimulations)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`numsimulations`	Number of simulations (if several being run).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverNumTrialsGet

SolverNumTrialsGet() is a property function that returns the number of trials per simulation.

Syntax:

```
SolverNumTrialsGet (HProblem prob, int* pnumtrials)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
pnumtrials	Number of trials per simulation.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverNumTrialsSet

SolverNumTrialsSet() is a property function that sets the number of trials per simulation.

Syntax:

```
SolverNumTrialsGet (HProblem prob, int numtrials)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
numtrials	Number of trials per simulation.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverSimulateStatus

SolverSimulateStatus() is a property function that returns the status of a simulation.

Syntax:

```
SolverSimulateStatus (HProblem prob, _Simulate_Status*
              psimstatus)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`psimstatus`	Result of simulation returned via the `_Simulate_Status` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverOptimizeStatus

SolverOptimizeStatus() is a property function that returns the status of a optimization.

Syntax:

```
SolverOptimizeStatus (HProblem prob, _Optimize_Status*
              poptstatus)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`poptstatus`	Result of optimization returned via the `_Optimize_Status` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverOptIIS

SolverOptIIS() is a property function that returns the parameters of the Irreducibly Infeasible Set (IIS).

Syntax:

```
SolverOptIIS (HProblem prob, int iisconindex[], _IIS_Status
              iisconstatus[], int iisbndindex[], _IIS_Status
              iisbndstatus[])
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`iisconindex [iisnumcons]`	Indices of the constraints included in IIS.
`iisconstatus [iisnumcons]`	Status of the constraints included in IIS presented via the `_IIS_Status` enum.
`iisbndindex [iisnumbnds]`	Indices of the variable bounds included in IIS.
`iisbndstatus [iisnumbnds]`	Status of the variable bounds included in IIS presented via the `_IIS_Status` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverNumSolutionsGet

SolverNumSolutionsGet() is a property function that returns the number of solutions found.

Syntax:

```
SolverNumSolutionsGet (HProblem prob, int* pnumsolutions)
```

Parameters:

Parameter Name	Parameter Description
Prob	Handle to the problem, passed by the user.
pnumsolutions	The number of solutions found.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverNumSolutionsSet

SolverNumSolutionsSet() is a property function that sets the number of solutions to be found.

Syntax:

```
SolverNumSolutionsSet (HProblem prob, int numsolutions)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`numsolutions`	The number of solutions to be found.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverSolutionsIndexGet

SolverSolutionsIndexGet() is a property function that retrieves the specified solution.

Syntax:

```
SolverSolutionsIndexGet (HProblem prob, int* psolutionindex)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`psolutionindex`	Index of the solution to be retrieved.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverSolutionsIndexSet

SolverSolutionsIndexSet() is a property function that sets the specified solution as current.

Syntax:

```
SolverSolutionsIndexSet (HProblem prob, int solutionindex)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`solutionindex`	Index number for the solution to be set as current.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverFunctionCurrent

SolverFunctionCurrent() is a property function that returns the name of the current function vector.

Syntax:

```
SolverFunctionCurrent (HProblem prob, LPCWSTR* pfunctioncurrent)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`pfunctioncurrent`	The name of the current function vector.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverFunctionType

SolverFunctionType() is a property function that returns the type of the current function vector.

Syntax:

```
SolverFunctionType (HProblem prob, _Function_Type*
              pfunctiontype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`pfunctiontype`	The type of the current function vector.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverFunctionIndex

SolverFunctionIndex() is a property function that returns the index of the current function inside the current function vector.

Syntax:

```
SolverFunctionIndex (HProblem prob, int* pfunctionindex)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
pfunctionindex	The index of the current function inside the current function vector.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverOptIndexGet

SolverOptIndexGet() is a method function that retrieves the index of the current optimization.

Syntax:

```
SolverOptIndexGet(HProblem prob, int* optindex)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`optindex`	Returns the index of the current optimization.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverOptIndexSet

SolverOptIndexSet() is a method function that sets the index of the current optimization.

Syntax:

```
SolverOptIndexSet(HProblem prob, int* optindex)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`optindex`	The index of the optimization to be set.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverNumOptimizationsGet

SolverNumOptimizationsGet() is a method function that retrieves the setting of the SolverNumOptimizationsSet() property.

Syntax:

```
SolverNumOptimizationsGet(HProblem prob, int* pnumoptimizations)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`pnumoptimizations`	Returns the setting of the SolverNumOptimizationsSet() property.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverNumOptimizationsSet

SolverNumOptimizationsSet() is a method function that sets the number of optimizations to be performed.

Syntax:

```
SolverNumSet(HProblem prob, int* optindex)
```

Parameters:

Parameter Name	Parameter Description
Prob	Handle to the problem, passed by the user.
Optindex	The index of the optimization to be set.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Engine Method Functions

Click to navigate to the desired topic:

- SolverEngineAdd
- SolverEngineRemove
- SolverEngineDiscover
- SolverEngParamLoad
- SolverEngParamSave
- SolverEngParamReset

SolverEngineAdd

SolverEngineAdd() is a method function that adds a new Engine to the Engine Collection.

Syntax:

```
SolverEngineAdd (HProblem prob, string enginename, string
                enginefilespec)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	Solver Engine standard name. If empty or NULL, the Solver Platform will use the name defined in the Engine dynamic library.
enginefilespec	Name/path of the Engine DLL.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngineRemove

SolverEngineRemove() is a method function that removes an Engine from the Engine Collection.

Syntax:

```
SolverEngineRemove (HProblem prob, string enginename)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	Solver Engine standard name.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngineDiscover

SolverEngineDiscover() is a method function that Searches an optionally given path for any external Engines and adds the Engines it will automatically add them to the Engine Collection. The method returns an integer, indicating how many Engines have been added. If no path or an empty string has been passed, it searches in the same folder where the SolverSDK.DLL resides.

Syntax:

```
SolverEngineDiscover(HProblem prob, LPCWSTR path)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`path`	Path to be searched for external Engines.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverEngParamLoad

SolverEngParamLoad() is a method function that loads Engine parameters from a file.

Syntax:

```
SolverEngParamLoad (HProblem prob, string enginename, string
                loadspecfile)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`enginename`	Solver Engine whose parameter values will be loaded; "" means the parameters of all Engines.
`loadspecfile`	Filename/path (or range ref string) that contains parameter information to be loaded.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngParamSave

SolverEngParamSave() is a method function that saves Engine parameters to file.

Syntax:

```
SolverEngParamSave (HProblem prob, string enginename, string
                savespecfile)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	Solver Engine whose parameter values will be saved; "" means the parameters of all Engines.
savespecfile	Filename/path (or range ref string) that will contain parameter information to be saved.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngParamReset

SolverEngParamReset() is a method function that resets Engine parameters to their default values.

Syntax:

```
SolverEngParamReset (HProblem prob, string enginename)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	Solver Engine whose parameter will be reset to their default values; "" means the parameters of all Engines.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Engine Property Functions

Click to navigate to the desired topic:

- SolverEngCount
- SolverEngName
- SolverEngType
- SolverEngParamCount
- SolverEngParamName
- SolverEngParamGet
- SolverEngParamSet
- SolverEngParamInfo
- SolverEngLimit
- SolverEngStat

SolverEngCount

SolverEngCount() is a property function that returns the number of Solver Engines in the Engines Collection.

Syntax:

```
SolverEngCount (HProblem prob, int* pcount)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
pcount	Number of Solver Engines in the Engines Collection.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngName

SolverEngName() is a property function that returns the string name of a Solver Engine.

Syntax:

```
SolverEngName (HProblem prob, int index, string* penginename,
               string* penginefilespec)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
index	Index of the Solver Engine in the Engines Collection.
penginename	Solver Engine standard name.
penginefilespec	Name/path of the Engine DLL.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngType

SolverEngType() is a property function that returns the Solver Type and/or Problem type for the current Engine.

Syntax:

```
SolverEngType (HProblem prob, _Solver_Type* psolvertype,
               _Problem_Type* pproblemtype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`psolvertype`	Solver type, returned via the `_Solver_Type` enum.
`pproblemtype`	Problem type, returned via the `_Problem_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngParamCount

SolverEngParamCount() is a property function that returns the number of parameters in the parameters Collection of an Engine.

Syntax:

```
SolverEngParamCount (HProblem prob, string enginename, int*
                pcount)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	Solver Engine standard name.
pcount	Number of parameters in the Parameters Collection.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngParamName

SolverEngParamName() is a property function that returns the name of a parameter from the parameters Collection of an Engine.

Syntax:

```
SolverEngParamName (HProblem prob, string enginename, int index,
                    string* pparamname)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	Solver Engine standard name.
index	Index of parameter in the Parameters Collection.
pparamname	Name of the parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngParamGet

SolverEngParamGet() is a property function that returns the current value of a Solver Engine parameter.

Syntax:

```
SolverEngParamGet (HProblem prob, string paramname, double*
                pparamvalue)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
paramname	Name of the parameter to get – may be common or Engine-specific.
pparamvalue	Current value of the parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngParamSet

SolverEngParamSet() is a property function that sets the value of a Solver Engine parameter.

Syntax:

```
SolverEngParamSet (HProblem prob,   paramname, double
              paramvalue)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
paramname	Name of the parameter to set – may be common or Engine-specific.
paramvalue	New value of the parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngParamInfo

SolverEngParamInfo() is a property function that returns the default / minimum / maximum value of a Solver Engine parameter.

Syntax:

```
SolverEngParamInfo (HProblem prob,   paramname, double*
              pparamdefault, double* pparammin, double*
              pparammax)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
paramname	Name of the parameter whose info will be returned – may be common or Engine-specific.
pparamdefault	Default value of the parameter.
pparammin	Minimum value of the parameter.
pparammax	Maximum value of the parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngLimit

SolverEngLimit() is a property function that returns the limits of a Solver Engine. Pass NULL for any arguments not needed.

Syntax:

```
SolverEngLimit (HProblem prob, _Problem_Type problemtype,
                enginename, int* pvarlimit, int* pfcnlimit, int*
                piterboundlimit, int* pcorrintegerlimit)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`problemtype`	Id of Problem type, specified via the `_Problem_Type` enum, for which size limits will be returned.
`enginename`	Solver Engine whose size limits should be returned.
`pvarlimit`	Maximum number of decision/uncertain variables.
`pfcnlimit`	Maximum number of constraint/uncertain functions.
`piterboundlimit`	Maximum number of iterations or variable bounds.
`pcorrintegerlimit`	Maximum number of correlations or integer variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngStat

SolverEngStat() is a property function that retrieves performance statistics for a Solver Engine. Pass NULL for any arguments not needed.

Syntax:

```
SolverEngStat (HProblem prob,   enginename, int* pmilliseconds,
               int* piterations, int* psubproblems, int*
               plocalsols, int* pfuncevals, int* pjacevals, int*
               phessevals)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
enginename	Solver Engine whose performance statistics will be retrieved (for the last Problem solved).
pmilliseconds	Total time in milliseconds used to find solution.
piterations	Total number of iterations (across all subproblems) used to find solution.
psubproblems	Number of subproblems solved, when top-level algorithm[1] used; otherwise 1.
plocalsols	Number of incumbents or locally optimal solutions found, when top-level algorithm used; otherwise 1.
pfuncevals	Total number of calls to funceval callback function used to find solution.
jpacevals	Total number of calls to Jacobian callback function used to find solution.
phessevals	Total number of calls to Hessian callback function used to find solution.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

[1] The top-level algorithm comprises the Branch and Bound algorithm and the Multistart methods.

Evaluator Method Functions

Click to navigate to the desired topic:

- SolverEvaluatorAdd
- SolverEvaluatorRemove

SolverEvaluatorAdd

SolverEvaluatorAdd() is a method function that adds an Evaluator.

Syntax:

```
SolverEvaluatorAdd (HProblem prob, EVALUATOR evaluator, void*
              refuser, _Eval_Type evaltype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`evaluator`	Evaluator to add. Interface method signature: `_Engine_Action Evaluator (HProblem prob, HProblem refuser, _Eval_Type evaltype)`
`refuser`	Pointer to a user data structure/class.
`evaltype`	Evaluator type, specified via the `_Eval_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEvaluatorRemove

SolverEvaluatorRemove() is a method function that removes an Evaluator.

Syntax:

```
SolverEvaluatorRemove (HProblem prob, _Eval_Type evaltype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`evaltype`	Evaluator type, specified via the `_Eval_Type` enum.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

Model Method Functions

Click to navigate to the desired topic:

- SolverModArrayOrder
- SolverModDependTest

SolverModArrayOrder

SolverModArrayOrder() is a method function that sets the array ordering for the matrices in the model (by rows or by columns).

Syntax:

```
SolverModArrayOrder (HProblem prob, _Array_Order arrayorder)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`arrayorder`	Array ordering specified via the `_Array_Order` enum.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverModDependTest

SolverModDependTest() is a method function that determines the Problem type of a model, and fills in the linear matrix coefficients when appropriate.

Syntax:

```
SolverModDependTest (HProblem prob, _Variable_Type vartypes, int
              numtests, double testvals[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`numtests`	Number of tests to perform.
`testvals [numtests * numvars, test major order]`	Test values for variables, or NULL.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Model Property Functions

Click to navigate to the desired topic:

- SolverModNumVariables
- SolverModNumFunctions
- SolverModNumAllGrad
- SolverModNumFcnHess
- SolverModNumDistArguments
- SolverModNumCorrelations
- SolverModAllLinearGet
- SolverModAllLinearSet
- SolverModAllSpLinearGet
- SolverModAllSpLinearSet
- SolverModAllGradDependGet
- SolverModAllGradDependSet
- SolverModAllGradientGet
- SolverModAllGradientSet
- SolverModAllSpGradientGet
- SolverModAllSpGradientSet
- SolverModFcnHessianGet
- SolverModFcnHessianSet
- SolverModFcnSpHessianGet
- SolverModFcnSpHessianSet
- SolverModFcnQuadraticGet
- SolverModFcnQuadraticSet
- SolverModFcnSpQuadraticGet
- SolverModFcnSpQuadraticSet
- SolverModFcnHessDependGet
- SolverModFcnHessDependSet
- SolverModDistTypeGet
- SolverModDistTypeSet
- SolverModDistNameGet
- SolverMod
- Solver
- SolverModParamGet
- SolverModParamSet
- SolverModParamInfo
- SolverModDistCorrelationsGet
- SolverModDistCorrelationsSet
- SolverModVarIndexName
- SolverModFcnIndexName
- SolverModNameVarIndex
- SolverModNameFcnIndex

SolverModNumVariables

SolverModNumVariables() is a property function that retrieves the number of variables of specified type.

Syntax:

```
SolverModNumVariables (HProblem prob, _Variable_Type vartype,
                int* pnumvariables)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`vartype`	Variable type specified via the `_Variable_Type` enum.
`pnumvariables`	Number of variables of the type specified.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModNumFunctions

SolverModNumFunctions() is a property function that retrieves the number of functions of specified type.

Syntax:

```
SolverModNumFunctions (HProblem prob, _Function_Type fcntype,
                int* pnumfunctions)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntype`	Function type specified via the `_Function_Type` enum.
`pnumfunctions`	Number of functions of the specified type.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModNumAllGrad

SolverModNumAllGrad() is a property function that retrieves the number of nonzero gradient elements in functions.

Syntax:

```
SolverModNumAllGrad (HProblem prob, _Function_Type fcntype,
                     _Variable_Type vartype, int* pnumallgradnz)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntype`	Function type specified via the `_Function_Type` enum.
`vartype`	Variable type specified via the `_Variable_Type` enum.
`pnumallgradnz`	Number of nonzero gradient elements in functions, with respect to variables, of specified types.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModNumFcnHess

SolverModNumFcnHess() is a property function that retrieves the number of nonzero elements in DistArguments matrix.

Syntax:

```
SolverModFcnHess (HProblem prob, _Function_Type fcntype,
                  _Variable_Type vartype, int fcnindex, int*
                  pnumfcnhessnz)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntype`	Function type specified via the `_Function_Type` enum.
`vartype`	Variable type specified via the `_Variable_Type` enum.
`fcnindex`	Function index (in Model flat address space).
`pnumfcnhessnz`	Number of nonzero Hessian elements in function.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModNumDistArguments

SolverModNumDistArguments() is a property function that retrieves the number of nonzero elements in DistArguments matrix.

Syntax:

```
SolverModNumDistArguments (HProblem prob, int* pnumdistargnz)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
pnumdistargnz	Number of nonzero elements in DistArguments matrix.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModNumCorrelations

SolverModNumCorrelations() is a property function that retrieves the number of nonzero elements in DistCorrelations matrix.

Syntax:

```
SolverModNumCorrelations (HProblem prob, int* pnumcorrelnz)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
pnumcorrelnz	Number of nonzero elements in DistCorrelations matrix.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllLinearGet

SolverModAllLinearGet() is a property function that returns the constant gradient elements (LP coefficients) of functions with respect to variables of specified types.

The difference between this function and `SolverModAllSpLinearGet` is that this function returns the full matrix while `SolverModAllSpLinearGet` returns only the non-zero elements.

Syntax:

```
SolverModAllLinearGet (HProblem prob, _Function_Type fcntypes,
                _Variable_Type vartypes, double coeff[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`coeff`	Constant gradient elements (LP coefficients) of functions with respect to variables of specified types.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllLinearSet

SolverModAllLinearSet() is a property function that sets the constant gradient elements (LP coefficients) of functions with respect to variables of specified types.

Syntax:

```
SolverModAllLinearSet (HProblem prob, _Function_Type fcntypes,
                       _Variable_Type vartypes, double coeff[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`coeff`	Constant gradient elements (LP coefficients) of functions with respect to variables of specified types.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllSpLinearGet

SolverModAllSpLinearGet() is a property function that returns constant gradient non-zero elements (LP coefficients) of functions with respect to variables of specified types.

The difference between this function and `SolverModAllLinearGet` is that this function returns only non-zeroes while `SolverModAllLinearGet` returns the full matrix.

Syntax:

```
SolverModAllSpLinearGet (HProblem prob, _Function_Type fcntypes
              _Variable_Type vartypes, int* pnumcoeffnz, int
              majorstart[], int minorindex[], double coeffnz[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	`Handle to the problem, passed by the user.`
`fcntypes`	`Function types specified via the _Function_Type enum.`
`vartypes`	`Variable types specified via the _Variable_Type enum.`
`pnumcoeffnz`	`Number of non-zeros (size of minorindex[], coeffnz[]).`
`majorstart [numvars]`	Index in `coeffnz[]` of starting coefficient for variable `i` (`Array_ByCol`) or function `i` (`Array_ByRow`).
`minorindex [numcoeffnz]`	`minorindex [j]` = k = index of function (`Array_ByCol`) or variable (`Array_ByRow`) for `coeffnz[j]`.
`Coeffnz`	`Constant gradient elements (LP coefficients) of functions with respect to variables of specified types.`

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllSpLinearSet

SolverModAllSpLinearSet() is a property function that sets the nature of dependence on variables of selected types.

Syntax:

```
SolverModAllSpLinearSet (HProblem prob, _Function_Type fcntypes,
                _Variable_Type vartypes, int numcoeffnz, int
                majorstart[], int minorindex[], double coeffnz[])
```

Parameters:

Parameter Name	Parameter Description
Prob	Handle to the problem, passed by the user.
fcntypes	Function types specified via the _Function_Type enum.
vartypes	Variable types specified via the _Variable_Type enum.
numcoeffnz	Number of non-zeroes (size of minorindex[], coeffnz[]).
majorstart [numvars]	Index in coeffnz[] of starting coefficient for variable i (Array_ByCol) or function i (Array_ByRow).
minorindex [numcoeffnz]	minorindex [j] = k = index of function (Array_ByCol) or variable (Array_ByRow) for coeffnz[j].
Coeffnz	Constant gradient elements (LP coefficients) of functions with respect to variables of specified types.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllGradDependGet

SolverModAllGradDependGet() is a property function that returns the nature of dependence of functions with respect to the variables of specified types.

Syntax:

```
SolverModAllGradDependGet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, , int*
              pnumdepend, int majorstart[], int minorindex[],
              _Depend_Type graddepend[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`pnumdepend`	Number of elements in `minorindex[]` and `graddepend[]`.
`majorstart [numvars]`	Index in `graddepend[]` of starting coefficient for variable `i` (`Array_ByCol`) or function `i` (`Array_ByRow`).
`minorindex [numcoeffnz]`	`minorindex [j]` = k = index of function (`Array_ByCol`) or variable (`Array_ByRow`) for `graddepend[j]`.
`graddepend [numdepend]`	Nature of dependence on variables of selected types specified via the `_Depend_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllGradDependSet

SolverModAllGradDependSet() is a property function that sets the nature of dependence on variables of selected types.

Syntax:

```
SolverModAllGradDependSet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, int numdepend,
              int majorstart[], int minorindex[], _Depend_Type
              graddepend[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`numdepend`	Number of elements in `minorindex[]` and `graddepend[]`.
`majorstart [numvars]`	Index in `graddepend[]` of starting coefficient for variable `i` (`Array_ByCol`) or function `i` (`Array_ByRow`).
`minorindex [numcoeffnz]`	`minorindex [j]` = k = index of function (`Array_ByCol`) or variable (`Array_ByRow`) for `graddepend[j]`.
`graddepend [numdepend]`	Nature of dependence on variables of selected types specified via the `_Depend_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllGradientGet

SolverModAllGradientGet() is a property function that returns all gradient values, dense.

The ordering of the returned array is based on the current `Array_Order` setting.

Syntax:

```
SolverModAllGradientGet (HProblem prob, _Function_Type fcntypes,
                _Variable_Type vartypes, double coeff[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`coeff[]`	Gradient values returned.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllGradientSet

SolverModAllGradientSet() is a property function that sets all gradient values, dense.

Syntax:

```
SolverModAllGradientSet (HProblem prob, _Function_Type fcntypes,
                _Variable_Type vartypes, double coeff[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`coeff[]`	Gradient values set.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllSpGradientGet

SolverModAllSpGradientGet() is a property function that returns all gradient values, sparse.

The ordering of the returned array is based on the current `Array_Order` setting.

Syntax:

```
SolverModAllSpLinearGet (HProblem prob, _Function_Type fcntypes,
                _Variable_Type vartypes, int* pnumcoeffnz, int
                majorstart[], int minorindex[], double coeffnz[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`pnumcoeffnz`	Number of coefficients to return.
`majorstart[]`	Array indicating the positions in the values array at which the elements for the next column (if column-wise) or the elements for the next row (if row-wise) start. (The starting position is 0, the next is 1, etc..)
`minorindex[]`	Array indicating the row positions (if column-wise) or column positions (if row-wise) at which the values appear in the values array. (The first row/column is 0, the next is 1, etc..)
`coeffnz[]`	The gradient values returned.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModAllSpGradientSet

SolverModAllSpGradientSet() is a property function that sets all gradient values, sparse.

Syntax:

```
SolverModAllSpGradientSet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, int
              numcoeffnz, int majorstart[], int minorindex[],
              double coeffnz[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`numcoeffnz`	Number of coefficients to set.
`majorstart[]`	Array indicating the positions in the values array at which the elements for the next column (if column-wise) or the elements for the next row (if row-wise) start. (The starting position is 0, the next is 1, etc..)
`minorindex[]`	Array indicating the row positions (if column-wise) or column positions (if row-wise) at which the values appear in the values array. (The first row/column is 0, the next is 1, etc..)
`coeffnz[]`	The gradient values set.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnHessianGet

SolverModFcnHessianGet() is a property function that returns the Hessian values for a function, dense.

The ordering of the returned array is based on the current `Array_Order` setting.

Syntax:

```
SolverModFcnHessianGet (HProblem prob, _Function_Type fcntypes,
                _Variable_Type vartypes, int fcnindex, double
                qcoeff[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`fcnindex`	Indexes of the values to return.
`qcoeff[]`	The Hessian values to return.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnHessianSet

SolverModFcnHessianSet() is a property function that sets Hessian values for a function, dense.

Syntax:

```
SolverModFcnHessianSet (HProblem prob, _Function_Type fcntypes,
                _Variable_Type vartypes, int fcnindex, double
                qcoeff[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`fcnindex`	Indexes of the values to set.
`qcoeff[]`	The Hessian values to set.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnSpHessianGet

SolverModFcnSpHessianGet() is a property function that returns the Hessian values for a function, sparse.

The ordering of the returned array is based on the current `Array_Order` setting.

Syntax:

```
SolverModFcnSpHessianGet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, int fcnindex,
              int *pnumqcoeffnz, int majorstart[], int
              minorindex[], double qcoeffnz[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`fcnindex`	Index of the function for which the Hessian values are returned.
`pnumqcoeffnz`	Number of coefficients to be returned.
`majorstart[]`	Array indicating the positions in the values array at which the elements for the next column (if column-wise) or the elements for the next row (if row-wise) start. (The starting position is 0, the next is 1, etc..)
`minorindex[]`	Array indicating the row positions (if column-wise) or column positions (if row-wise) at which the values appear in the values array. (The first row/column is 0, the next is 1, etc..)
`qcoeffnz[]`	The Hessian values returned for the function.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnSpHessianSet

SolverModFcnSpHessianSet() is a property function that sets Hessian values for a function, sparse.

Syntax:

```
SolverModFcnSpHessianSet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, int fcnindex,
              int numqcoeffnz, int majorstart[], int
              minorindex[], double qcoeffnz[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`fcnindex`	Index of the function for which the Hessian values are set.
`numqcoeffnz`	Number of coefficients to be set.
`majorstart[]`	Array indicating the positions in the values array at which the elements for the next column (if column-wise) or the elements for the next row (if row-wise) start. (The starting position is 0, the next is 1, etc..)
`minorindex[]`	Array indicating the row positions (if column-wise) or column positions (if row-wise) at which the values appear in the values array. (The first row/column is 0, the next is 1, etc..)
`qcoeffnz[]`	The Hessian values set for the function.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnQuadraticGet

SolverModFcnQuadraticGet() is a property function that returns the constant Hessian matrix elements, with respect to variables of specified types for a given function.

The ordering of the returned array is based on the current `Array_Order` setting.

Syntax:

```
SolverModFcnQuadraticGet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, int fcnindex,
              double qcoeff[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`fcnindex`	Index of function (in Model flat address space).
`qcoeff`	Constant Hessian matrix elements, with respect to variables of specified types.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnQuadraticSet

SolverModFcnQuadraticSet() is a property function that sets the constant Hessian matrix elements, with respect to variables of specified types.

Syntax:

```
SolverModFcnQuadraticSet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, int fcnindex,
              double qcoeff[])
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
vartypes	Variable types specified via the _Variable_Type enum.
fcntypes	Function types specified via the _Function_Type enum.
fcnindex	Index of function (in Model flat address space).
qcoeff	Constant Hessian matrix elements, with respect to variables of specified types.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnSpQuadraticGet

SolverModFcnSpQuadraticGet() is a property function that returns the Constant Hessian matrix elements, with respect to variables of specified types.

The ordering of the returned array is based on the current `Array_Order` setting.

Syntax:

```
SolverModSpQuadraticGet (HProblem prob, _Function_Type fcntypes,
              _Variable_Type vartypes, int fcnindex, int
              *pnumqcoeffnz, int majorstart[], int
              minorindex[], double qcoeffnz[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`fcnindex`	Index of function (in Model flat address space).
`pnumqcoeffnz`	Number of non-zeroes (size of `qminorindex[]`, `qcoeffnz[]`).
`majorstart [numvars]`	Index in `coeffnz[]` of starting coefficient for first variable `i`.
`minorindex [numqcoeffnz]`	`minorindex[j]` = k = index of second variable for `qcoeffnz[j]`.
`qcoeffnz [numqcoeffnz]`	Constant Hessian matrix elements, with respect to variables of specified types.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnSpQuadraticSet

SolverModFcnSpQuadraticSet() is a property function that sets the Constant Hessian matrix elements, with respect to variables of specified types.

Syntax:

```
SolverModSpQuadraticSet (HProblem prob, _Function_Type fcntypes
              _Variable_Type vartypes, int fcnindex, int
              numqcoeffnz, int majorstart[], int minorindex[],
              double qcoeffnz[])
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
fcntypes	Function types specified via the _Function_Type enum.
vartypes	Variable types specified via the _Variable_Type enum.
fcnindex	Index of function (in Model flat address space).
numqcoeffnz	Number of non-zeroes (size of qminorindex[], qcoeffnz[]).
majorstart [numvars]	Index in coeffnz[] of starting coefficient for first variable i.
minorindex [numqcoeffnz]	minorindex[j] = k = index of second variable for qcoeffnz[j].
qcoeffnz [numqcoeffnz]	Constant Hessian matrix elements, with respect to variables of specified types.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnHessDependGet

SolverModFcnHessDependGet() is a property function that returns the Hessian dependence type.

The ordering of the returned array is based on the current `Array_Order` setting.

Syntax:

```
SolverModFcnHessDependGet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, int fcnindex,
              int* pnumdepend, int majorstart[], int
              minorindex[], _Depend_Type hessdepend[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`fcnindex`	Index of function (in Model flat address space).
`pnumdepend`	Number of elements in `minorindex[]` and `hessdepend[]`.
`majorstart [numvars]`	Index in `hessdepend[]` of starting coefficient for first variable `i`.
`minorindex [numdepend]`	`minorindex[j]` = k = index of second variable for `hessdepend[j]`.
`hessdepend [numdepend]`	Hessian dependence, specified via the `_Depend_Type` enum: 2 = constant (quadratic), 16 = non-constant (smooth nonlinear).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnHessDependSet

SolverModFcnHessDependSet() is a property function that sets the Hessian dependence type.

Syntax:

```
SolverModFcnHessDependSet (HProblem prob, _Function_Type
              fcntypes, _Variable_Type vartypes, int fcnindex,
              int numdepend, int majorstart[], int
              minorindex[], _Depend_Type hessdepend[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`fcntypes`	Function types specified via the `_Function_Type` enum.
`vartypes`	Variable types specified via the `_Variable_Type` enum.
`fcnindex`	Index of function (in Model flat address space).
`numdepend`	Number of elements in `minorindex[]` and `hessdepend[]`.
`majorstart [numvars]`	Index in `hessdepend[]` of starting coefficient for first variable `i`.
`minorindex [numdepend]`	`minorindex[j]` = k = index of second variable for `hessdepend[j]`.
`hessdepend [numdepend]`	Hessian dependence, specified via the `_Depend_Type` enum: 2 = constant (quadratic), 16 = nonconstant (smooth nonlinear).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistTypeGet

SolverModDistTypeGet() is a property function that returns the type of distribution for uncertain variables or functions

Syntax:

```
SolverModDistTypeGet (HProblem prob, _Dist_VarFcn varfcn, int
              index, _Distribution_Type *disttype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`varfcn`	Specifies that the call applies to an uncertain variable or an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	The index of the variable or function in the flat index space.
`disttype`	Types of distributions for uncertain variables, specified via the `_Distribution_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistTypeSet

SolverModDistTypeSet() is a property function that sets the type of distribution for uncertain variables or functions

Syntax:

```
SolverModDistTypeSet (HProblem prob, _Dist_VarFcn varfcn, int
              index, _Distribution_Type disttype))
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`varfcn`	Specifies that the call applies to an uncertain variable or an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	The index of the variable or function in the flat index space.
`disttype`	Types of distributions for uncertain variables, specified via the `_Distribution_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistNameGet

SolverModDistNameGet() is a property function that retrieves the name of the uncertain variables distribution.

Syntax:

```
SolverModDistNameGet(void* prob, _Dist_VarFcn  varfcn, int
              index, LPCWSTR* lpszName)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
varfcn	Specifies that the call applies to an uncertain variable or an uncertain function. For more information, refer to enum Dist_VarFcn.
lpszName	The name of the distribution returned.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModParamCount

SolverModParamCount() is a property function that returns the number of model parameters.

Syntax:

```
SolverModParamCount (HProblem prob, LPCWSTR enginename,
                int* pcount)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`enginename`	Solver Engine standard name.
`pcount`	Returns the number of model parameters.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverModParamName

SolverModParamName() is a property function that returns the name of the model parameter for the given index.

Syntax:

```
SolverModParamName (HProblem prob, int index, LPCWSTR*
                pparamname)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
index	The index of the model parameter.
pparamname	Returns the name of the model parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModParamGet

SolverModParamGet() is a property function that returns the value for the given parameter.

Syntax:

```
SolverModParamGet (HProblem prob, LPCWSTR paramname, double*
              pparamvalue)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`paramname`	Name of the parameter.
`pparamvalue`	Returns the value of the named parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModParamSet

SolverModParamSet() is a property function that sets the value for the named parameter.

Syntax:

```
SolverModParamSet (HProblem prob, LPCWSTR paramname, double
              paramvalue)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`paramname`	The name of the model parameter.
`paramvalue`	Sets the value for the named parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModParamInfo

SolverModParamInfo() is a property function that gets the default, minimum, and maximum values for the named parameter.

Syntax:

```
SolverModParamInfo (HProblem prob, LPCWSTR paramname, double*
              pparamdefault, double* pparammin, double*
              pparammax)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`paramname`	The name of the model parameter.
`pparamdefault`	Returns the default value for the named parameter.
`pparammin`	Returns the minimum value allowed for the named parameter.
`pparammax`	Returns the maximum value allowed for the named parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistCorrelationsGet

SolverModDistCorrelationsGet() is a property function that returns the correlation coefficient between uncertain variables.

Syntax:

```
SolverModDistCorrelationsGet (HProblem prob, int* pnumcorrsnz,
              int majorstart[], int minorindex[], double
              corrsnz[])
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`pnumcorrsnz`	number of nonzero correlations (size of `minorindex[]`, `corrsnz[]` arrays).
`majorstart [numuncvars]`	Index in `corrsnz[]` of starting argument for uncertain variable `i`.
`minorindex [numcorrsnz]`	`minorindex[j]` = k = index of other variable for `corrsnz[j]`.
`corrsnz [numargsnz]`	Correlation coefficient between uncertain variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistCorrelationsSet

SolverModDistCorrelationsSet() is a property function that sets the correlation coefficient between uncertain variables.

Syntax:

```
SolverModDistCorrelationsSet (HProblem prob, int numcorrsnz, int
              majorstart[], int minorindex[], double corrsnz[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`numcorrsnz`	Number of nonzero correlations (size of `minorindex[]`, `corrsnz[]` arrays).
`majorstart [numuncvars]`	Index in `corrsnz[]` of starting argument for uncertain variable `i`.
`minorindex [numcorrsnz]`	`minorindex[j]` = k = index of other variable for `corrsnz[j]`.
`corrsnz [numargsnz]`	Correlation coefficient between uncertain variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModVarIndexName

SolverModVarIndexName() is a method function that returns the name of a variable range. (Not implemented yet.)

Syntax:

```
SolverModVarIndexName (HProblem prob, int index, int elements,
                LPCWSTR * pname)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
index	Index of first variable in range, in Model index space.
elements	Number of variables in range.
pname	Symbolic name of this range in modeling language.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModFcnIndexName

SolverModFcnIndexName() is a method function that returns the name of a function range. (Not implemented yet.)

Syntax:

```
SolverModFcnIndexName (HProblem prob, int index, int elements,
                LPCWSTR * pname)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
index	Index of first function in range, in Model index space.
elements	Number of function in range.
pname	Symbolic name of this range in modeling language.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverModNameVarIndex

SolverModNameVarIndex() is a method function that returns the index of a variable and the number of variables in a range.

Syntax:

```
SolverModNameVarIndex (HProblem prob, LPCWSTR name, int* pindex,
                int* pelements)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Symbolic name of this range in modeling language.
pindex	Index of first variable in range, in Model index space.
pelements	Number of variables in range.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModNameFcnIndex

SolverModNameFcnIndex() is a method function that the index of a function and the number of functions in a range.

Syntax:

```
SolverModNameFcnIndex (HProblem prob, LPCWSTR name, int* pindex,
                int* pelements)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Symbolic name of this range in modeling language.
pindex	Index of first function in range, in Model index space.
pelements	Number of functions in range.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Variable Method Functions

Click to navigate to the desired topic:

- SolverVariableAdd
- SolverVariableRemove
- SolverVarGetCorrelation
- SolverVarGetFrequency
- SolverVarNonNegative

SolverVariableAdd

SolverVariableAdd() is a method function that adds a vector of variables of a given type to the Variables Collection.

Syntax:

```
SolverVariableAdd (HProblem prob, LPCWSTR name, _Variable_Type
                vartype, int size)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of vector of variables.
`vartype`	Variable type specified via the `_Variable_Type` enum.
`size`	Number of elements in vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVariableRemove

SolverVariableRemove() is a method function that removes a vector of variables from the Variables Collection.

Syntax:

```
SolverVariableRemove (HProblem prob, name)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of vector of variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarGetCorrelation

SolverVarGetCorrelation() is a method function that returns the correlation coefficients between elements of a variable vector and other uncertain variable.

Syntax:

```
SolverVarGetCorrelation (HProblem prob, name, int varindex,
                double correlation[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of vector of variables. (Must be of `Uncertain_Variable` type.)
`varindex`	Index of other uncertain variable with which elements of this variable vector are correlated.
`correlation`	Correlation coefficients between other uncertain variable and elements of this variable vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarGetFrequency

SolverVarGetFrequency() is a method function that returns the frequency of observations of variables in each bin.

Syntax:

```
SolverVarGetFrequency (HProblem prob, LPCWSTR name,
                _Frequency_Type frequencytype, int numbins,
                double binBoundary, double binFrequency)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of vector of variables. (Must be of `Uncertain_Variable` type.)
`frequencytype`	Frequency type – density or cumulative – specified via the `_Frequency_Type` enum.
`numbins`	Number of histogram bars or cumulative plot points.
`binBoundary [numbins+1]`	Lower bound of each bin, the extra element is the upper bound of the last bin.
`binFrequency`	Frequency of observations in each bin – sorted up or down if `_Frequency_Type` is cumulative.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarNonNegative

SolverVarNonNegative() is a method function that declares a lower bound of 0 for variables that do not have a lower bound.

Syntax:

```
SolverVarNonNegative (HProblem prob, LPCWSTR name)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of a vector of variables. (Must be of `Decision_Variable` type.)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Variable Property Functions

Click to navigate to the desired topic:

- SolverVarCount
- SolverVarNameGet
- SolverVarNameSet
- SolverVarTypeGet
- SolverVarTypeSet
- SolverVarValue
- SolverVarSizeGet
- SolverVarTypeSet
- SolverVarPosition
- SolverVarStatistics
- SolverVarPercentiles
- SolverVarAllTrialsGet
- SolverVarAllTrialsSet
- SolverVarLowerBoundGet
- SolverVarLowerBoundSet
- SolverVarUpperBoundGet
- SolverVarUpperBoundSet
- SolverVarIntegerTypeGet
- SolverVarIntegerTypeSet
- SolverVarGroupIndexGet
- SolverVarGroupIndexSet
- SolverVarConeTypeGet
- SolverVarConeTypeSet
- SolverSolutionsIndexGet
- SolverVarConeIndexSet
- SolverVarInitialValueGet
- SolverVarInitialValueSet
- SolverVarFinalValue
- SolverVarFinalValue
- SolverVarDualLower
- SolverVarDualUpper
- SolverVarMeanAbsDev
- SolverVarSemiDeviation
- SolverVarValueAtRisk
- SolverVarCondValueAtRisk
- SolverVarMeanCI
- SolverVarCITrials
- SolverVarTarget

SolverVarCount

SolverProblem() is a property function that returns the number of variable vectors in the Variables Collection.

Syntax:

```
SolverVarCount (HProblem prob,  int* pcount)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
pcount	Number of variable vectors in the Variables Collection.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarNameGet

SolverVarNameGet() is a property function that returns the name of a vector of variables.

Syntax:

```
SolverVarNameGet (HProblem prob, int index, LPCWSTR* pname)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
index	Index of variable vector in the Variables Collection.
Pname	Name of vector of variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarNameSet

SolverVarNameSet() is a property function that sets the name of a vector of variables.

Syntax:

```
SolverVarNameSet (HProblem prob, int index, LPCWSTR name)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
index	Index of variable vector in the Variables Collection.
Name	Name of vector of variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarTypeGet

SolverVarTypeGet() is a property function that returns variable type for a vector of variables.

Syntax:

```
SolverVarTypeGet (HProblem prob, LPCWSTR name, _Variable_Type*
                  pvartype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables.
`pvartype`	Variable type specified via the `_Variable_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarTypeSet

SolverVarTypeSet() is a property function that sets variable type for a vector of variables.

Syntax:

```
SolverVarTypeSet (HProblem prob, LPCWSTR name, _Variable_Type
                  vartype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables.
`vartype`	Variable type specified via the `_Variable_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarValue

SolverVarValue() is a property function that returns the values of a vector of variables.

Syntax:

```
SolverVarValue (HProblem prob, LPCWSTR name, double value[])
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of variables.
Value	Current value of the variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarSizeGet

SolverVarSizeGet() is a property function that returns the number of elements in variable vector.

Syntax:

```
SolverVarSizeGet (HProblem prob, LPCWSTR name, int* psize)
```

Parameters:

Parameter Name	Parameter Description
Prob	Handle to the problem, passed by the user.
Name	Name of the vector of variables.
psize	Current number of elements in the variable vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarSizeSet

SolverVarSizeSet() is a property function that sets the number of elements in variable vector.

Syntax:

```
SolverVarSizeSet (HProblem prob, LPCWSTR name, int size)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of variables.
size	New number of elements in the variable vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarPosition

SolverVarPosition() is a property function that returns the position of the first element in Model flat index space.

Syntax:

```
SolverVarPosition (HProblem prob, name, int* pposition)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of variables.
pposition	Position of the first element in Model flat index space.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarStatistics

SolverVarStatistics() is a property function that retrieves various statistics for a vector of variables.

Syntax:

```
SolverVarStatistics (HProblem prob, LPCWSTR name,  int*
              pnumvalues,  int* pnumerrors,  double minimum[],
              double maximum[],  double mean[],  double
              stddev[],  double variance[],  double skewness[],
              double kurtosis[],  double mode[])
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of variables.
pnumvalues	Number of values in uncertain variable sample.
pnumerrors	Number of errors in uncertain variable sample.
minimum [varsize]	Minimum value of each variable across sample.
maximum [varsize]	Maximum value of each variable across sample.
mean [varsize]	Mean of each variable across sample.
stddev [varsize]	Standard deviation of each variable across sample.
variance [varsize]	Variance of each variable across sample.
skewness [varsize]	Skewness of each variable across sample.
kurtosis [varsize]	Kurtosis of each variable across sample.
mode [varsize]	Mode of each variable across sample.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarPercentiles

SolverVarPercentiles() is a property function that returns the 5th, 10th, ..., 95th percentiles for a vector of variables.

Syntax:

```
SolverVarPercentiles (HProblem prob, LPCWSTR name, _Array_Order
              arrayorder, double percentiles[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Uncertain_Variable` type.)
`arrayorder`	Array order specified via the `_Array_Order` enum – `Array_ByCols` makes variable index major, `Array_ByRows` makes percentile index major.
`percentiles [varsize*99]`	The 5th, 10th, ..., 95th percentiles. (Variable is column index, percentile is row index.)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarAllTrialsGet

SolverVarAllTrialsGet() is a property function that returns all trial values of variables for a vector of variables.

Syntax:

```
SolverVarAllTrialsGet (HProblem prob, LPCWSTR name, _Array_Order
              arrayorder, double alltrials[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Uncertain_Variable` type.)
`arrayorder`	Array order specified via the `_Array_Order` enum – `Array_ByCols` makes variable index major, `Array_ByRows` makes percentile index major.
`alltrials [varsize*trials]`	Trial values for variables. (Variable is column index, trial is row index.)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarAllTrialsSet

SolverVarAllTrialsSet() is a property function that sets all trial values of variables for a vector of variables.

Syntax:

```
SolverVarAllTrialsSet (HProblem prob, LPCWSTR name, _Array_Order
              arrayorder, double alltrials[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Uncertain_Variable` type.)
`arrayorder`	Array order specified via the _Array_Order enum – `Array_ByCols` makes variable index major, `Array_ByRows` makes percentile index major.
`alltrials [varsize*trials]`	Trial values for variables. (Variable is column index, trial is row index.)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarLowerBoundGet

SolverVarLowerBoundGet() is a property function that returns the lower bounds on variables for a vector of variables.

Syntax:

```
SolverVarLowerBoundGet (HProblem prob, LPCWSTR name, double
              varlower[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Uncertain_Variable` type.)
`varlower [varsize]`	Lower bounds on variables, -1E30 means no lower bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarLowerBoundSet

SolverVarLowerBoundSet() is a property function that sets the lower bounds on variables for a vector of variables.

Syntax:

```
SolverVarLowerBoundSet (HProblem prob, LPCWSTR name, double
              varlower[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Uncertain_Variable` type.)
`varlower [varsize]`	Lower bounds on variables, -1E30 means no lower bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarUpperBoundGet

SolverVarUpperBoundGet() is a property function that returns the upper bounds on variables for a vector of variables.

Syntax:

```
SolverVarUpperBoundGet (HProblem prob, LPCWSTR name, double
              varupper[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Uncertain_Variable` type.)
`varupper [varsize]`	Upper bounds on variables, +1E30 means no upper bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarUpperBoundSet

SolverVarUpperBoundSet() is a property function that sets the upper bounds on variables for a vector of variables.

Syntax:

```
SolverVarUpperBoundSet (HProblem prob, LPCWSTR name, double
              varupper[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Uncertain_Variable` type.)
`varupper [varsize]`	Upper bounds on variables, +1E30 means no upper bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarIntegerTypeGet

SolverVarIntegerTypeGet() is a property function that returns the type of the integer variables in a vector of variables.

Syntax:

```
SolverVarIntegerTypeGet (HProblem prob, LPCWSTR name,
                _Integer_Type integertype[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`integertype [varsize]`	Integer type specified via the `_Integer_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarIntegerTypeSet

SolverVarIntegerTypeSet() is a property function that sets the type of the integer variables in a vector of variables.

Syntax:

```
SolverVarIntegerTypeSet (HProblem prob, LPCWSTR name,
                _Integer_Type integertype[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`integertype [varsize]`	Integer type specified via the `_Integer_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarGroupIndexGet

SolverVarGroupIndexGet() is a property function that returns the index number of the alldifferent group that a variable belongs to.

Syntax:

```
SolverVarGroupIndexGet (HProblem prob, LPCWSTR name, int
              groupindex[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`groupindex [varsize]`	Index number (≥ 1) of the group that variable belongs to, or 0.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarGroupIndexSet

SolverVarGroupIndexSet() is a property function that sets the index number of the alldifferent group that a variable belongs to.

The index is 1-based, i.e. the index for the first alldifferent group is 1, next one - 2, etc.

Syntax:

```
SolverVarGroupIndexSet (HProblem prob, LPCWSTR name, int
                groupindex[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`groupindex [varsize]`	Index number (≥ 1) of the group that variable belongs to, or 0.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarConeTypeGet

SolverVarConeTypeGet() is a property function that returns the cone type for a vector of variables.

Syntax:

```
SolverVarConeTypeGet (HProblem prob, LPCWSTR name, _Cone_Type
                conetype[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`conetype [varsize]`	Cone type specified via the `_Cone_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarConeTypeSet

SolverVarConeTypeSet() is a property function that sets the cone type for a vector of variables.

Syntax:

```
SolverVarConeTypeSet (HProblem prob, LPCWSTR name, _Cone_Type
                conetype[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`conetype [varsize]`	Cone type specified via the `_Cone_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarConeIndexGet

SolverVarConeIndexGet() is a property function that returns the index number of the cone that a variable belongs to.

Syntax:

```
SolverVarConeIndexGet (HProblem prob, LPCWSTR name, int
              coneindex[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`coneindex [varsize]`	Index number (≥ 1) of cone that the variable belongs to, or 0.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarConeIndexSet

SolverVarConeIndexSet() is a property function that sets the index number of the cone that a variable belongs to.

The index is 1-based, i.e. the index for the first cone is 1, the next one - 2, etc.

Syntax:

```
SolverVarConeIndexSet (HProblem prob, LPCWSTR name, int
              coneindex[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`coneindex [varsize]`	Index number (≥ 1) of cone that the variable belongs to, or 0.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarInitialValueGet

SolverVarInitialValueGet() is a property function that returns the initial values of variables.

Syntax:

```
SolverVarInitValueGet (HProblem prob, LPCWSTR name, double
              varinitial[])
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of variables.
varinitial [varsize]	Initial values of the variables for NLP.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarInitialValueSet

SolverVarInitialValueSet() is a property function that sets the initial values of variables for NLP.

Syntax:

```
SolverVarInitValueSet (HProblem prob, LPCWSTR name, double
              varinitial[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables.
`varinitial [varsize]`	Initial values of the variables for NLP.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarFinalValue

SolverVarFinalValue() is a property function that returns the final values for variables for a vector of variables.

Syntax:

```
SolverVarFinalValue (HProblem prob, LPCWSTR name,  double
              varfinal[])
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of variables.
varfinal [varsize]	Final primal values for variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarDualValue

SolverVarDualValue() is a property function that returns the final dual values for variables for a vector of variables.

Syntax:

```
SolverVarDualValue (HProblem prob, LPCWSTR name,  double
               vardual[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`vardual [varsize]`	Final dual values for variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarDualLower

SolverVarDualLower() is a property function that returns, for a vector of variables, the lower bound of the range over which variable dual values are valid.

Syntax:

```
SolverVarDualLower (HProblem prob, LPCWSTR name,  double
              duallower[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`duallower [varsize]`	Lower bound of range over which variable dual values are valid or -1E30 to indicate that there is no lower bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarDualUpper

SolverVarDualUpper() is a property function that returns, for a vector of variables, the upper bound of the range over which variable dual values are valid.

Syntax:

```
SolverVarDualUpper (HProblem prob, LPCWSTR name,  double
              dualupper[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of variables. (Must be of `Decision_Variable` type.)
`dualupper [varsize]`	Upper bound of range over which variable dual values are valid or +1E30 to indicate that there is no upper bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarMeanAbsDev

SolverVarMeanAbsDev() returns the average of the absolute deviations of the specified uncertain variable sample values from their mean. This is also known as Mean Absolute Deviation (MAD), especially in time series analysis applications. It is defined as:

$$MAD(X) = \frac{1}{n}\sum_{i=1}^{n}|\mu - x_i|$$

Here μ is the mean of the sample values.

Syntax:

```
SolverVarMeanAbsDev (HProblem prob, LPCWSTR name,  double*
              values)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`Values`	Average of the absolute deviations of the specified uncertain variable sample values from their mean.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarSemiVariance

SolverVarSemiVariance() returns the semivariance for the specified uncertain variable computed relative to the *target* argument. This is a measure of the dispersion of values of an uncertain variable, but unlike the variance which measures (or penalizes) both positive and negative deviations from the target, the semivariance or lower partial moment is only concerned with *one-sided* deviations from the target. It is usually used in finance and insurance applications, when we are only concerned with downside risks (or loss in portfolio value). The semivariance is computed by summing only the downside differences from the target of all the trials, raised to the given power *q*, divided by the number of trials:

$$\frac{1}{n}\sum_{i=1}^{n}\left(t - x_i\right)_+^q$$

$$\left(x\right)_+ = \max\left(x, 0\right)$$

t = target value

All trials – not just the trials with downside deviations – are included in *n*. If *q* is different from 2, the result is called the 'lower partial moment.'

Syntax:

```
SolverVarSemiVariance (HProblem prob, LPCWSTR name,  int*
              pnPower, double* pfTarget, double* values)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`pnPower`	The power argument, q in the above function.
`pfTarget`	Value to which all sampled values are compared in the computation (t in the above function).
`Values`	Returns the measure of the dispersion of values of the uncertain variables.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverVarSemiDeviation

SolverVarSemiDeviation() returns the semideviation for the specified uncertain variable, relative to the *target*. This is a *one-sided* measure of dispersion of values of the uncertain variable. The semideviation is the square root of the semivariance, described directly below. If a *q* argument different from 2 is specified, SolverVarSemiDeviation() returns the *q*th root of the lower partial moment at power *q* of the uncertain variable.

Syntax:

```
SolverVarSemiDeviation (HProblem prob, LPCWSTR name,  int*
              pnPower, double* pfTarget, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`pnPower`	The power argument, q.
`pfTarget`	Value to which all sampled values are compared in the computation (t in the above function).
`values`	Returns the semideviation for the specified uncertain variable, relative to the pfTarget argument.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarValueAtRisk

SolverVarValueAtRisk() returns the Value at Risk for the specified uncertain variable at the specified 'confidence level,' which is better described as a *percentile* – for example, 0.95 or 0.99.

In finance applications, the Value at Risk is the maximum loss that can occur at a given confidence level. In a distribution of returns or profits, losses would lie at the "left end" of the distribution and would be represented by negative numbers and smaller percentiles (say 0.01 or 0.05). But it is customary in Value at Risk analysis to treat losses as positive numbers at the "right end" of the distribution.

Syntax:

```
SolverVarValueAtRisk (HProblem prob, LPCWSTR name, double
              percentile, double* values)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`percentile`	Specified "confidence level".
`values`	Value at Risk for the specified uncertain variable at the specified "confidence level" in the percentile argument.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarCondValueAtRisk

SolverVarCondValueAtRisk() returns the conditional Value at Risk for the specified uncertain variable, at the specified 'confidence level' which is better described as a *percentile* – for example, 0.95 or 0.99. The conditional Value at Risk is defined as the *expected value* of a loss *given that* a loss at the specified percentile occurs.

Syntax:

```
SolverVarValueAtRisk (HProblem prob, LPCWSTR name, double
              percentile, double* values)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`percentile`	Specified "confidence level".
`Values`	Conditional Value at Risk for the specified uncertain variable at the specified "confidence level" in the percentile argument.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarMeanCI

SolverVarMeanCI() returns the confidence "half-interval" for the estimated mean value for the specified uncertain variable, at the specified *confidence level* (for example 0.95 or 0.99). If μ is the mean value returned by SolverVarStatistics() and δ is the value returned by SolverVarMeanCI(), the true mean is estimated to lie within the interval $\mu - \delta$ to $\mu + \delta$.

The *confidence level* can be interpreted as follows: If we compute a large number of independent estimates of confidence intervals on the true mean of the uncertain variable, each based on *n* observations with *n* sufficiently large, then the proportion of these confidence intervals that contain the true mean of the function should equal the *confidence level*.

If σ^2 (*n*) is the sample variance from *n* trial values, $\alpha = 1 -$ *confidence level*, and $t_{n-1, 1-\alpha/2}$ is the upper $1-\alpha/2$ critical point of the Student's t-distribution with *n*-1 degrees of freedom, the confidence half-interval δ is computed as:

$$t_{n-1,1-\alpha/2}\sqrt{\frac{\sigma^2(n)}{n}}$$

The confidence interval measures the precision with which we have estimated the true mean. Larger half widths imply that there is a lot of variability in our estimates. The above formula for the half-width assumes that the individual x_is are normally distributed; when this is not the case, the above formula still gives us an *approximate* confidence interval on the true mean of the uncertain variable.

Syntax:

```
SolverVarMeanCI (HProblem prob, LPCWSTR name, double confidence,
                 double* values)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`confidence`	Specified "confidence level" (for example 0.95 or 0.99).
`Values`	Confidence "half-interval" for the estimated mean value for the specified uncertain variable, at the specified *confidence level*.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarStdDevCI

SolverVarStdDevCI() returns the confidence 'half-interval' for the estimated standard deviation of the simulation trials (returned by the SolverVarSemiDeviation() function) for the specified uncertain variable, at *confidence level* (for example 0.95 or 0.99). If σ is the value returned by SolverVarSemiDeviation () and δ is the value returned by SolverVarDevCI(), the true mean is estimated to lie within the interval σ - δ to σ + δ.

If σ^2 (n) is the sample variance from n trial values, $\alpha = 1 - $ *confidence level*, and $t_{n-1,\,1-\alpha/2}$ is the upper 1-α/2 critical point of the Student's t-distribution with n-1 degrees of freedom, the confidence half-interval δ is computed as:

$$t_{n-1,1-\alpha/2}\sigma(n)\sqrt{\frac{k-1}{4(n-1)}}$$

See also the description of the SolverVarMeanCI() function.

Syntax:

```
SolverVarStdDevCI (HProblem prob, LPCWSTR name, double
                confidence, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`confidence`	Specified "confidence level" (for example 0.95 or 0.99).
`values`	Confidence 'half-interval' for the estimated standard deviation of the simulation trials for the specified uncertain variable, at *confidence level* (for example 0.95 or 0.99).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarCITrials

SolverVarCITrials() returns the estimated number of simulation trials needed to ensure that the specified uncertain variable sample mean value (returned by the SolverVarStatistics() function) lies within the confidence interval specified by *confidence level* (for example 0.95 or 0.99) and the half-interval size given by *tolerance*. Note that this number of trials is sufficient only to ensure that the single output value specified lies within the confidence interval. To ensure that *N* output cells lie within confidence intervals at level 1 - α (e.g. 0.95 = 1 – 0.05), use a confidence level of α / N.

Syntax:

```
SolverVarCITrials (HProblem prob, LPCWSTR name, double
                confidence, double tolerance, double* values)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`confidence`	Specified "confidence level" (for example 0.95 or 0.99).
`tolerance`	Half-Interval size.
`Values`	Estimated number of simulation trials needed to ensure that the specified uncertain variable sample mean value lies within the confidence interval specified by the *confidence* argument and the half-interval size given by *tolerance*.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverVarTarget

SolverVarTarget() returns the cumulative frequency of the target value in the distribution of trial values for the specified uncertain variable. This function returns the proportion of simulated values that are less than or equal to the *target value*.

Syntax:

```
SolverVarTarget (HProblem prob, LPCWSTR name, double
              targetvalue, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Variable_Type_Uncertain` type.)
`targetvalue`	The function returns the proportion of simulated values that are less than or equal to this value.
`values`	Cumulative frequency of the target value in the distribution of trial values for the specified uncertain variable.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Function Method Functions

Click to navigate to the desired topic:

- SolverFunctionAdd
- SolverFunctionRemove
- SolverFcnGetCorrelation
- SolverFcnGetFrequency
- SolverFcnNonNegative
- SolverFcnRelation

SolverFunctionAdd

SolverFunctionAdd() is a method function that adds a vector of functions of a given type to the Functions Collection.

Syntax:

```
SolverFunctionAdd (HProblem prob, LPCWSTR name, _Function_Type
                fcntype, int size)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of vector of functions.
`fcntype`	Function type specified via the `_Function_Type` enum.
`size`	Number of elements in a vector.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverFunctionRemove

SolverFunctionRemove() is a method function that removes a vector of functions from the Functions Collection.

Syntax:

```
SolverFunctionRemove (HProblem prob, LPCWSTR name)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of vector of functions.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnGetCorrelation

SolverFcnGetCorrelation() is a method function that returns the correlation coefficients between the uncertain functions and the elements of a specified function vector.

Syntax:

```
SolverFcnGetCorrelation (HProblem prob, LPCWSTR name, int
                fcnindex,  double correlation[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of vector of functions. (Must be of `Uncertain_Function` type.)
`fcnindex`	Index of the uncertain function.
`correlation`	Coefficients of correlation between the uncertain functions and the elements of the `name` function vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnGetFrequency

SolverFcnGetFrequency() is a method function that returns the boundaries and frequency of observations of the solution trial points in each bin.

Syntax:

```
SolverFcnGetFrequency (HProblem prob, LPCWSTR name,
              _Frequency_Type frequencytype, int numbins,
              double binBoundary,  double binFrequency)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of vector of functions. (Must be of `Uncertain_Function` type.)
`frequencytype`	Frequency type – either density or cumulative – specified via the `_Frequency_Type` enum.
`numbins`	Number of histogram bars or cumulative plot points.
`binBoundary [numbins+1]`	Lower bound of each bin; the extra element is the upper bound of the last bin.
`binFrequency`	Frequency of observations in each bin. Sorted up if the `_Frequency_Type` is “density” or down if `_Frequency_Type` is “cumulative”.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnNonNegative

SolverFcnNonNegative() is a method function that sets the lower bound of functions to 0, if not already given.

Syntax:

```
SolverFcnNonNegative (HProblem prob, LPCWSTR name)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of vector of functions. (Must be of `_Function_Type_Constraint` type.)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnRelation

SolverFcnRelation() is a method function sets the bounds on functions, based on the specified parameters.

Syntax:

```
SolverFcnRelation (HProblem prob, LPCWSTR name, _Cons_Rel
                consrel[], double righthandside [])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of vector of functions. (Must be of `_Function_Type_Constraint` type.)
`consrel`	Relation specified via the `_Cons_Rel` enum: 0 is ≤ 1 is = 2 is ≥
`righthandside`	Constraint right-hand side value.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Function Property Functions

Click to navigate to the desired topic:

- SolverFcnCount
- SolverFcnNameGet
- SolverFcnNameSet
- SolverFcnTypeGet
- SolverFcnTypeSet
- SolverFcnValueGet
- SolverFcnValueSet
- SolverFcnSizeGet
- SolverFcnSizeSet
- SolverFcnPosition
- SolverFcnStatistics
- SolverFcnPercentiles
- SolverFcnAllTrials
- SolverFcnLowerBoundGet
- SolverFcnLowerBoundSet
- SolverFcnUpperBoundGet
- SolverFcnUpperBoundSet
- SolverFcnFinalValue
- SolverFcnDualValue
- SolverFcnDualLower
- SolverFcnDualUpper
- SolverFcnMeanAbsDev
- SolverFcnSemiVariance
- SolverFcnSemiDeviation
- SolverFcnValueAtRisk
- SolverFcnCondValueAtRisk
- SolverFcnMeanCI
- SolverFcnStdDevCI
- SolverFcnCITrials
- SolverFcnTarget
- SolverFcnChanceTypeGet
- SolverFcnChanceTypeSet
- SolverFcnChanceGet
- SolverFcnChanceSet
- SolverFcnInitValueGet
- SolverFcnInitValueSet

SolverFcnCount

SolverFcnCount() is a property function that returns the number of function vectors in the Functions Collection.

Syntax:

```
SolverFcnCount (HProblem prob,  int* pcount)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
pcount	Number of function vectors in the Functions Collection.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnNameGet

SolverFcnNameGet() is a property function that returns the name of a vector of functions.

Syntax:

```
SolverFcnNameSet (HProblem prob, int index, LPCWSTR* pname)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`index`	Index of the function vector in the Functions Collection.
`pname`	Name of the vector of functions.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnNameSet

SolverFcnNameSet() is a property function that sets the name of a vector of functions.

Syntax:

```
SolverFcnNameSet (HProblem prob, int index, LPCWSTR pname)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
index	Index of the function vector in the Functions Collection.
Pname	Name of the vector of functions.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnTypeGet

SolverFcnTypeGet() is a property function that returns the type of a vector of functions.

Syntax:

```
SolverFcnTypeGet (HProblem prob, LPCWSTR name, _Function_Type*
                  pfcntype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions.
`pfcntype`	Functions type presented via the `_Function_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnTypeSet

SolverFcnTypeSet() is a property function that sets the type of a vector of functions.

Syntax:

```
SolverFcnTypeSet (HProblem prob, LPCWSTR name, _Function_Type
                fcntype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions.
`fcntype`	Functions type specified via the `_Function_Type` enum.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnValueGet

SolverFcnValueGet() is a property function that returns the values of a vector of functions.

Syntax:

```
SolverFcnValueGet (HProblem prob, LPCWSTR name, double value[])
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of functions.
value	Current value of the functions.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnValueSet

SolverFcnValueSet() is a property function that sets the value of vector of functions.

Syntax:

```
SolverFcnValueSet (HProblem prob, LPCWSTR name, double value[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions.
`value`	New value of the functions.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverFcnSizeGet

SolverFcnSizeGet() is a property function that returns the number of elements in a function vector.

Syntax:

```
SolverFcnSizeGet (HProblem prob, LPCWSTR name, int* psize)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions.
`psize`	Current number of elements in function vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnSizeSet

SolverFcnSizeSet() is a property function that sets the number of elements in a function vector.

Syntax:

```
SolverFcnSizeSet (HProblem prob, LPCWSTR name, int size)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of functions.
size	New number of elements in function vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnPosition

SolverFcnPosition() is a property function that retrieves the position of the first element in Model flat index space.

Syntax:

```
SolverFcnPosition (HProblem prob, LPCWSTR name, int* pposition)
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of functions.
pposition	Position of the first element in Model flat index space.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnStatistics

SolverFcnStatistics() is a property function that retrieves various statistics for a given function vector.

Syntax:

```
SolverFcnStatistics (HProblem prob, LPCWSTR name,  int*
              pnumvalues,  int* pnumerrors,  double minimum[],
              double maximum[],  double mean[],  double
              stddev[],  double variance[],  double skewness[],
              double kurtosis[],  double mode[])
```

Parameters:

Parameter Name	Parameter Description
prob	Handle to the problem, passed by the user.
name	Name of the vector of functions.
pnumvalues	Number of values in uncertain function sample.
pnumerrors	Number of errors in uncertain function sample.
minimum [fcnsize]	Minimum value of each function across sample.
maximum [fcnsize]	Maximum value of each function across sample.
mean [fcnsize]	Mean of each function across sample.
stddev [fcnsize]	Standard deviation of each function across sample.
variance [fcnsize]	Variance of each function across sample.
skewness [fcnsize]	Skewness of each function across sample.
kurtosis [fcnsize]	Kurtosis of each function across sample.
mode [fcnsize]	Mode of each function across sample.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnPercentiles

SolverFcnPercentiles() is a property function that returns the 5th, 10th, ..., 95th percentiles for a vector of functions.

Syntax:

```
SolverFcnPercentiles (HProblem prob, LPCWSTR name, _Array_Order
              arrayorder, double percentiles[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `Uncertain_Function` type.)
`arrayorder`	Array order specified via the `_Array_Order` enum – `Array_ByCols` makes function index major, `Array_ByRows` makes percentile index major.
`percentiles [fcnsize*99]`	The 5th, 10th, ..., 95th percentiles (function is column index, percentile is row index).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnAllTrials

SolverFcnAllTrials() is a property function that returns the trial values for functions of vector of functions.

Syntax:

```
SolverFcnAllTrials (HProblem prob, LPCWSTR name, _Array_Order
              arrayorder, double alltrials[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `Uncertain_Function` type.)
`arrayorder`	Array order specified via the `_Array_Order` enum – `Array_ByCols` makes function index major, `Array_ByRows` makes trials index major.
`alltrials [varsize*trials]`	Trial values for functions (function is column index, trial is row index).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnLowerBoundGet

SolverFcnLowerBoundGet() is a property function that returns the lower bounds on functions for a vector of functions.

Syntax:

```
SolverFcnLowerBoundGet (HProblem prob, LPCWSTR name, double
                fcnlower[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Constraint` type.)
`fcnlower[fcnsize]`	Lower bounds on functions, -1E30 means no lower bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnLowerBoundSet

SolverFcnLowerBoundSet() is a property function that sets the lower bounds on functions for a vector of functions.

Syntax:

```
SolverFcnLowerBoundSet (HProblem prob, LPCWSTR name, double
                fcnlower[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Constraint` type.)
`fcnlower[fcnsize]`	Lower bounds on functions, -1E30 means no lower bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnUpperBoundGet

SolverFcnUpperBoundGet() is a property function that returns the upper bounds on functions for a vector of functions.

Syntax:

```
SolverFcnUpperBoundGet (HProblem prob, LPCWSTR name, double
                fcnupper[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Constraint` type.)
`fcnupper[fcnsize]`	Upper bounds on functions, +1E30 means no upper bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnUpperBoundSet

SolverFcnUpperBoundSet() is a property function that sets the upper bounds on functions for a vector of functions.

Syntax:

```
SolverFcnUpperBoundSet (HProblem prob, LPCWSTR name, double
                fcnupper[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Constraint` type.)
`fcnupper[fcnsize]`	Upper bounds on functions, +1E30 means no upper bound.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnFinalValue

SolverFcnFinalValue() is a property function that returns the final primal values for functions of a function vector.

Syntax:

```
SolverFcnFinalValue (HProblem prob, LPCWSTR name, double
                fcnfinal[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions.
`fcnfinal [fcnsize]`	Final primal values for the functions.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnDualValue

SolverFcnDualValue() is a property function that returns the final dual values for constraint functions of a function vector.

Syntax:

```
SolverFcnDualValue (HProblem prob, LPCWSTR name, double
                fcndual[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Constraint` type.)
`fcndual [numconfcns]`	Final dual values for the constraint functions.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnDualLower

SolverFcnDualLower() is a property function that returns, for a vector of functions, the lower bound of range over which the constraint dual values are valid.

Syntax:

```
SolverFcnDualLower (HProblem prob, LPCWSTR name,  double
              fcnlower[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Constraint` type.)
`fcnlower [fcnsize]`	Lower bound of range over which constraint dual values are valid. No lower bound is indicated with -1E30.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnDualUpper

SolverFcnDualUpper() is a property function that returns, for a vector of functions, the upper bound of range over which the constraint dual values are valid.

Syntax:

```
SolverFcnDualUpper (HProblem prob, LPCWSTR name,  double
              fcnupper[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Constraint` type.)
`fcnlupper [fcnsize]`	Upper bound of range over which constraint dual values are valid. No upper bound is indicated with +1E30.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnMeanAbsDev

SolverFcnMeanAbsDev() returns the average of the absolute deviations of the specified uncertain function sample values from their mean. This is also known as Mean Absolute Deviation (MAD), especially in time series analysis applications. It is defined as:

$$MAD(X) = \frac{1}{n}\sum_{i=1}^{n}|\mu - x_i|$$

Here μ is the mean of the sample values.

Syntax:

```
SolverFcnMeanAbsDev (HProblem prob, LPCWSTR name,  double*
              values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`values`	Average of the absolute deviations of the specified uncertain function sample values from their mean.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnSemiVariance

SolverFcnSemiVariance() returns the semivariance for the specified uncertain function computed relative to the *target* argument. This is a measure of the dispersion of values of an uncertain function, but unlike the variance which measures (or penalizes) both positive and negative deviations from the target, the semivariance or lower partial moment is only concerned with *one-sided* deviations from the target. It is usually used in finance and insurance applications, when we are only concerned with downside risks (or loss in portfolio value). The semivariance is computed by summing only the downside differences from the target of all the trials, raised to the given power *q*, divided by the number of trials:

$$\frac{1}{n}\sum_{i=1}^{n}\left(t - x_i\right)_+^q$$

$$\left(x\right)_+ = \max\left(x, 0\right)$$

t = target value

All trials – not just the trials with downside deviations – are included in *n*. If *q* is different from 2, the result is called the 'lower partial moment.'

Syntax:

```
SolverFcnSemiVariance (HProblem prob, LPCWSTR name,  int*
              pnPower, double* pfTarget, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`pnPower`	The power argument, q in the above function.
`pfTarget`	Value to which all sampled values are compared in the computation (t in the above function).
`Values`	Returns the measure of the dispersion of values of an uncertain function.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnSemiDeviation

SolverFcnSemiDeviation() returns the semideviation for the specified uncertain function, relative to the *target*. This is a *one-sided* measure of dispersion of values of the uncertain function. The semideviation is the square root of the semivariance, described directly below. If a *q* argument different from 2 is specified, PsiSemiDev() returns the *q*th root of the lower partial moment at power *q* of the uncertain function.

Syntax:

```
SolverFcnSemiDeviation (HProblem prob, LPCWSTR name,  int*
              pnPower, double* pfTarget, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`pnPower`	The power argument, q.
`pfTarget`	Value to which all sampled values are compared in the computation (t in the above function).
`values`	Returns the semideviation for the specified uncertain function, relative to the pfTarget argument.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnValueAtRisk

SolverFcnValueAtRisk() returns the Value at Risk for the specified uncertain function at the specified 'confidence level,' which is better described as a *percentile* – for example, 0.95 or 0.99.

In finance applications, the Value at Risk is the maximum loss that can occur at a given confidence level. In a distribution of returns or profits, losses would lie at the "left end" of the distribution and would be represented by negative numbers and smaller percentiles (say 0.01 or 0.05). But it is customary in Value at Risk analysis to treat losses as positive numbers at the "right end" of the distribution.

Syntax:

```
SolverFcnValueAtRisk (HProblem prob, LPCWSTR name, double
              percentile, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`percentile`	Specified "confidence level".
`values`	Value at Risk for the specified uncertain function at the specified "confidence level" in the percentile argument.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnCondValueAtRisk

SolverFcnCondValueAtRisk() returns the conditional Value at Risk for the specified uncertain function, at the specified 'confidence level' which is better described as a *percentile* – for example, 0.95 or 0.99. The conditional Value at Risk is defined as the *expected value* of a loss *given that* a loss at the specified percentile occurs.

Syntax:

```
SolverFcnValueAtRisk (HProblem prob, LPCWSTR name, double
              percentile, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`percentile`	Specified "confidence level".
`values`	Conditional Value at Risk for the specified uncertain function at the specified "confidence level" in the percentile argument.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnMeanCI

SolverFcnMeanCI() returns the confidence "half-interval" for the estimated mean value for the specified uncertain function, at the specified *confidence level* (for example 0.95 or 0.99). If μ is the mean value returned by SolverFcnStatitics() and δ is the value returned by SolverFcnMeanCI(), the true mean is estimated to lie within the interval $\mu - \delta$ to $\mu + \delta$.

The *confidence level* can be interpreted as follows: If we compute a large number of independent estimates of confidence intervals on the true mean of the uncertain function, each based on *n* observations with *n* sufficiently large, then the proportion of these confidence intervals that contain the true mean of the function should equal the *confidence level*.

If σ^2 (*n*) is the sample variance from *n* trial values, $\alpha = 1 -$ *confidence level*, and $t_{n-1, 1-\alpha/2}$ is the upper $1-\alpha/2$ critical point of the Student's t-distribution with *n*-1 degrees of freedom, the confidence half-interval δ is computed as:

$$t_{n-1,1-\alpha/2}\sqrt{\frac{\sigma^2(n)}{n}}$$

The confidence interval measures the precision with which we have estimated the true mean. Larger half widths imply that there is a lot of variability in our estimates. The above formula for the half-width assumes that the individual x_is are normally distributed; when this is not the case, the above formula still gives us an *approximate* confidence interval on the true mean of the uncertain function.

Syntax:

```
SolverFcnMeanCI (HProblem prob, LPCWSTR name, double confidence,
                 double* values)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`confidence`	Specified "confidence level" (for example 0.95 or 0.99).
`values`	Confidence "half-interval" for the estimated mean value for the specified uncertain function, at the specified *confidence level*.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnStdDevCI

SolverFcnStdDevCI() returns the confidence 'half-interval' for the estimated standard deviation of the simulation trials (returned by the SolverFcnSemiDeviation() function) for the specified uncertain function, at *confidence level* (for example 0.95 or 0.99). If σ is the value returned by SolverFcnSemiDeviation () and δ is the value returned by SolverFcnDevCI(), the true mean is estimated to lie within the interval σ - δ to σ + δ.

If σ^2 (n) is the sample variance from n trial values, α = 1 – *confidence level*, and $t_{n-1, 1-\alpha/2}$ is the upper 1-α/2 critical point of the Student's t-distribution with n-1 degrees of freedom, the confidence half-interval δ is computed as:

$$t_{n-1,1-\alpha/2}\sigma(n)\sqrt{\frac{k-1}{4(n-1)}}$$

See also the description of the SolverFcnMeanCI() function.

Syntax:

```
SolverFcnMeanCI (HProblem prob, LPCWSTR name, double confidence,
                 double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`confidence`	Specified "confidence level" (for example 0.95 or 0.99).
`values`	Confidence 'half-interval' for the estimated standard deviation of the simulation trials for the specified uncertain function, at *confidence level* (for example 0.95 or 0.99).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnCITrials

SolverFcnCITrials() returns the estimated number of simulation trials needed to ensure that the specified uncertain function sample mean value (returned by the SolverFcnStatistics() function) lies within the confidence interval specified by *confidence level* (for example 0.95 or 0.99) and the half-interval size given by *tolerance*. Note that this number of trials is sufficient only to ensure that the single output value specified lies within the confidence interval. To ensure that *N* output cells lie within confidence intervals at level 1 - α (e.g. 0.95 = 1 – 0.05), use a confidence level of α / N.

Syntax:

```
SolverFcnCITrials (HProblem prob, LPCWSTR name, double
                confidence, double tolerance, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`confidence`	Specified "confidence level" (for example 0.95 or 0.99).
`tolerance`	Half-Interval size.
`values`	Estimated number of simulation trials needed to ensure that the specified uncertain function sample mean value (returned by the SolverFcnStatistics() function) lies within the confidence interval specified by *confidence level* (for example 0.95 or 0.99) and the half-interval size given by *tolerance*.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnTarget

SolverFcnTarget() returns the cumulative frequency of the target value in the distribution of trial values for the specified uncertain function. This function returns the proportion of simulated values that are less than or equal to the *target value*.

Syntax:

```
SolverFcnTarget (HProblem prob, LPCWSTR name, double
              targetvalue, double* values)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`targetvalue`	The function returns the proportion of simulated values that are less than or equal to this value.
`values`	Cumulative frequency of the target value in the distribution of trial values for the specified uncertain function

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnChanceTypeGet

SolverFcnChanceTypeGet() retrieves the chance type of a function of type chance. The chance type applies to functions of type Function_Type_Chance or Function_Type_Objective. If the function is of type Function_Type_Chance, there are three types of chance, Var, CVar, and USet. If the function is of type Function_Type_Objective, there are five types, Normal, Expected, Var, CVar, and USet.

Syntax:

```
SolverFcnChanceTypeGet (HProblem prob, LPCWSTR name,
                _Chance_Type* chancetype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`Chancetype`	The chance type of a function of type chance.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnChanceTypeSet

SolverFcnChanceTypeSet() sets the chance type of chance constraint. The chance type applies to functions of type Function_Type_Chance or Function_Type_Objective. If the function is of type Function_Type_Chance, there are three types of chance, Var, CVar, and USet. If the function is of type Function_Type_Objective, there are five types, Normal, Expected, Var, CVar, and USet.

Syntax:

```
SolverFcnChanceTypeSet (HProblem prob, LPCWSTR name,
                _Function_Type fcntype)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`Fcntype`	The chance type of a function of type chance.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnChanceGet

SolverFcnChanceGet() retrieves the type of chance constraint.

Syntax:

```
SolverFcnChanceGet (HProblem prob, LPCWSTR name, double* chance)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`Chance`	The type of the current function vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnChanceSet

SolverFcnChanceSet() retrieves the type of chance constraint.

Syntax:

```
SolverFcnChanceGet (HProblem prob, LPCWSTR name, double chance)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`Chance`	The type of the current function vector.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnInitValueGet

SolverFcnInitValueGet() retrieves the initial value of the function.

Syntax:

```
SolverFcnInitValueGet (HProblem prob, LPCWSTR name, double
              value[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`value[]`	Array of initial function values.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverFcnInitValueSet

SolverFcnInitValueSet() sets the initial value of the function. These values are overwritten when the SolverOptimize() function is executed.

Syntax:

```
SolverFcnInitValueSet (HProblem prob, LPCWSTR name, double
              value[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Handle to the problem, passed by the user.
`name`	Name of the vector of functions. (Must be of `_Function_Type_Uncertain` type.)
`value[]`	Array of initial function values.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Environment Method Functions

Click to navigate to the desired topic:

- SolverEnvInit
- SolverEnvFree
- SolverEnvRuntimeIDGet
- SolverEnvRuntimeIDSet
- SolverEnvLockCodeGet
- SolverEnvLicenseGet
- SolverEnvLicenseSet
- SolverEngTrialPeriod
- SolverEnvLicenseFileGet
- SolverEnvLicenseFileSet
- SolverEnvExecutableLocation

SolverEnvInit

SolverEnvInit() is a function that initializes the environment variable.

Syntax:

```
SolverEnvInit (Henv* env, LPCWSTR RuntimeID)
```

Parameters:

Parameter Name	Parameter Description
env	Name of the environment variable.
RuntimeID	Runtime ID of the environment variable.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

SolverEnvFree

SolverEnvFree() is a function that frees a previously created environment variable.

Syntax:

```
SolverEnvFree (HEnv* env)
```

Parameters:

Parameter Name	Parameter Description
env	Name of the environment variable.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEnvRuntimeIDGet

SolverEnvRuntimeIDGet() is a function that retrieves the runtime ID of the environment variable.

Syntax:

```
SolverEnvRuntimeIDGet (HEnv env, LPCWSTR* RuntimeID)
```

Parameters:

Parameter Name	Parameter Description
`env`	Name of the environment variable.
`RuntimeID`	Runtime ID of the environment variable.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEnvRuntimeIDSet

SolverEnvRuntimeIDSet() is a function that sets the runtime ID of the environment variable.

Syntax:

```
SolverEnvRuntimeIDSet (HEnv env, LPCWSTR RuntimeID)
```

Parameters:

Parameter Name	Parameter Description
`env`	Name of the environment variable.
`RuntimeID`	Runtime ID of the environment variable.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEnvLockCodeGet

SolverEnvLockCodeGet() is a function that gets the lock code of the machine running the SDK.

Syntax:

```
SolverEnvLockCodeGet (HEnv env, _License_SelectorCode
              nSelectorCode, LPCWSTR* lockcode)
```

Parameters:

Parameter Name	Parameter Description
`env`	Name of the environment variable.
`mSelectorCode`	The selector code for the license. For more information, refer to enum License_SelectorCode.
`lockcode`	The lock code of the machine running the SDK.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEnvLicenseGet

SolverEnvLicenseGet() is a function that retrieves the Solver Platform usage license.

Syntax:

```
SolverEnvLicenseGet(HEnv env, _License_Loc* licenseloc, LPCWSTR*
                    license)
```

Parameters:

Parameter Name	Parameter Description
`Env`	Name of the environment variable.
`licenseloc`	Location of the license string specified via the `licenseloc` enum.
`license`	The license string.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEnvLicenseSet

SolverEnvLicenseSet() is a function that sets the Solver Platform usage license by adding a license string to the license file.

Syntax:

```
SolverEnvLicenseSet(HEnv env, _License_Loc licenseloc, LPCWSTR
                license)
```

Parameters:

Parameter Name	Parameter Description
`Env`	Name of the environment variable.
`licenseloc`	Location of the license string specified via the `licenseloc` enum.
`license`	The license string.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEngTrialPeriod

SolverEngTrialPeriod() is a function that returns the number of days remaining before the expiration date of the license for a particular Engine.

Syntax:

```
SolverEngTrialPeriod(HProblem prob, LPCWSTR enginename, int*
              trialdaysleft)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Handle to the problem, passed by the user.
`Enginename`	Solver Engine standard name.
`trialdaysleft`	Number of seconds until the Engine's trial license expires.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEnvLicenseFileGet

SolverEnvLicenseFileGet() retrieves the location of the license file. (This function must be called before a Problem is created.)

Syntax:

```
SolverEnvLicenseFileGet(HEnv env, LPCWSTR* lpszLicFile)
```

Parameters:

Parameter Name	Parameter Description
`Env`	Name of the environment variable.
`lpszLicFile`	Location of the license file.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEnvLicenseFileSet

SolverEnvLicenseFileSet() sets the location of the license file. (This function must be called before a Problem is created.)

Syntax:

```
SolverEnvLicenseFileSet(HEnv env, LPCWSTR lpszLicFile)
```

Parameters:

Parameter Name	Parameter Description
`env`	Name of the environment variable.
`lpszLicFile`	Location of the license file (i.e. "c:\\Test\\Solver.lic").

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverEnvExecutableLocation

SolverEnvExecutableLocation() returns the location of file, solversdk.dll.

Syntax:

```
SolverEnvExecutableLocation(HEnv env, LPCWSTR* location)
```

Parameters:

Parameter Name	Parameter Description
`env`	Name of the environment variable.
`location`	Location of the file, solversdk.dll.

Requirements:

Compile with: ProceduralSDK.h
Link with: SolverSDK.lib
Run with: SolverSDK.dll

Probability Distribution Method Functions

Click to navigate to the desired topic:

- SolverModDistAutoFit
- SolverModDistCDF
- SolverModDistCDFInv
- SolverModDistPDF

SolverModDistAutoFit

SolverModDistAutoFit() is a probability distribution method function that automatically fits a function or a variable sample to the most suitable distribution.

Syntax:

```
SolverModDistAutoFit(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double SigLevel, BOOL bContinuous, BOOL
              bFitAll))
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`SigLevel`	Level of statistical significance of the autofit outcome to the resulting type of distribution. (1 – α)
`bContinuous`	Indicates whether to fit to continuous (= `true`) or discrete distribution.
`bFitAll`	Indicates whether to enforce fitting to all available distributions (= `true`).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistCDF

SolverModDistCDF() is a probability distribution method function that returns the Cumulative Distribution Function value for a function or variable.

Syntax:

```
SolverModDistCDF(HProblem prob, _Dist_VarFcn  varfcn, int index,
                 double x, double* f)
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
varfcn	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
index	Index of the variable or the function in the corresponding collection. (Zero-based.)
x	Cumulative distribution argument.
f	Computed cumulative distribution value.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistCDFInv

SolverModDistCDFInv() is a probability distribution method function that returns the Inverse Cumulative Distribution function value for a function or variable.

Syntax:

```
SolverModDistCDFInv(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double f, double* x)
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
varfcn	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
index	Index of the variable or the function in the corresponding collection. (Zero-based.)
f	Inversed cumulative distribution argument. (Belongs to the set of values of CDF.)
x	Computed inversed cumulative distribution value. (Belongs to the domain of CDF.)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistPDF

SolverModDistPDF() is a probability distribution method function that returns the Probability Density Function value for a function or variable.

Syntax:

```
SolverModDistPDF(HProblem prob, _Dist_VarFcn  varfcn, int index,
                 double x, double* f)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`x`	Probability density argument
`f`	Computed probability density value

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Simulation Distribution Arguments Functions

Click to navigate to the desired topic:

- SolverModDistControlParamsGet
- SolverModDistControlParamsSet

SolverModDistControlParamsGet

SolverModDistControlParamsGet() is a simulation distribution arguments function that returns the control parameters for the distribution of a function or variable.

Syntax:

```
SolverModDistControlParamsGet(HProblem prob, _Dist_VarFcn
              varfcn, int index, double controlparams[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`controlparams`	Retrieves the control parameters. (in order shift, lower cut, upper cut)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistControlParamsSet

SolverModDistControlParamsSet() is a simulation distribution arguments function that sets the control parameters for the distribution of a function or variable.

Syntax:

```
SolverModDistControlParamsSet(HProblem prob, _Dist_VarFcn
              varfcn, int index, double controlparams[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`controlparams`	Sets the control parameters in the following order: shift, lower cut, upper cut.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Fitting Method Functions

Click to navigate to the desired topic:

- SolverModDistFit
- SolverModDistInitSample

SolverModDistFit

SolverModDistFit() is a fitting method function that fits a function or a variable sample to a given distribution.

Syntax:

```
SolverModDistFit(HProblem prob, _Dist_VarFcn  varfcn, int index,
                 _DistFit_Type Type, double SigLevel, BOOL bShift)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`Type`	Distribution type to which the sample is fit. For more information, refer to enum DistFit_Type.
`SigLevel`	Level of statistical significance of the fit outcome to the resulting type of distribution. (1 – α)
`bShift`	If true, the data will be shifted for better fitting by one-dimensional optimization.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistInitSample

SolverModDistInitSample() is a fitting method function that initializes a sample for the distribution of a function or a variable.

Syntax:

```
SolverModDistInitSample(HProblem prob, _Dist_VarFcn  varfcn, int
              index, int size, double sample[], int occurs[])
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
varfcn	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
index	Index of the variable or the function in the corresponding collection. (Zero-based.)
size	Length of Sample and Occurs must be the same.
sample	Sample one-dimensional array. if Sample = Occurs = NULL, initialization generates random Sample from the current DistType.
occurs	Number of occurrences for each element in Sample. If NULL, Size and Sample are the same as in getSample and getSize.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Fitting Quality Criteria Functions

Click to navigate to the desired topic:

- SolverModDistFitLimit
- SolverModDistFitSigLevel
- SolverModDistFitStatistic
- SolverModDistFitSuccess
- SolverModDistFitTest
- SolverModDistIndepTestGet

SolverModDistFitLimit

SolverModDistFitLimit() is a fitting quality criteria function that retrieves the critical fitting value adjusted to the sample size. If not exceeded by the computed test value, fitting is successful.

Syntax:

```
SolverModDistFitLimit(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double* fitlimit)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`Index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`fitlimit`	The critical fitting value to be retrieved.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistFitSigLevel

SolverModDistFitSigLevel() is a fitting quality criteria function that retrieves the significance level achieved to a particular distribution type as a result of a fitting operation. The significance level known as "1–α" (usually a number between 0.9 and 0.99).

Syntax:

```
SolverModDistFitSigLevel(HProblem prob, _Dist_VarFcn  varfcn,
              int index, double* siglevel)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`siglevel`	Level of statistical significance of the fit outcome to the resulting type of distribution.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistFitStatistic

SolverModDistFitStatistic() is a fitting quality criteria function that retrieves the fitting statistic dependent on the sample's size. The statistic is Kolmogorov-Smirnof for continuous and Chi-squared for discrete distributions.

Syntax:

```
SolverModDistFitStatistic(HProblem prob, _Dist_VarFcn  varfcn,
              int index, double* fitstatistic)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`fitstatistic`	Statistic retrieved. (Kolmogorov-Smirnof for continuous distributions or Chi-squared is for discrete distributions.)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistFitSuccess

SolverModDistFitSuccess() is a fitting quality criteria function that returns a Boolean value indicating if the distribution fit was successful.

Syntax:

```
SolverModDistFitSuccess(HProblem prob, _Dist_VarFcn  varfcn, int
              index, BOOL* fitsuccess)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`fitsuccess`	Retrieves the fitting result. `true` means fitting has succeeded and the result is good (assuming independent data in the sample).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistFitTest

SolverModDistFitTest() is a fitting quality criteria function that retrieves statistics adjusted to the sample's size. If the test value is less than the limit value, then the fitting is successful.

Syntax:

```
SolverModDistFitTest(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double* fittest)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`fittest`	Fitting statistic adjusted to the sample's size. If less than `fitlimit`, the fitting is considered successful.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistIndepTestGet

SolverModDistIndepTestGet() is a fitting quality criteria function that retrieves a value indicating the appropriateness of the fitting (`indeptest`). Fitting is appropriate if values to fit (i.e. the sample) are independent (uncorrelated). A small positive number for `indeptest` indicates independent values.

Syntax:

```
SolverModDistIndepTestGet(HProblem prob, _Dist_VarFcn  varfcn,
              int index, double* indeptest)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`indeptest`	Value indicating the appropriateness of the fitting. Small positive number indicates uncorrelated data in the sample, otherwise the fitting may not be valid.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Initialization Method Function

SolverModDistInit

SolverModDistInit() is an initialization method function that initializes the distribution for a function or a variable.

Syntax:

```
SolverModDistInit(HProblem prob, _Dist_VarFcn  varfcn, int
              index, LPCWSTR lpszDistribDecl)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`lpszDistribDecl`	Distribution function declaration e.g. `PsiNormal(0,1)`.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Distribution Parameter Functions

Click to navigate to the desired topic:

- SolverModDistKurtosis
- SolverModDistMean
- SolverModDistMedian
- SolverModDistMode
- SolverModDistSkewness
- SolverModDistVariance

SolverModDistKurtosis

SolverModDistKurtosis() is a distribution parameter function that returns the analytic kurtosis for the distribution of a function or variable.

Syntax:

```
SolverModDistKurtosis(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double* kurtosis)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`kurtosis`	Computed distribution kurtosis value

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistMean

SolverModDistMean() is a distribution parameter function that returns the analytic mean value for the distribution of a function or variable.

Syntax:

```
SolverModDistMean(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double* mean)
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
varfcn	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
index	Index of the variable or the function in the corresponding collection. (Zero-based.)
mean	Computed distribution mean value

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistMedian

SolverModDistMedian() is a distribution parameter function that returns the analytic median value for the distribution of a function or variable.

Syntax:

```
SolverModDistMedian(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double* median)
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
varfcn	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
index	Index of the variable or the function in the corresponding collection. (Zero-based.)
median	Computed distribution median value

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistMode

SolverModDistMode() is a distribution parameter function that returns the analytic mode value for the distribution of a function or variable.

Syntax:

```
SolverModDistMode(HProblem prob, _Dist_VarFcn  varfcn, int
                index, double* mode)
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
varfcn	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
index	Index of the variable or the function in the corresponding collection. (Zero-based.)
mode	Computed distribution mode value

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistSkewness

SolverModDistSkewness() is a distribution parameter function that returns the analytic skewness for the distribution of a function or a variable.

Syntax:

```
SolverModDistSkewness(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double* skewness)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`skewness`	Computed distribution skewness value

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistVariance

SolverModDistVariance() is a distribution parameter function that returns the analytic variance for the distribution of a function or a variable.

Syntax:

```
SolverModDistVariance(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double* variance)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`variance`	Computed distribution variance value

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Distribution Arguments Functions

Click to navigate to the desired topic:

- SolverModDistNumControlParams
- SolverModDistNumParams
- SolverModDistParamsGet
- SolverModDistParamsSet

SolverModDistNumControlParams

SolverModDistNumControlParams() is a distribution arguments function that returns the number of control parameters for the distribution of a function or variable.

Syntax:

```
SolverModDistNumControlParams(HProblem prob, _Dist_VarFcn
              varfcn, int index, int* numcontrolparams)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`numcontrolparams`	Retrieves the number of control parameters. (Three in the current implementation.)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistNumParams

SolverModDistNumParams() is a distribution arguments function that returns the number of parameters for the distribution of a function or variable.

Syntax:

```
SolverModDistNumParams(HProblem prob, _Dist_VarFcn varfcn, int
              index, int* numparams)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`numparams`	Retrieves the number of distribution parameters.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistParamsGet

SolverModDistParamsGet() is a distribution arguments function that returns the parameters for the distribution of a function or variable.

Syntax:

```
SolverModDistParamsGet(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double distparams[])
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
varfcn	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
index	Index of the variable or the function in the corresponding collection. (Zero-based.)
distparams	Retrieves the distribution parameters (location, shape, scale etc.).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistParamsSet

SolverModDistParamsSet() is a distribution arguments function that sets the parameters for the distribution of a function or variable.

Syntax:

```
SolverModDistParamsSet(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double distparams[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`distparams[]`	Sets the distribution parameters (location, shape, scale etc.).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Sample Management Functions

Click to navigate to the desired topic:

- SolverModDistSampleGet
- SolverModDistSampleSet
- SolverModDistSizeGet
- SolverModDistSizeSet

SolverModDistSampleGet

SolverModDistSampleGet() is a sample management function that retrieves the sample for a distribution of a function or variable.

Syntax:

```
SolverModDistSampleGet(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double sample[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`sample`	Sample as one-dimensional array of real values

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistSampleSet

SolverModDistSampleSet() is a sample management function that sets the sample for a distribution of a function or variable.

Syntax:

```
SolverModDistSampleSet(HProblem prob, _Dist_VarFcn  varfcn, int
              index, double sample[])
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`sample`	Sample as one-dimensional array of real values

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistSizeGet

SolverModDistSizeGet() is a sample management function that returns the size of the sample for a distribution of a function or a variable.

Syntax:

```
SolverModDistSizeGet(HProblem prob, _Dist_VarFcn  varfcn, int
              index, int* size)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`size`	Length of the sample

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverModDistSizeSet

SolverModDistSizeSet() is a sample management function that sets the size of the sample for a distribution of a function or a variable.

Syntax:

```
SolverModDistSizeSet(HProblem prob, _Dist_VarFcn  varfcn, int
              index, int size)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`varfcn`	Specifies whether the call applies to an uncertain variable or to an uncertain function. For more information, refer to enum Dist_VarFcn.
`index`	Index of the variable or the function in the corresponding collection. (Zero-based.)
`size`	Length of the sample

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Parameter Functions

Click to navigate to the desired topic:

- SolverParamCount
- SolverParamTypeGet
- SolverParamTypeSet
- SolverParamNameGet
- SolverParamNameSet
- SolverParamSizeGet
- SolverParamSizeSet
- SolverParamNumTicksGet
- SolverParamNumTicksSet
- SolverParamReturnValue
- SolverParamValue
- SolverParamAllValuesGet
- SolverParamAllValuesSet
- SolverParamLowerBoundGet
- SolverParamLowerBoundSet
- SolverparamUpperBoundGet
- SolverParamUpperBoundSet

SolverParamCount

SolverParamCount() returns the number of parameters on the problem object, for example, the number of PsiOptParams, PsiSenParams or PsiSimParams.

Syntax:

```
SolverParamCount(HProblem prob, int* pcount)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`pcount`	Number of input parameters in Problem.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamTypeGet

SolverParamTypeGet() returns the type of input parameter located at the cell address, or defined name, in the Excel workbook: Parameter_Type_Optimization, Parameter_Type_Simulation, or Parameter_Type_Sensitivity.

Syntax:

```
SolverParamTypeGet(HProblem prob, LPCWSTR name, _Parameter_Type*
                paramtype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Parameter is located.
`paramtype`	Variable which holds the Parameter_Type (Parameter_Type_Optimization, Parameter_Type_Simulation, or Parameter_Type_Sensitivity)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamTypeSet

SolverParamTypeSet() sets the type of input parameter located at the cell address, or defined name, in the Excel workbook: Parameter_Type_Optimization, Parameter_Type_Simulation, or Parameter_Type_Sensitivity.

Syntax:

```
SolverParamTypeSet(HProblem prob, LPCWSTR name, _Parameter_Type
                paramtype)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Parameter is located.
`paramtype`	Variable which holds the Parameter_Type (Parameter_Type_Optimization, Parameter_Type_Simulation, or Parameter_Type_Sensitivity)

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamNameGet

SolverParamNameGet() retreives the name of the input parameter in the Excel workbook for the index specified.

Syntax:

```
SolverParamNameGet(HProblem prob, int index, LPCWSTR* nameout)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Returns the problem instance handle.
`index`	The index of the input parameter.
`paramtype`	Holds the name of the input parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamNameSet

SolverParamNameGet() sets the name of the input parameter in the Excel workbook for the index specified.

Syntax:

```
SolverParamNameSet(HProblem prob, int index, LPCWSTR nameout)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`index`	The index of the input parameter.
`paramtype`	Holds the name of the input parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamSizeGet

SolverParamSizeGet() retrieves the number of input parameters located at the cell address or defined name given in the second argument, *name*.

Syntax:

```
SolverParamSizeGet(HProblem prob, LPCWSTR name, int* size)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Parameter is located.
`Size`	Number of input parameters in the `name` argument.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamSizeSet

SolverParamSizeSet() retrieves the number of input parameters located at the cell address or defined name given in the second argument, *name*.

Syntax:

```
SolverParamSizeSet(HProblem prob, LPCWSTR name, int size)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`Size`	Number of input parameters in the `name` argument.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamNumTicksGet

SolverParamNumTicksGet() retrieves the size of the generated values array. If using PsiOptParam in the Excel workbook, the number of ticks will be the number of optimizations. If using PsiSimParam in the Excel workbook, the number of ticks will be the number of simulations. If using PsiSenParam in the Excel workbook, the number of ticks is a defined value in the workbook.

Syntax:

```
SolverParamSizeSet(HProblem prob, LPCWSTR name, int* numticks)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`numticks`	Number of optimizations (PsiOptParam), simulations (PsiSimParam), or a defined number (PsiSenParam).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamNumTicksSet

SolverParamNumTicksSet() sets the size of the generated values array. Currently, this can only be used to set the defined number for a parameter of type Parameter_Type_Sensitivity. If using a parameter of type Parameter_Type_Optimization, set the number of optimizations to set the number of ticks. If using a parameter of type Parameter_Type_Simulation, set the number of simulations to set the number of ticks.

Syntax:

```
SolverParamSizeSet(HProblem prob, LPCWSTR name, int* numticks)
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
name	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
numticks	Sets the size of the generated values array for a paremeter of type Parameter_Type_Sensitivity.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamReturnValue

SolverParamReturnValue() returns the base case value of the Input Parameter in the Excel Workbook. If the input values to PsiOptParam, PsiSimParam or PsiSenParam are specified as ({6, 7, 8}) in the Excel workbook, this function will return "7". If the input values to PsiOptParam, PsiSimParam, or PsiSenParam are specified as (100, 80, 75) in the Excel workbook, this function will return "75".

Syntax:

```
SolverParamReturnValue(HProblem prob, LPCWSTR name, double
              value)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Returns the problem instance handle.
`Name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`Value[number of input parameters]`	Base case value of the Input Parameter in the Excel Workbook.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamValue

SolverParamValue() returns

Syntax:

```
SolverParamValue(HProblem prob, LPCWSTR name, double value)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Returns the problem instance handle.
`Name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`Value[number of input parameters]`	Base case value of the Input Parameter in the Excel Workbook.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamAllValuesGet

SolverParamAllValuesGet() returns the values (minimum, maximu, and base case) specified for the input parameter.

Syntax:

```
SolverParamAllValuesGet(HProblem prob, LPCWSTR name, double
              value)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Returns the problem instance handle.
`Name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`Value[number of values]`	Array of values (minimum, maximum, or base case) specified for the input parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamAllValuesSet

SolverParamAllValuesSet() sets the values (minimum, maximum, and base case) for the input parameter.

Syntax:

```
SolverParamAllValuesSet(HProblem prob, LPCWSTR name, double
              value)
```

Parameters:

Parameter Name	Parameter Description
`Prob`	Returns the problem instance handle.
`Name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`Value[number of values]`	Array of values (minimum, maximum, or base case) specified for the input parameter.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamLowerBoundGet

SolverParamLowerBoundGet() gets the minimum value of the input parameter.

Syntax:

```
SolverParamLowerBoundGet(HProblem prob, LPCWSTR name, double
              paramlower)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`paramlower[size]`	Minimum value of the input parameter(s).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamLowerBoundSet

SolverParamLowerBoundSet() sets the minimum value of the input parameter.

Syntax:

```
SolverParamLowerBoundSet(HProblem prob, LPCWSTR name, double
              paramlower)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`paramlower[size]`	Minimum value of the input parameter(s).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamUpperBoundGet

SolverParamUpperBoundGet() sets the minimum value of the input parameter.

Syntax:

```
SolverParamUpperBoundGet(HProblem prob, LPCWSTR name, double
              paramlower)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`paramupper[size]`	Maximum value of the input parameter(s).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverParamUpperBoundSet

SolverParamUpperBoundSet() sets the minimum value of the input parameter.

Syntax:

```
SolverParamUpperBoundSet(HProblem prob, LPCWSTR name, double
              paramlower)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`paramupper[size]`	Maximum value of the input parameter(s).

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Worksheet Input Functions

Click to navigate to the desired topic:

- SolverInputCount
- SolverInputName
- SolverInputValueGet
- SolverInputValueSet
- SolverInputSize

SolverInputCount

SolverInputCount() returns the number of PsiInput functions present in an Excel workbook.

Syntax:

```
SolverInputCount(HProblem prob, int* count)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`count`	Number of PsiInput functions present on the Excel workbook.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverInputName

SolverInputName() returns either the physical cell address or the defined names of the PsiInput values. For example, if PsiInput(A1:A2) exists on the Excel worksheet, SolverInputName() will return "A1:A2".

Syntax:

```
SolverInputName(HProblem prob, LPCWSTR* nameout)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`nameout`	Contents of the PsiInput function located on the Excel worksheet.

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverInputValueGet

SolverInputValueGet() returns the PsiInput values specified in the Excel worksheet. For example, if PsiInput(A1:A2) exists in an Excel workbook where A1 = 300 and A2 = 450, SolverInputValueGet() will return an array containing two elements, 300 and 450.

Syntax:

```
SolverInputValueGet(HProblem prob, LPCWSTR name, double value)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`value[]`	Array of PsiInput values. The size of the array should be equal to the return value (size) of SolverInputSize().

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverInputValueSet

SolverInputValueSet() sets the PsiInput values.

Syntax:

```
SolverInputValueSet(HProblem prob, LPCWSTR name, double value)
```

Parameters:

Parameter Name	Parameter Description
prob	Returns the problem instance handle.
name	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
value[]	Array of PsiInput values. The size of the array should be equal to the return value (size) of SolverInputSize().

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

SolverInputSize

SolverInputSize() retrieves the number of values specified by the PsiInput function on the Excel worksheet.

Syntax:

```
SolverInputSize(HProblem prob, LPCWSTR name, int* size)
```

Parameters:

Parameter Name	Parameter Description
`prob`	Returns the problem instance handle.
`name`	Either the Excel cell address or the defined name in the Excel model where the Input Parameter is located.
`size`	Number of values specified by the PsiInput function on the Excel worksheet. For example, if the function PsiInput(A1:A2) exists on the Excel worksheet, SolverInputSize() would return "2".

Requirements:

Compile with: ProceduralSDK.h

Link with: SolverSDK.lib

Run with: SolverSDK.dll

Functions Error Codes

All Procedural API functions return error codes that are the same as in the Object-Oriented API (specified in enum APIerr).

Functions Error Codes:

Error Code	Description
0	In the Solver SDK Platform, return code 0 always indicates the function has been successful.
1	Out of memory.
2	This feature not implemented in this version.
3	Invalid problem handle. A Problem must first be defined.
4	Problem definition is incomplete.
5	No license found.
6	No valid license found.
7	License has expired.
8	License usage exhausted.
9	Problem is invalid.
10	Problem type must be `Solver_Type_Maximize`, `Solver_Type_Minimize`, or `Solver_Type_FindFeas`.
11	Matrix is invalid. `Majorstart` or `Minorindex` is missing.
12	Matrix is invalid. `Majorstart` values must be increasing, and less than the total number "L" of elements in the matrix.
13	Matrix is invalid. `Minorindex` element is out of range.
14	Objective Type is invalid.
15	Engine Library could not be found.
16	Engine entry point could not be found.
17	Engine could not be loaded.
18	The selected Engine cannot solve a problem of this type. Please select another

	Engine.
19	Engine could not be found.
20	Multiple threads are not supported in this version.
21	Result is missing.
22	The group index for an alldifferent constraint must be 1 for the first constraint, 2 for the second, etc.
23	An alldifferent constraint must include at least 2 variables.
24	The lower bounds on the variables in an alldifferent constraint must be 1; the upper bounds must be equal to the number of variables included in the constraint.
25	Parameter does not exist.
26	Parameter value is out of range.
27	Invalid index into the Collection.
28	The group index for a cone constraint must be 1 for the first constraint, 2 for the second, etc.
29	A cone constraint must include at least 2 variables.
30	Function is not a member of the Function Collection.
31	Variable is not a member of the Variable Collection.
32	Variable is already in use by another problem.
33	Function is already in use by another problem.
34	Function name is already in use.
35	Function name is invalid.
36	Variable name is already in use.
37	Variable name is invalid.
38	Case is not a member of the Case Collection.
39	Invalid distribution.
40	Problem could not be saved.
41	You cannot unload one of the standard Engines.
42	Problem could not be loaded.
43	Array is being accessed outside its bounds.

44	Invalid value.
45	Collection cannot be changed at this time. (This code is thrown when a change made to a Collection could easily render a Problem invalid. For example, you set a matrix and try to remove a variable.
46	Feature becomes read-only after being assigned to a Problem.
47	Objects are assigned to different Problems.
48	Assignment between objects of different types.
49	Previously initialized object cannot be reinitialized.
50	Input array size does not match the calculated size.
51	Unexpected error. Please contact Technical Support.
52	Operation cannot be performed because the matrix is in fast update mode.
53	Invalid Statistics
54	Weight matrix has invalid values.
55	Invalid Distribution
56	Invalid distribution computed bounds.
57	Duplicate Name
58	Not associated array
59	Not associated Excel object model.
60	Invalid Excel range name.
61	Input is not a member of the Inputs collection.
62	Property or function not implemented.
63	Property or function not implemented.
64	Parameter name is already in use.
65	Parameter name is invalid.
66	Parameter is not a member of the Parameters collection.
67	Duplicated indices

Solver SDK Platform Legacy API Reference

Click to navigate to the desired section:

- Functions and Parameters Overview
- Problem Definition Functions
- Solution Functions
- Diagnostic Functions
- Use Control Functions
- Callback Functions for Nonlinear Problems
- Other Callback Functions
- Parameter Management Functions
- Solver Parameters

Functions and Parameters Overview

Click to navigate to the desired topic:

- Functions Overview
- Solver Parameters Overview

Functions Overview

Problem Definition Functions

Parameter Name	Description
`loadlp`	Defines a linear optimization problem.
`loadquad`	Loads the quadratic part of a problem once a problem has been defined with `loadlp`.
`loadctype`	Specifies a mixed-integer problem once a problem has been defined with `loadlp` or `loadnlp`.
`loadnlp`	Defines a non-linear optimization problem.
`loadnltype`	Specifies whether each variable occurs in a linear, nonlinear, or nonsmooth/discontinuous way in the function.
`testnltype`	Determines the problem type.
`unloadprob`	Unloads the current problem from the Solver DLL's memory.

Solution Functions

Parameter Name	Description
`optimize`	Solves the specified non-integer optimization problem.
`mipoptimize`	Solves the specified mixed-integer optimization problem..
`solution`	Retrieves the final status and values of the optimization.
`objsa`	For linear problems, returns the ranges of values over which the sensitivity analysis information is valid.
`rhssa`	For linear problems, returns the ranges of values over which the sensitivity analysis information is valid.
`varstat`	For nonsmooth optimization problems, returns statistical information about the variable values in the final population of candidate solutions.
`constat`	For nonsmooth optimization problems, returns statistical information about the constraint values in final population of candidate solutions.

Diagnostic Functions

Parameter Name	Description
findiis	Computes an Irreducibly Infeasible Subset (IIS) for a linear or quadratic problem.
getiis	Returns information about the specific constraints (rows) and variable bounds (columns) included in the IIS.
iiswrite	Writes a linear or quadratic problem together with its Irreducibly Infeasible Subset (IIS) to a text file.
lpwrite	Writes a linear or quadratic problem to a text file.
lpread	Creates a linear, quadratic or mixed-integer problem from a text file.

Use Control Functions

Parameter Name	Description
getuse	Returns information about the number of uses since the Solver DLL was placed in service.
reportuse	Retrieves Solver DLL license and usage information and either writes it in a text file to disk or sends an automated email to Frontline Systems.
getproblimits	Returns the optimization problem limits (available Engines, max. number of variables and constraints) with regards to the license purchased.

Callback Functions for Nonlinear Problems

Parameter Name	Description
funceval	Computes values for the problem functions (objective and constraints) for any given values of the variables.
jacobian	Computes the objective value and gradient and the Jacobian matrix of partial derivatives of the problem functions.

Other Callback Functions

Parameter Name	Description
setlpcallbackfunc	Specifies the callback function which the Solver DLL should call on each “pivot” or Simplex iteration.
getlpcallbackfunc	Retrieves the callback function which the Solver DLL calls on each “pivot” or Simplex iteration.
setmipcallbackfunc	Specifies the callback function which the Solver DLL should call on each “branch” or LP subproblem in a MIP problem.

`getmipcallbackfunc`	Retrieves the callback function which the Solver DLL calls on each “branch” or LP subproblem in a MIP problem.
`getcallbackinfo`	Returns information about the optimization from within a callback function.

Parameter Management Functions

Parameter Name	Description
`setintparam`	Sets the value of a specified integer parameter in a problem.
`getintparam`	Retrieves the current value of a specified integer parameter in a problem.
`infointparam`	Retrieves the minimum, maximum, and default values of a specified integer parameter in a problem.
`setdblparam`	Sets the value of a specified double parameter in a problem.
`getdblparam`	Retrieves the value of a specified double parameter in a problem.
`infodblparam`	Retrieves the minimum, maximum, and default values of a specified double parameter in a problem.
`setdefaults`	Sets all integer and double parameters for a specified problem to their default values.

Solver Parameters Overview

Integer Parameters (General)

Parameter Name	Description
PARAM_ARGCK	On argument errors, either returns the error number or displays a Windows message box describing the error.
PARAM_ARRAY	Sets the array type arguments supplied to the Solver (C-style arrays or Visual Basic or OLE-style SAFEARRAYs).
PARAM_USERP	Controls the reporting of "use" information in certain licensed versions of the Solver DLL.
PARAM_IISBND	Instructs the Solver DLL's IIS finder whether to skip the steps (and computing time) required to eliminate variable bounds from an Irreducibly Infeasible Set.
PARAM_ITLIM	The maximum number of iterations (pivots) that the Solver will perform on an NLP, LP or QP problem (or a subproblem of a MIP problem).

Integer Parameters (Solver Engines)

Parameter Name	Description
PARAM_SCAIND	Determines how the LP matrix, variable and constraint bounds are (re)scaled during the solution process.
PARAM_CRAIND	Determines whether the Solver should start by constructing an advanced or a normal basis.
PARAM_LINVAR	Determines whether the nonlinear Solver Engine will attempt to recognize variables occurring linearly in all of the problem functions.
PARAM_DERIV	Controls the method used by the Solver to compute approximate first partial derivatives.
PARAM_ESTIM	Controls the method used to estimate initial values for the basic variables at the beginning of each one-dimensional line search.
PARAM_DIREC	Controls the method used by the Solver to determine a direction vector for the variables being changed on a given iteration.
PARAM_SAMPSZ	Determines the number of points in the Evolutionary Solver's population of candidate solutions.
PARAM_NOIMP	Controls a stopping criteria used by the Evolutionary Solver Engine.

Integer Parameters (Mixed-Integer Problems)

Parameter Name	Description
PARAM_NDLIM	The maximum number of nodes that the Branch & Bound algorithm will fathom on a MIP problem.
PARAM_MIPLIM	The maximum number of feasible integer solutions that the Branch & Bound algorithm will find on a MIP problem.
PARAM_RELAX	Controls whether the Solver will ignore the integer variables in order to solve a "relaxation" version of a mixed-integer programming problem.
PARAM_PREPRO	Controls the Probing strategy for MIP problems (in the Solver DLL Plus).
PARAM_OPTFIX	Controls the Optimality Fixing strategy for MIP problems (in the Solver DLL Plus).
PARAM_REORDR	Controls the Variable Reordering strategy for MIP problems (in the Solver DLL Plus).
PARAM_IMPBND	Controls the Bounds Improvement strategy for MIP problems (in the Solver DLL Plus).

Double Parameters

Parameter Name	Description
PARAM_TILIM	The maximum number of seconds for the Solver to spend on the overall LP or MIP problem.
PARAM_EPOPT	The "optimality" or "reduced cost" tolerance for LP problems.
PARAM_EPPIV	The "pivot tolerance" for LP problems.
PARAM_EPSOL	The "solution tolerance" in the Large-Scale LP/MIP Solver DLL.
PARAM_EPRHS	The "feasibility tolerance" in MIP problems.
PARAM_EPGAP	The (relative) integer objective "gap tolerance" for MIP problems.
PARAM_CUTLO	The "integer objective cutoff value" for MIP maximization problems.
PARAM_CUTHI	The "integer objective cutoff value" for MIP minimization problems.
PARAM_EPNEWT	The tolerance within which constraints are considered "binding," or satisfied with equality.
PARAM_EPCONV	Change tolerance used for the "slowly changing objective" test in the nonlinear GRG Solver and the Evolutionary Solver
PARAM_MUTATE	The probability used by the Evolutionary Solver Engine when generating a

new trial point.

Problem Definition Functions

Click to navigate to the desired topic:

- loadlp
- loaquad
- loadctype
- loadnlp
- loadnltype
- testnltype
- unloadprob

loadlp

loadlp() is a function that defines a linear optimization problem.

`loadlp()` returns `NULL` if there is an error in its arguments, if linear problems are not supported by this version of the Solver DLL, if the `probname` key string is not recognized, or in the single-threaded version of the DLL, if another problem is still active (i.e. if `loadlp()` or `loadnlp()` has been called without a corresponding call to `unloadprob()`.). You should not change or free the memory for arguments passed to `loadlp()` until after you make the corresponding call to `unloadprob()`.

Syntax:

```
HPROBLEM loadlp (probname, numcols, numrows, objsen, obj, rhs,
                 sense, matbeg, matcnt, matind, matval, lb, ub,
                 rngval, colspace, rowspace, nzspace)
LPSTR probname;
INTARG numcols, numrows, objsen;
LPREALARG obj, rhs;
LPBYTEARG sense;
LPINTARG matbeg, matcnt;
HPINTARG matind;
HPREALARG matval;
LPREALARG lb, ub, rngval;
INTARG colspace, rowspace, nzspace;
```

Parameters:

Parameter Name	Parameter Description
`probname`	A character string (currently 16 characters plus a 0 terminator byte) containing a unique "key" which is assigned to you when you license the Solver DLL from Frontline Systems. The DLL that is licensed to you recognizes your specific key string; if it is not supplied when `loadlp()` or `loadnlp()` is called, these functions will return NULL, and calls to these and other functions will not be successful.
`numcols`	The number of columns in the constraint matrix (i.e. the number of variables).
`numrows`	The number of rows in the constraint matrix, not including the objective function or bounds on the variables.
`objsen`	Indicates minimization or maximization: 1 = minimize, -1 = maximize.
`obj`	An array (of dimension `numcols`) containing the objective function coefficients.

rhs	An array (of dimension numrows) containing the constant or "right-hand side" term for each row in the constraint matrix. See also the rngval argument below.
sense	An array (of dimension numrows) containing the sense of each constraint (row): sense[i] = 'L' ≤ constraint sense[i] = 'E' = constraint sense[i] = 'G' ≥ constraint sense[i] = 'R' ranged constraint Note that the value 'R' cannot be used with the Small-Scale Solver DLL.
matbeg matcnt matind matval	These four arguments describe the nonzero elements of the constraint matrix. You may pass NULL values for matbeg, matcnt, and matind if your matval array is a full-size dense matrix, i.e. consisting of numcols * numrows elements (where the elements for each column are consecutive). If matbeg, matcnt and matind are not NULL values, the array matval contains the nonzero coefficients grouped by column (i.e. all coefficients for the same variable are consecutive). matbeg[i] contains the index in matval of the first coefficient for column i. matcnt[i] contains the number of coefficients in column i. Note that the indices in matbeg must be in ascending order. Each entry matind[i] is the row number of the corresponding coefficient in matval[i]. The entries in matind are not required to be in ascending row order. The coefficient M[i,j] in the full constraint matrix (if it is nonzero) would be stored in matval[matbeg[j]+k] where k is between 0 and matcnt[j]-1 (inclusive) and the corresponding entry matind[matbeg[j]+k] = i, the row index.
lb	An array (of dimension numcols) containing the lower bound on each variable. A lower bound less than or equal to -INFBOUND will be treated as "minus infinity."
ub	An array (of dimension numcols) containing the upper bound on each variable. A upper bound greater than or equal to +INFBOUND will be treated as "plus infinity."
rngval	An array (of dimension rows) containing the range value of each constraint right-hand side. Ranged rows are specified in the sense array. If the row is not ranged, the corresponding rngval element should be 0.0. A range value for row i means that the constraint left-hand side must be between rhs[i] and rhs[i] + rngval[i]. rngval[i] may be positive or negative. If there are no ranged rows at all, rngval may be NULL (it is ignored when the Small-Scale Solver DLL is used).
colspace rowspace nzspace	These three arguments are included for compatibility with the Large-Scale Solver DLL. In the Small-Scale Solver DLL, colspace should be equal to cols; rowspace should be equal to rows; and nzspace should be equal to the number of entries in matval. When PARAM_ARRAY is 1, the actual size of the matval SAFEARRAY is checked and must match the value of nzspace.

loaquad

loadquad() is a function that loads the quadratic part of a problem once a problem has been defined with `loadlp`.

`loadquad()` returns 0 unless there is an error in its arguments, in which case it returns an integer value indicating the ordinal position of the first argument found to be in error. `objsa()` and `rhssa()` (which return LP range information) may not be used when `loadquad()` has been called.

Syntax:

```
INTARG loadquad (lp, qmatbeg, qmatcnt, qmatind, qmatval,
                 qnzspace, var)
HPROBLEM lp;
LPINTARG qmatbeg, qmatcnt;
HPINTARG qmatind;
HPREALARG qmatval;
INTARG qnzspace;
LPREALARG var;
```

Parameters:

Parameter Name	Parameter Description
`lp`	Handle to the current problem as returned by a previous call to `loadlp()`.

`qmatbeg`
`qmatcnt`
`qmatind`
`qmatval`
	These four arguments describe the nonzero elements of the Q matrix, which provides the coefficients of the quadratic portion of the objective function. (The coefficients of the linear portion of the objective are still specified with the `obj` argument of `loadlp()`.) They are analogous to the `matxxx` arguments of `loadlp()`; however both the row and column indices refer to decision variables, which are multiplied together and by the coefficient, then summed to form the objective.
	
	You may pass NULL values for `qmatbeg`, `qmatcnt`, and `qmatind` if your `qmatval` array is a full-size dense matrix, i.e. consisting of `numcols` * `numcols` elements – which is often the case for quadratic problems. In this case, the elements for each matrix column should be consecutive.
	
	If `qmatbeg`, `qmatcnt` and `qmatind` are not NULL values, the array `qmatval` contains the nonzero coefficients grouped by column (i.e. all coefficients for the same variable are consecutive). `qmatbeg[i]` contains the index in `qmatval` of the first coefficient for column i. `qmatcnt[i]` contains the number of coefficients in column i. Note that the indices in `qmatbeg[]` must be in ascending order. Each entry `qmatind[i]` is the row number of the corresponding coefficient in `qmatval[i]`. The entries in `qmatind[]` are not

	required to be in ascending row order. The coefficient `M[i,j]` in the `Q` matrix (if it is nonzero) would be stored in `qmatval[qmatbeg[j]+k]` where k is between 0 and `qmatcnt[j]`-1 (inclusive) and the corresponding entry `qmatind[qmatbeg[j]+k] = i`, the row index.
`qnzspace`	The number of elements in the arrays `qmatind` and `qmatval`. If `PARAM_ARRAY` is 1, the actual sizes of the arrays are checked against this value
`var`	An array of `numcols` elements containing initial values for the decision variables. These initial values are relevant only if the Q matrix is semi-definite or indefinite, in which case they will influence the path that the Solver DLL takes, and the final solution to which it will converge.

Notes:

1. `loadquad()` uses a numerical test to determine whether the `Q` matrix is at least positive semi-definite (for minimization problems) or negative semi-definite (for maximization problems). If the `Q` matrix is indefinite, `loadquad()` will return 5 (the ordinal position of the `qmatval` argument) and, if `PARAM_ARGCK` is 1, it will display an error message dialog indicating this condition. See the section "Solution Properties of Quadratic Problems" in the chapter "Designing Your Application."
2. `objsa()` and `rhssa()` (which return LP range information) may not be used when `loadquad()` has been called.

loadctype

loadctype() is a function that specifies a mixed-integer problem once a problem has been defined with `loadlp` or `loadnlp`.

`loadctype()` returns 0 unless there is an error in its arguments, in which case it returns 1 or 2 to indicate the ordinal position of the argument found to be in error.

Syntax:

```
INTARG loadctype (lp, ctype)
HPROBLEM lp;
LPBYTEARG ctype;
```

Parameters:

Parameter Name	Parameter Description
`lp`	Handle to the current problem as returned by a previous call to `loadlp()` or `loadnlp()`.
`ctype`	An array (of dimension `numcols`) indicating the types of variables in MIP problems. The element `ctype[i]` indicates the type of variable i as follows: `ctype[i] = 'C'` Continuous variable. `ctype[i] = 'I'` General integer variable. `ctype[i] = 'B'` Binary (0-1) integer variable.

loadnlp

loadnlp() is a function that defines a non-linear optimization problem.

Syntax:

```
HPROBLEM loadnlp (probname, numcols, numrows, objsen, obj, rhs,
              sense,matbeg, matcnt, matind, matval, var, lb,
              ub, rngval, nzspace, funceval, jacobian)
LPSTR probname;
INTARG numcols, numrows, objsen;
LPREALARG obj, rhs;
LPBYTEARG sense;
LPINTARG matbeg, matcnt;
HPINTARG matind;
HPREALARG matval;
LPREALARG var, lb, ub, rngval;
INTARG nzspace;
_FUNCEVAL funceval;
_JACOBIAN jacobian;
```

Parameters:

Parameter Name	Parameter Description
`probname`	A character string (currently 16 characters plus a 0 terminator byte) containing a unique "key" which is assigned to you when you license the Solver DLL from Frontline Systems. The DLL that i*s licensed to you recognizes your specific key string; if it is not supplied when* `loadlp()` or `loadnlp()` is called, these functions will return NULL, and calls to these and other functions will not be successful.
`numcols`	The number of columns in the constraint matrix (i.e. the number of variables).
`numrows`	The number of rows in the constraint matrix, not including the objective function or bounds on the variables.
`objsen`	Indicates minimization or maximization: 1 = minimize, -1 = maximize.
`obj`	An array (of dimension `numcols`) containing the objective function gradient. This array must be present and of the appropriate size, but its contents need be initialized only if you are calling `loadnltype()` to specify that some or all of objective gradient elements are constant. Otherwise, the Solver DLL will fill in this array when it calls your `funceval()` and/or `jacobian()` functions.

rhs	An array (of dimension numrows) containing the constant or "right-hand side" term for each row in the constraint matrix. See also the rngval argument below.
sense	An array (of dimension numrows) containing the sense of each constraint (row): sense[i] = 'L' ≤ constraint sense[i] = 'E' = constraint sense[i] = 'G' ≥ constraint sense[i] = 'R' ranged constraint Note that the value 'R' cannot be used with the Small-Scale Solver DLL.
matbeg matcnt matind matval	These four arguments describe the nonzero elements of the Jacobian matrix. If the problem has no constraints, you may pass NULL values for all of these arguments. Otherwise, the matval array, at a minimum, must be present and must be of the appropriate size, but its contents need be initialized only if you are calling loadnltype() to specify that some or all of Jacobian matrix elements are constant. Otherwise, the Solver DLL will fill in this array when it calls your funceval() and/or jacobian() functions. If the matbeg, matcnt, and matind arrays are present, these arrays and the matval array must account for all of the partial derivatives which could become nonzero at any time during the solution process. You may pass NULL values for matbeg, matcnt, and matind if your matval array is a full-size dense matrix, i.e. consisting of numcols * numrows elements (and the elements of each column are consecutive). If matbeg, matcnt and matind are not NULL values, the array matval contains the nonzero coefficients grouped by column (i.e. all coefficients for the same variable are consecutive). matbeg[i] contains the index in matval of the first coefficient for column i. matcnt[i] contains the number of coefficients in column i. Note that the indices in matbeg must be in ascending order. Each entry matind[i] is the row number of the corresponding coefficient in matval[i]. The entries in matind are not required to be in ascending row order. The coefficient M[i,j] in the full constraint matrix (if it is nonzero) would be stored in matval[matbeg[j]+k] where k is between 0 and matcnt[j]-1 (inclusive) and the corresponding entry matind[matbeg[j]+k] = i, the row index.
var	An array (of dimension numcols) containing initial values for the variables. These values may influence the progress of the nonlinear Solver Engine and determine which of several locally optimal points are found during the solution process.
lb	An array (of dimension numcols) containing the lower bound on each variable. A lower bound less than or equal to -INFBOUND will be treated as "minus infinity."
ub	An array (of dimension numcols) containing the upper bound on each variable. A upper bound greater than or equal to +INFBOUND will be treated as "plus infinity."
rngval	An array (of dimension rows) containing the range value of each constraint

	right-hand side. Ranged rows are specified in the `sense` array. If the row is not ranged, the corresponding `rngval` element should be 0.0. A range value for row i means that the constraint left-hand side must be between `rhs[i]` and `rhs[i]` + `rngval[i]`. `rngval[i]` may be positive or negative. If there are no ranged rows at all, `rngval` may be NULL (it is ignored when the Small-Scale Solver DLL is used).
`nzspace`	This argument should be equal to the number of entries in `matval`. When `PARAM_ARRAY` is 1, the actual size of the `matval` SAFEARRAY is checked and must match the value of `nzwspace`.
`funceval`	The address of a user-written callback function which computes the values of your objective function and constraints (i.e. a procedure pointer), declared as shown in the Callback Functions sections.
`jacobian`	The address of a user-written callback function which computes a gradient for your objective function and a Jacobian matrix for your constraints (i.e. a procedure pointer), declared as shown in the Callback Functions sections.

Note:

`Loadnlp()` returns NULL if there is an error in its arguments, if nonlinear problems are not supported by this version of the Solver DLL, if the `probname` key string is not recognized, or in the single-threaded version of the DLL, if another problem is still active (i.e. if `loadlp()` or `loadnlp()` has been called without a corresponding call to `unloadprob()`.). You should not change or free the memory for arguments passed to `loadnlp()` until after you make the corresponding call to `unloadprob()`.

loadnltype

loadnltype() is a function that specifies whether each variable occurs in a linear, nonlinear, or nonsmooth/discontinuous way in the function.

The use of this function is optional. If called, it gives the Solver DLL information about whether each variable occurs in a linear, nonlinear, or nonsmooth/discontinuous way in each problem function. (The relevant variable and constraint are identified by the corresponding elements of the `matbeg`, `matcnt` and `matind` arrays.) The Solver can use this information to choose the appropriate Solver Engine and/or to speed up the solution process.

Syntax:

```
INTARG loadnltype (lp, objtype, matvaltype)
HPROBLEM lp;
LPBYTEARG objtype, matvaltype;
```

Parameters:

Parameter Name	Parameter Description
`lp`	Handle to the current problem as returned by a previous call to `loadnlp()`.
`objtype`	An array (of dimension `numcols`) indicating whether the variables occur in a linear, smooth nonlinear, or nonsmooth/discontinuous way in the objective function: `objtype[i] = 'L'` Variable occurs linearly (or not at all) in the objective. `objtype[i] = 'N'` Variable is (smooth) nonlinear in the objective. `objtype[i] = 'D'` Variable is nonsmooth/discontinuous in the objective.
`matvaltype`	An array (of dimension `nzwspace`) indicating whether the variables occur in a linear, smooth nonlinear, or nonsmooth/discontinuous way in the constraint functions: `matvaltype[i] = 'L'` where `matval[i]` represents a linear occurrence of a variable; `matvaltype[i] = 'N'` where `matval[i]` represents a (smooth) nonlinear occurrence; `matvaltype[i] = 'D'` where `matval[i]` represents a nonsmooth/discontinuous occurrence.

Note:

You may pass NULL values for `objtype`, `matvaltype`, or both to indicate that the objective or the constraints are, in general, nonsmooth or discontinuous functions of the variables. In Version 3.5 of the Solver DLL, if either of these arguments is NULL, or if any variable occurs in a nonsmooth/discontinuous way in the objective or in any constraint, the Evolutionary Solver Engine is used to solve the problem.

testnltype

testnltype() is a function that determines the Problem Type.

The use of this function is optional. It can be called to compute information about whether each variable occurs linearly or nonlinearly in each problem function, through a numerical test.

`testnltype()` computes the gradient for your objective function and Jacobian values for your constraints at the origin, storing them in the `obj` and `matval` arrays that you supplied in your call to `loadnlp()`. It then tests whether these gradient and Jacobian values remain constant over a range around the origin. Your `matval` array must be of appropriate size, and if they are used, your `matbeg`, `matcnt` and `matind` arrays must be correctly initialized to indicate which elements of the Jacobian matrix can be nonzero for any variable values over the domain of your functions. `testnltype()` will ignore elements of the Jacobian that you have indicated are zero (through your `matbeg`, `matcnt` and `matind` arrays); hence it will not diagnose linearity correctly if you have "missed" some Jacobian entries which can be nonzero for certain variable values. (Remember that you can always pass NULL values for `matbeg`, `matcnt` and `matind` and pass a `matval` array of size `numcols` * `numrows`.)

The information used to compute function values in your `funceval()` function may provide a more certain way to determine the linearity or nonlinearity of your model. If you have this information, it is better to make the decision to call `loadnlp()` or `loadlp()` based on your own test, and optionally call `loadnltype()` to supply the information to the Solver DLL.

If `testnltype()` returns 0 in `pstat`, the `matval` array will contain the (constant) Jacobian values which make up the LP coefficient matrix for the Simplex method. You may switch to the Simplex method and solve the problem by calling `unloadprob()`, then calling `loadlp()` with the same arguments (including `matval`) you supplied in your call to `loadnlp()`.

Syntax:

```
INTARG testnltype (lp, numtests, testvals, pstat, objtype,
                   matvaltype)

HPROBLEM lp;

INTARG numtests;

LPREALARG testvals;

LPINTARG pstat;

LPBYTEARG objtype, matvaltype;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to loadnlp().
numtests	Indicates the number of tests performed to validate the gradient and Jacobian values. Each test is performed by choosing test values for each variable, computing the objective and constraint values using the gradient and Jacobian just determined, computing the objective and constraint values by calling your funceval() function, and comparing the results. If you supply a NULL value for testvals, the test values for the variables are chosen randomly. If you supply an array for testvals, the test values are drawn from this array.
testvals	An array (of dimension numtests * numcols) containing test values for the variables to be used in validating the gradient and Jacobian determined by testnltype(), as described above. If you supply a NULL value, the Solver DLL will choose test values at random between the bounds you supply in the lb and ub arrays.
pstat	A pointer to a (long) integer indicating whether the entire problem was diagnosed as nonlinear or linear: pstat = 1 — Problem was diagnosed as (smooth) nonlinear. pstat = 0 — Problem was diagnosed as linear. pstat = -1 — Diagnosis aborted because your funceval() function returned 1 (failure). **Note:** If pstat returns 1, at least one of the elements of either objtype or matvaltype will be 'N'; if pstat returns 0, all elements of both arrays will be 'L'.
objtype	An array (of dimension numcols) which testnltype() will fill in to indicate whether the variables occur linearly or nonlinearly in the objective function: objtype[i] = 'L' — Variable occurs linearly (or not at all) in the objective. objtype[i] = 'N' — Variable occurs nonlinearly in the objective.
matvaltype	An array (of dimension nzwspace) which testnltype() will fill in to indicate whether the variables occur linearly or nonlinearly in the constraint functions: matvaltype[i] = 'L' — matval[i] represents a linear occurrence of a variable. matvaltype[i] = 'N' — matval[i] represents a nonlinear occurrence of a variable. **Note:**

The relevant variable and constraint are identified by the corresponding elements of the `matbeg`, `matcnt` and `matind` arrays. You may pass a NULL value for `objtype` and/or `matvaltype` if you are not interested in this information.

Note:

The numerical test used by `testnltype()` is effective on a broad range of problems, but since it samples function values returned by the `funceval()` function, it is not fool-proof. An all-linear problem which is reasonably well scaled will always return 0 (linear model) in `pstat`; but a very poorly scaled linear problem could return 1 (nonlinear model), and a nonlinear problem which behaves linearly over a significant region around the origin could return 0 (linear model). Given the nature of the numerical test, `testnltype()` cannot determine whether any of the variables occur in a nonsmooth/discontinuous way in your problem.

unloadprob

unloadprob() is a function that unloads ("erases") the current problem from the Solver DLL's memory.

After calling `unloadprob()` you may change or free the memory for the arguments passed to `loadlp()` or `loadnlp()`.

`unloadprob()` returns 0 unless its `lp` argument is invalid, in which case it returns 1.

Syntax:

```
INTARG unloadprob (lp)
HPROBLEM *lp;
```

Parameters:

Parameter Name	Parameter Description
`lp`	`Handle to the current problem as returned by a previous call to loadlp() or loadnlp(). After this call, the value of lp is invalid and should no longer be used.`

Note:

The single-threaded version of the Solver DLL handles only one problem at a time. Hence, you may call `loadlp()` or `loadnlp()` for a second problem only after you have called `unloadprob()` for the first problem. The multi-threaded version of the Solver DLL supports multiple concurrent or recursive calls to these functions.

Solution Functions

Click to navigate to the desired topic:

- optimize
- mipoptimize
- solution
- objsa
- rhssa
- varstat
- constat

optimize

optimize() is a function that solves the specified non-integer optimization problem.

From a programming standpoint, this means that `optimize()` is used to solve problems for which `loadctype()` has not been called. Thus, when this function is called, the Solver DLL will solve the specified linear, quadratic, nonlinear, or nonsmooth optimization problem.

`optimize()` returns a value of 0 if successful (i.e. the solution process ran to completion, though it may or may not have found an optimal solution – see `solution()` for details). If the solution process cannot be carried out, `optimize()` returns a nonzero value.

Syntax:

```
INTARG optimize (lp)
HPROBLEM lp;
```

Parameters:

Parameter Name	Parameter Description
`lp`	Handle to the current problem as returned by a previous call to `loadlp()` or `loadnlp()`.

mipoptimize

mipoptimize() is a function that solves the specified mixed-integer optimization problem.

`mipoptimize()` is used instead of `optimize()` if the problem has one or more integer variables. When this function is called, the Solver DLL will solve the specified optimization problem.

`mipoptimize()` returns a value of 0 if successful (i.e. the solution process ran to completion, though it may or may not have found an optimal solution – see `solution()` for details). If the solution process cannot be carried out, `mipoptimize()` returns a nonzero value.

Syntax:

```
INTARG mipoptimize (lp)
HPROBLEM lp;
```

Parameters:

Parameter Name	Parameter Description
`lp`	Handle to the current problem as returned by a previous call to `loadlp()` or `loadnlp()`.

solution

solution() is a function that retrieves the final status of the optimization, the final value of the decision variables and the objective function, and if appropriate, dual values (basic sensitivity analysis information) for the variables and constraints.

`solution()` returns 0 unless its `lp` argument is invalid, in which case it returns 1.

Syntax:

```
INTARG solution (lp, pstat, pobj, x, piout, slack, dj)
HPROBLEM lp;
LPINTARG pstat;
LPREALARG pobj, x, piout, slack, dj;
```

Parameters:

Parameter Name	Parameter Description
`lp`	Handle to the current problem as returned by a previous call to `loadlp()` or `loadnlp()`.
`pstat`	A pointer to a (long) integer where the result of the optimization will be stored. The specific values which stat (pointed to by `pstat`) can take and their meanings are:

Value	Meaning
`stat = 1 = PSTAT_OPTIMAL`	Optimal solution found.
`stat = 2 = PSTAT_INFEASIBLE`	No feasible solution.
`stat = 3 = PSTAT_UNBOUNDED`	Objective unbounded.
`stat = 5 = PSTAT_IT_LIM_FEAS`	Iteration limit exceeded, feasible.
`stat = 6 = PSTAT_IT_LIM_INFEA`	SIteration limit exceeded, not yet feasible.
`stat = 7 = TIME_LIM_FEAS`	Time limit exceeded, feasible.
`stat = 8 = TIME_LIM_INFEAS`	

Time limit exceeded, not yet feasible.

`stat = 12 = PSTAT_ABORT_FEAS`
User interrupted solution, feasible.

`stat = 13 = PSTAT_ABORT_INFEAS`
User interrupted solution, not yet feasible.

`stat = 65 = PSTAT_FRACT_CHANGE`
Objective function changing too slowly.

`stat = 66 = PSTAT_NO_REMEDIES`
All remedies failed to find a better point.

`stat = 67 = PSTAT_FLOAT_ERROR`
Error in evaluating problem functions.

`stat = 68 = PSTAT_MEM_LIM`
Could not allocate enough memory.

`stat = 69 = PSTAT_ENTRY_ERROR`
Attempt to re-enter DLL during solution.

`stat = 101 = PSTAT_MIP_OPTIMAL`
MIP Optimal solution found.

`stat = 102 = PSTAT_MIP_OPTIMAL_TOL`
MIP solution within `EPGAP` tolerance.

`stat = 103 = PSTAT_MIP_INFEASIBLE`
No feasible integer solution.

`stat = 104 = PSTAT_MIP_SOL_LIM`
Integer solutionlimit exceeded.

`stat = 105 = PSTAT_MIP_NODE_LIM_FEAS`
Node limit exceeded, feasible.

`stat = 106 = PSTAT_MIP_NODE_LIM_INFEAS`
Node limit exceeded, not feasible.

`stat = 107 = PSTAT_MIP_TIME_LIM_FEAS`
Time limit exceeded, feasible.

`stat = 108 = PSTAT_MIP_TIME_LIM_INFEAS`
Time limit exceeded, not feasible.

Note:

A `pstat` value of 69 (`PSTAT_ENTRY_ERROR`) is returned only by the single-threaded version of the Solver DLL, and only if `optimize()` or `mipoptimize()` is called, directly or indirectly, from a callback function that the Solver DLL called during execution of an earlier `optimize()` or `mipoptimize()` call. In the multi-threaded version of the Solver DLL, these types of recursive calls are permitted.

`pobj` A pointer to a double variable where the optimal objective function value will be

	stored.
`x`	A array of dimension equal to the number of columns (variables) in which the optimal values of the (primal) variables will be stored.
`piout`	An array of dimension equal to the number of rows (constraints) in which the dual values (shadow prices) of the constraints will be stored. The `piout` values are available only if an optimal solution to an LP or QP problem has been found.
`slack`	An array of dimension equal to the number of rows (constraints) in which the optimal slack or surplus values for the constraints will be stored. The `slack` values are available only if an optimal solution to an LP or QP problem has been found.
`dj`	An array of dimension equal to the number of columns (variables) in which the dual values (reduced costs) of the variables will be stored. The `dj` values are available only if an optimal solution to an LP or QP problem has been found.

Notes:

1. All arguments except `lp` are pointers to locations where data may be written. This data may include the status of the optimization, the value of the objective function, the values of the (primal) variables, the dual values, the slacks and the reduced costs.
2. If some of the data is not required, a NULL value may be passed for that argument.
3. If there is no optimal solution, sensitivity analysis information is not available, so the `piout`, `slack` and `dj` arguments will be ignored.
4. If you are solving a MIP problem, sensitivity analysis information is not meaningful and the `piout`, `slack` and `dj` arguments must be NULL values.

objsa

objsa() is a function that, for linear programming problems, returns the ranges of values over which the sensitivity analysis information is valid.

Since "range information" is not meaningful for NSP, NLP, QP and MIP problems, `objsa()` should not be called for them.

`objsa()` returns 0 unless there is an error in its arguments, in which case it returns an integer value indicating the ordinal position of the first argument found to be in error.

Syntax:

```
INTARG objsa (lp, begidx, endidx, lower, upper)
HPROBLEM lp;
INTARG begidx, endidx;
LPREALARG lower, upper;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadlp()`.
begidx	The first column number (index as in `loadlp()` arguments) for which sensitivity range information should be returned. Used to limit the set of columns (variables) for which the range information is computed.
endidx	The last column number (index as in `loadlp()` arguments) for which sensitivity range information should be returned. Used to limit the set of columns (variables) for which the range information is computed.
lower	An array of dimension `endidx` - `begidx` + 1 where the objective function coefficient lower range values will be returned.
upper	An array of dimension `endidx` - `begidx` + 1 where the objective function coefficient upper range values will be returned.

rhssa

rhssa() is a function that, for linear programming problems, returns the ranges of values over which the sensitivity analysis information is valid.

This function need be called only if sensitivity ranges for the constraint right-hand sides are desired. Since range information is not meaningful for NSP, NLP, QP and MIP problems, `rhssa()` should not be called for them.

`rhssa()` returns 0 unless there is an error in its arguments, in which case it returns an integer value indicating the ordinal position of the first argument found to be in error.

Syntax:

```
INTARG rhssa (lp, begidx, endidx, lower, upper)
HPROBLEM lp;
INTARG begidx, endidx;
LPREALARG lower, upper;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadlp()`.
begidx	The first row number (index as in `loadlp()` arguments) for which sensitivity range information should be returned. Used to limit the set of rows (constraints) for which the range information is computed.
endidx	The last row number (index as in `loadlp()` arguments) for which sensitivity range information should be returned. Used to limit the set of rows (constraints) for which the range information is computed.
lower	An array of dimension `endidx` - `begidx` + 1 where the constraint right-hand side lower range values will be returned.
upper	An array of dimension `endidx` - `begidx` + 1 where the constraint right-hand side upper range values will be returned.

varstat

varstat() is a function that, for nonsmooth optimization problems, returns statistical information about the variable values in the final population of candidate solutions.

This function need be called only if statistical information about the variable values in the final population of candidate solutions (found by the Evolutionary Solver Engine) is desired. Since this information is meaningful only for NSP (nonsmooth optimization) problems, `varstat()` should not be called for other types of problems.

`varstat()` returns 0 unless there is an error in its arguments, in which case it returns an integer value indicating the ordinal position of the first argument found to be in error.

Syntax:

```
INTARG varstat (lp, begidx, endidx, mid, disp, lower, upper)
HPROBLEM lp;
INTARG begidx, endidx;
LPREALARG mid, disp, lower, upper;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadnlp()`.
begidx	The first row number (index as in `loadnlp()` arguments) for which statistical information should be returned. Used to limit the set of rows (constraints) for which the statistical information is computed.
endidx	The last row number (index as in `loadnlp()` arguments) for which statistical information should be returned. Used to limit the set of rows (constraints) for which the statistical information is computed.
mid	An array of dimension `endidx - begidx + 1` where the mean of each constraint's values in the final population of candidate solutions will be returned.
disp	An array of dimension `endidx - begidx + 1` where the standard deviation of each constraint's values in the final population of candidate solutions will be returned.
lower	An array of dimension `endidx - begidx + 1` where the minimum of each constraint's values in the final population of candidate solutions will be returned.
upper	An array of dimension `endidx - begidx + 1` where the maximum of each

constraint's values in the final population of candidate solutions will be returned.

constat

constat() is a function that, for nonsmooth optimization problems, returns statistical information about the constraint values in final population of candidate solutions.

This function need be called only if statistical information about the constraint values in the final population of candidate solutions (found by the Evolutionary Solver Engine) is desired. Since this information is meaningful only for NSP (nonsmooth optimization) problems, `constat()` should not be called for other types of problems.

`constat()` returns 0 unless there is an error in its arguments, in which case it returns an integer value indicating the ordinal position of the first argument found to be in error.

Syntax:

```
INTARG constat (lp, begidx, endidx, mid, disp,
   lower, upper)HPROBLEM lp;
HPROBLEM lp;
INTARG begidx, endidx;
LPREALARG mid, disp, lower, upper;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadnlp()`.
begidx	The first column number (index as in `loadnlp()` arguments) for which statistical information should be returned. Used to limit the set of columns (variables) for which the statistical information is computed.
endidx	The last column number (index as in `loadnlp()` arguments) for which statistical information should be returned. Used to limit the set of columns (variables) for which the statistical information is computed.
mid	An array of dimension `endidx - begidx + 1` where the mean of each variable's values in the final population of candidate solutions will be returned.
disp	An array of dimension `endidx - begidx + 1` where the standard deviation of each variable's values in the final population of candidate solutions will be returned.
lower	An array of dimension `endidx - begidx + 1` where the minimum of each variable's values in the final population of candidate solutions will be returned.

upper An array of dimension `endidx - begidx + 1` where the maximum of each variable's values in the final population of candidate solutions will be returned.

Diagnostic Functions

Click to navigate to the desired topic:

- findiis
- getiis
- iiswrite
- lpwrite
- lpread

findiis

findiis() is a function that computes an Irreducibly Infeasible Subset (IIS) for a linear or quadratic problem.

This function helps you identify problems in your constraint formulation which make the problem infeasible. You may call `findiis()` when the `pstat` argument value returned by `solution()` is 2 (`PSTAT_INFEASIBLE`).

If the parameter `PARAM_IISBND` is 0 (the default), both constraints and variable bounds will be considered and eliminated from the IIS if possible. If `PARAM_IISBND` is 1, only the constraints will be considered for elimination from the IIS, leaving all variable bounds in force.

Syntax:

```
INTARG findiis (lp, pnumrows, pnumcols)
HPROBLEM lp;
LPINTARG pnumrows, pnumcols;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadlp()` or `loadnlp()`.
pnumrows	On return, the variable pointed to by `pnumrows` will contain the number of constraints (rows) in the IIS.
pnumcols	On return, the variable pointed to by `pnumcols` will contain the number of variable bounds (columns) in the IIS.

Note:

To obtain details of the IIS, call the function `getiis()`. The function `iiswrite()` calls `findiis()` automatically, if it has not already been called.

getiis

getiis() is a function that returns information about the specific constraints (rows) and variable bounds (columns) included in the IIS. It is used to examine the IIS programmatically.

You may call this function after calling `findiis()`, which returns the information you need to properly dimension the arrays `rowind`, `rowbdstat`, `colind` and `colbdstat` passed to `getiis()`. `getiis()` returns information about the specific constraints (rows) and variable bounds (columns) included in the IIS.

Syntax:

```
INTARG getiis (lp, pstat, rowind, rowbdstat, pnumrows, colind,
               colbdstat, pnumcols)
HPROBLEM lp;
LPINTARG pstat;
LPINTARG rowind, rowbdstat, pnumrows;
LPINTARG colind, colbdstat, pnumcols;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadlp()` or `loadnlp()`.
rowind **rowbdstat** **pnumrows**	The arguments `rowind` and `rowbdstat` should be arrays of integers with dimension at least equal to the `pnumrows` argument value returned by `findiis()`. On return, `rowind` contains the indices (same as those used in the `sense` and `rhs` arguments of `loadlp()` and `loadnlp()`) of the constraints (rows) included in the IIS. The corresponding elements of `rowbdstat` will contain 0 if the constraint (normally with `sense` 'G') is at its lower bound; 1 if the constraint (normally with `sense` 'E') is fixed at one value; and 2 if the constraint (normally with `sense` 'L') is at its upper bound.
colind **colbdstat** **pnumcols**	The arguments `colind` and `colbdstat` should be arrays of integers with dimension at least equal to the `pnumcols` argument value returned by `findiis()`. On return, `colind` contains the indices (same as those used in the `lb` and `ub` arguments of `loadlp()` and `loadnlp()`) of the variable bounds (columns) included in the IIS. The corresponding elements of `colbdstat` will contain 0 if the variable is at its lower bound; 1 if the variable is fixed at one value; and 2 if the variable is at its upper bound.

Note:

If the value returned via `pstat` is 2 – meaning that a complete IIS was not found – the information returned by `getiis()` may still be useful: It will identify a subset of the constraints and variable bounds containing the source of infeasibility, even though this subset may contain more rows and columns than would be included in an IIS.

iiswrite

iiswrite() is a function that writes a linear or quadratic problem together with its Irreducibly Infeasible Subset (IIS) to a text file.

You may call this function (for linear or quadratic programming problems only) when the `pstat` argument value returned by `solution()` is 2 (`PSTAT_INFEASIBLE`).

The `iiswrite()` function calls `findiis()`, if necessary, to compute an Irreducibly Infeasible Set of constraints and bounds, as described above. It then writes an ASCII text file to disk, named filename, which lists the objective, the constraints included in the IIS, and the variable bounds included in the IIS, in the same "algebraic" format used by `lpwrite()`. This file is designed to be quickly readable in a text editor, and you can use it to identify the problem(s) in your constraints which are causing the overall model to be infeasible.

Syntax:

```
INTARG iiswrite (lp, filename)
HPROBLEM lp;
LPSTR filename;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadlp()`.
filename	Name (or complete path) of the ASCII text file to be saved to disk.

lpwrite

lpwrite() is a function that writes a linear or quadratic problem to a text file.

This function may be called at any time after `loadlp()`, `loadquad()` and/or `loadctype()` have been called (and before `unloadprob()` is called). It cannot be called for a nonlinear problem, defined via `loadnlp()`.

`lpwrite()` will write an output text file filename, containing an algebraic statement of the problem you have defined. By calling `lpwrite()` and examining the resulting file, you can verify that the problem defined by your arguments is the one you intended. You can also use `lpwrite()` to save a problem to disk, and `lpread()` to read it in and solve it later.

`lpwrite()` returns 0 unless there is an error in its arguments, in which case it returns 1 for an invalid `lp` argument, or 2 for any I/O error related to opening and writing the contents of filename.

As an example, the following text appears in the output file when `lpwrite()` is called for Example 2 in the sample source programs supplied with the Solver DLL:

```
Maximize LP/MIP
   obj: 2.0 x1 + 3.0 x2
Subject To
   c1:  9.0 x1 + 6.0 x2 ≤ 54.0
   c2:  6.0 x1 + 7.0 x2 ≤ 42.0
   c3:  5.0 x1 + 10.0 x2 ≤ 50.0
Bounds
   0.0 ≤ x1 ≤ +infinity
   0.0 ≤ x2 ≤ +infinity
Integers
   x1
   x2
End
```

Syntax:

```
INTARG loadctype (lp, ctype)
LPSTR filename;
```

Parameters:

Parameter Name	Parameter Description
lp	**Handle to the current problem as returned by a previous call to `loadlp()`.**
filename	**Name (or complete path) of the output text file. `lpwrite()` will create this file, which will replace any existing file of the same name. If filename does not include an explicit path, the file will be created in the current directory.**

Note:

For compatibility with previous versions of the Solver DLL, `lprewrite()` is a synonym for `lpwrite()`.

lpread

lpread() is a function that creates a linear, quadratic or mixed-integer problem from a text file.

This function can be used to read in the definition of a linear, quadratic, or mixed-integer problem, in algebraic format, from a text file. The file may be previously written by the `lpwrite()` function created either by hand or by another program. As long as the problem definition follows the format and syntax described here (and illustrated above under `lpwrite()`) and is within the size limits of your version of the Solver DLL, you can read it in through `lpread()` and solve it.

The text file should consist of lines ending in a new line (linefeed) or a carriage return/linefeed combination. The keywords `Maximize` or `Minimize`, `Subject To`, `Bounds`, `Integers` and `End` should appear on separate lines. The objective expression must be preceded by `obj`: and each constraint expression preceded by `c1:`, `c2:`, etc., but otherwise the expressions may be split across multiple lines. Each variable occurrence must have a form such as x1, x123, etc.; the highest variable index found in the file is returned in the variable pointed to by `numcols_p`. Lower and upper bounds on variables should either be numbers, -infinity or +infinity. The quadratic part of the objective (if used) must be surrounded by brackets, and each term should consist of a number followed by either two variable names separated by *, or one variable name followed by ^2. For example:

```
[ x1^2 - x1*x2 + x2^2 ]
```

Syntax:

```
INTARG lpread (lp, filename, objsen_p, numcols_p, numrows_p,
               numints_p, matcnt, qmatcnt)
HPROBLEM lp;
LPSTR filename;
LPINTARG objsen_p, numcols_p, numrows_p, numints_p;
LPINTARG matcnt, qmatcnt;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadlp()`. (May be NULL value.) If this argument is NULL, `lpread()` will scan the input text file and return values for the `objsen_p`, `numcols_p`, `numrows_p`, `numints_p`, `matcnt` and `qmatcnt` arguments (if they are specified), but it will not store any of

the coefficient, bound or integer variable information. If `lp` is non-NULL, `lpread()` will scan the file, return values for the other arguments mentioned above, and fill in values for all of the arguments previously passed to `loadlp()`, `loadquad()` and/or `loadctype()` when the problem was defined, except for `matbeg` and `matcnt` (if used) and `qmatbeg` and `qmatcnt` (if used). It is your responsibility to ensure that the arrays previously passed to these functions are of the correct dimensions for the problem being read.

filename	Name (or complete path) of the input text file.
objsen_p	On return, the variable pointed to by `objsen_p` will contain the "sense" of the optimization problem (1 for minimize, -1 for maximize), as used by `loadlp()`. You may pass a NULL value for this argument if you don't need this information.
numcols_p **numrows_p** **numints_p**	On return, the variable pointed to by `numcols_p` will contain the number of columns (variables) in the problem defined by the input file; the variable pointed to by `numrows_p` will contain the number of rows (constraints) in the problem; and the variable pointed to by `numints_p` will contain the number of integer variables in the problem. You may pass NULL values for any of these arguments if you don't need the corresponding information.
matcnt	If used, this argument must be an integer array of dimension at least equal to the number of columns (variables) in the problem. On return, each element of the array will contain the number of nonzero coefficients of the corresponding variable found in the constraints of the problem. The re*sulting array has the same meaning as, and may be passed as, the* `matcnt` argument of the `loadlp()` function. You may pass a NULL value for this argument if you don't need this information, for example if you are using a dense matrix for the `matval` array of coefficients passed to `loadlp()`.
qmatcnt	If used, this argument must be an integer array of dimension at least equal to the number of columns (variables) in the problem. On return, each element of the array will contain the number of nonzero coefficients of the corresponding variable found in the quadratic objective (if any) of the problem. The resulting array has the same meaning as, and may be passed as, the `qmatcnt` argument of the `loadquad()` function. You may pass a NULL value for this argument if you don't need this information, for example if you are using a dense matrix for the `qmatval` array of coefficients passed to `loadquad()`.

Notes:

1. If you know the sense and dimensions (number of variables, constraints and integers) of the problem you are going to read in and solve, you can simply call `loadlp()` (plus `loadquad()` and/or `loadctype()`) to define the problem with arrays of the correct dimension, then call `lpread()` once to fill in the coefficient, bound and integer variable information.
   If you do not know the sense and/or dimensions of the problem in advance, you can call `lpread()` with a NULL first argument to obtain this information; then allocate arrays of the appropriate size, and call `loadlp()` (plus `loadquad()` and/or `loadctype()`) to define the problem; and finally call `lpread()` again – this time

with a non-NULL first argument – to fill in the coefficient, bound and integer variable information.

2. To satisfy the Solver DLL's checks for argument validity, you must initialize the `sense` array – say, with all elements equal to '`E`', the `ctype` array (if used for integer problems) – say, with all elements equal to '`C`', and the `matbeg`, `matcnt` and `matind` arrays (if used for sparse problems). `matbeg` and `matcnt` may be initialized as described below, and `matind` may be initialized with all elements equal to zero. (`qmatbeg`, `qmatcnt` and `qmatind` should be treated similarly.)
   If you are using sparse arrays to define `matval` or `qmatval`, and you do not know the dimensions or sparsity pattern in advance, you will need to call `lpread()` twice as outlined above, passing the `matcnt` and/or `qmatcnt` array arguments on the first call. These arrays must be of dimension at least equal to the number of columns (variables) in the problem being read; to create them, you can either use the maximum number of columns returned by the `getproblimits()` function, or you can call `lpread()` one more time in advance to obtain the actual number of columns defined in the text file. The number of non-zeroes in the constraint matrix (the `nzwspace` argument passed to `loadlp()`) is equal to the sum of the counts returned in the elements of `matcnt` (similarly for `qmatcnt` and `qnzspace`). You can initialize the `matbeg` array based on the counts in `matcnt` (similarly for `qmatbeg` and `qmatcnt`). For example, in C/C++:

```
   for (nzspace = i = 0; i < numcols; i++) nzspace +=
matcnt[i];
   for (i = 0; i < nzspace; i++) matind[i] = 0;
   for (i = 0; i < numcols; i++)
        matbeg[i] = (i == 0 ? 0 : matbeg[i-1] + matcnt[i-
1]);
```

Use Control Functions

Click to navigate to the desired topic:

- getuse
- reportuse
- getproblimits

getuse

getuse() is a function that returns information about the number of uses since the Solver DLL was placed in service. The number of uses is determined by the number calls made to either `loadlp()` or `loadnlp()`, or to `optimize()` or `mipoptimize()`.

You may pass NULL for any of the other arguments, if you do not need the corresponding information.

Syntax:

```
INTARG getuse (lp, loadprob_p, optimize_p, verify_p, reload_p,
               repopt_p, repdate_p)
HPROBLEM lp;
LPINTARG loadprob_p, optimize_p, verify_p;
LPINTARG reload_p, repopt_p, repdate_p;
```

Parameters:

Parameter Name	Parameter Description
lp	This argument is ignored and may be NULL.
loadprob_p	A pointer to a variable of type long, where `getuse()` will store the cumulative number of calls to `loadlp()` or `loadnlp()` to date. (May be NULL value.)
optimize_p	A pointer to a variable of type long, where `getuse()` will store the cumulative number of calls to `optimize()` or `mipoptimize()` to date. (May be NULL value.).
verify_p	A pointer to a variable of type long, where `getuse()` will store an encrypted form of the cumulative number of uses to date. (May be NULL value.)
reload_p	A pointer to a variable of type long, where `getuse()` will store the number of calls to `loadlp()` or `loadnlp()` previously reported to Frontline Systems (via an earlier call to `reportuse()`, or automatically as determined by the `PARAM_USERP` setting). (May be NULL value.)
repopt_p	A pointer to a variable of type long, where `getuse()` will store the number of calls to `optimize()`/`mipoptimize()` previously reported to Frontline Systems (via an earlier call to `reportuse()`, or automatically as determined by the `PARAM_USERP` setting). (May be NULL value.)
repdate_p	A pointer to a variable of type long, where `getuse()` will store the date of the last report to Frontline Systems, as an integer of the form YYYYMMDD. (May be NULL value.)

Note:

Frontline's use-based licensing terms can take into account the number of calls to `loadlp()` or `loadnlp()`, the number of calls to `optimize()` or `mipoptimize()`, or both counts. In many Solver DLL applications these counts may be the same. However, in some applications, one count or the other may correspond more closely to the number of "sessions" with individual users served by the application.

reportuse

reportuse() is a function that Retrieves Solver DLL license and usage information and either writes it in a text file to disk or sends an automated email to Frontline Systems (info@frontsys.com).

The information retrieved consists of use information returned by `getuse()`, plus your application-specific "key" string and the Windows name of the computer on which the application is running. The email message, if used, is sent via the Win32 Messaging API (MAPI) which is normally available on the Windows system where the Solver DLL is running.

Syntax:

```
INTARG reportuse (lp, probname, filename, profilename, password)
HPROBLEM lp;
LPSTR probname, filename, profilename, password;
```

Parameters:

Parameter Name	Parameter Description
lp	This argument is ignored and may be NULL.
probname	A character string (currently 16 characters plus a 0 terminator byte) containing a unique "key" which is assigned to you when you license the Solver DLL from Frontline Systems – the same as the argument you supply to `loadlp()` or `loadnlp()`.
filename	If used, the name (or complete path) of the output text file that will contain the use information. `reportuse()` will create this file, which will replace any existing file of the same name. If filename does not include an explicit path, the file will be created in the current directory. If this argument is NULL or an empty string, `reportuse()` will send an email message instead.
profilename	This argument is optional. If `reportuse()` is sending an email message (i.e. the filename argument is NULL), this is the profile name (typically, the user name) of the email account used to send the message. If this argument is NULL or an empty string, `reportuse()` uses the MAPI "shared session" to send the email message
password	This argument is optional. If `reportuse()` is sending an email message (i.e. the filename argument is NULL), this is the password of the email account used to send the message. If this argument is NULL or an empty string, `reportuse()` uses the MAPI "shared session" to send the email message.

getproblimits

getproblimits() is a function that returns the optimization problem limits (available Engines, max. number of variables and constraints) with regards to the license purchased.

`getproblimits()` allows you to determine the capabilities of the Solver DLL version you are using, at runtime. The Solver DLL V3.5 can be configured to include or exclude the nonlinear GRG and Evolutionary Solver Engines, the linear Solver Engine, or the quadratic Solver Engine, and the maximum number of decision variables (columns) and constraints (rows) supported by each Engine can also be configured.

Syntax:

```
INTARG getproblimits (lp, type, pnumcols, pnumrows, pnumints)
HPROBLEM lp;
INTARG type;
LPINTARG pnumcols, pnumrows, pnumints;
```

Parameters:

Parameter Name	Parameter Description
`lp`	This argument is ignored and may be NULL.

`pnumcols`
`pnumrows`
`pnumints`
	These three arguments should be pointers to variables of type long, where `getproblimits()` will store the maximum number of decision variables (columns), constraints (rows), and integer variables supported by the Solver Engine specified by the type argument. If the number returned is zero, the corresponding Engine is not supported in this configuration of the Solver DLL.

Callback Functions for Nonlinear Problems

Click to navigate to the desired topic:

- funceval
- jacobian

funceval

funceval() is a function that computes values for the problem functions (objective and constraints) for any given values of the variables.

`funceval()` is passed a vector of variable values, and a vector of function values (including the objective) to be computed. It uses the variables to compute the functions, stores these into the vector of function values, and returns an integer indicating success or failure. The function computes and stores the objective value, computes constraint left-hand side values and stores them in `lhs`, and return 0 for success or 1 for failure (1 will cause the Solver DLL to stop and return from its `optimize()` call).

When the Solver is computing partial derivatives on its own via finite differencing (in the absence of the `jacobian()` callback function), it calls `funceval()` repeatedly, perturbing one variable at a time (and resetting the previously perturbed variable). On these calls (usually about half of all `funceval()` calls), `varone` and `vartwo` are the indices (0 to `numcols`-1) of the (only) variables whose values have changed since the last call. Your `funceval()` function must still compute values for `objval` and all elements of the `lhs` array, but you may be able to do less work based on knowledge of which variables have changed.

Syntax:

```
INTARG funceval (lp, numcols, numrows, objval, lhs, var, varone,
                 vartwo)
HPROBLEM lp;
INTARG numcols, numrows;
LPREALARG objval, lhs, var;
INTARG varone, vartwo;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadnlp()`.
numcols	The number of columns in the constraint matrix (i.e. the number of variables) supplied to `loadnlp()`.
numrows	The number of rows in the constraint matrix, not including the objective function or bounds on the variables as supplied to `loadnlp()`.
objval	A pointer to a location where the (scalar) objective value is stored.

lhs	An array with dimension `numrows`.
var	An array with dimension `numcols`.
varone **vartwo**	May be ignored unless your function evaluation process is designed to take advantage of information about which elements of the `var` array have changed since the last call to `funceval()`. When the Solver is supplying entirely new variable values (i.e. exploring a new trial point), `varone` and `vartwo` will both be –1.

jacobian

jacobian() is a function that computes the objective value, the objective gradient, and the Jacobian matrix of partial derivatives of the problem functions. The computation is performed at a trial point represented by the values in the `var` argument.

The `jacobian()` function, if supplied, is called in preference to `funceval()` – but only by the nonlinear GRG Solver Engine – to compute the objective gradient and the Jacobian matrix of partial derivatives of the problem functions, at a specific trial point found by the Solver. (Each row of the Jacobian is the gradient of the corresponding constraint function.) Note that if `jacobian()` is supplied, `funceval()` will still be called, but its `varone` and `vartwo` arguments will always be -1

Syntax:

```
INTARG jacobian (lp, numcols, numrows, nzspace, objval, obj,
              matbeg, matcnt, matind, matval, var, objtype,
              matvaltype)

HPROBLEM lp;

INTARG numcols, numrows, nzspace;

LPREALARG objval, obj;

LPINTARG matbeg, matcnt;

HPINTARG matind;

HPREALARG matval;

LPREALARG var;

LPBYTEARG objtype, matvaltype;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem as returned by a previous call to `loadnlp()`.
numcols	The number of columns in the constraint matrix (i.e. the number of variables) supplied to `loadnlp()`. (May be NULL value.)
numrows	The number of rows in the constraint matrix, not including the objective function or bounds on the variables as supplied to `loadnlp()`. (May be NULL value.)
nzspace	The same values you supplied to `loadnlp()`. (May be NULL values.)
objval	A pointer to a location where the (scalar) objective value is stored.

obj	An array (of dimension `numcols`) containing the objective function gradient as supplied to `loadnlp()`.
matbeg **matcnt** **matind**	Same values as supplied to `loadnlp()`. (May be NULL values.)
matval	The array in which the partial derivative values are stored. (Be sure not to exceed the dimension `nzwspace` for `matind` and `matval`)
var	The trial point at which the computation is carried out.
objtype **matvaltype**	The arrays of characters supplied to `loadnltype()`. (May be NULL values.) If all elements of the `objtype` array are 'L', you can skip computing and storing the objective value and gradient; if all elements of the `matvaltype` array are 'L', you can skip computing the partial derivatives of the problem functions.

Other Callback Functions

Click to navigate to the desired topic:

- setlpcallbackfunc
- getlpcallbackfunc
- setmipcallbackfunc
- getmipcallbackfunc
- getcallbackinfo

setlpcallbackfunc

setlpcallbackfunc() is a function that specifies the callback function which the Solver DLL should call on each "pivot" or Simplex iteration. It must be called prior to `optimize()` or `mipoptimize()`.

Syntax:

```
INTARG setlpcallbackfunc (lp, callback)
HPROBLEM lp;
_CCPROC callback;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current LP or MIP problem. This is the value returned by a previous call to `loadlp()`.
callback	The address of a user-written callback function (i.e. a procedure pointer), declared as shown above. To eliminate use of the callback function, pass an argument value of NULL to `setlpcallbackfunc()`.

getlpcallbackfunc

getlpcallbackfunc() is a function that retrieves the callback function which the Solver DLL calls on each "pivot" or Simplex iteration.

Syntax:

```
void getlpcallbackfunc (lp, callback_p)
HPROBLEM lp;
_CCPROC *callback_p;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current LP or MIP problem. This is the value returned by a previous call to `loadlp()`.
callback_p	The address of a procedure pointer variable which will be set to the address of the current LP pivot callback function. Before you set the callback with `setlpcallbackfunc()`, `getlpcallbackfunc()` will store NULL in this variable.

setmipcallbackfunc

setmipcallbackfunc() is a function that specifies the callback function which the Solver DLL should call on each "branch" or LP subproblem in a MIP problem.

Syntax:

```
INTARG setmipcallbackfunc (lp, callback)
HPROBLEM lp;
_CCPROC callback;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current LP or MIP problem. This is the value returned by a previous call to `loadlp()`.
callback	The address of a user-written callback function (i.e. a procedure pointer), declared as shown above. To eliminate use of the callback function, pass an argument value of NULL to `setmipcallbackfunc()`.

getmipcallbackfunc

getmipcallbackfunc() is a function that retrieves the callback function which the Solver DLL calls on each "branch" or LP subproblem in a MIP problem.

Syntax:

```
void getmipcallbackfunc (lp, callback_p)
HPROBLEM lp;
_CCPROC *callback_p;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current LP or MIP problem. This is the value returned by a previous call to `loadlp()`.
callback_p	The address of a procedure pointer variable which will be set to the address of the current MIP callback function. Before you set the callback with `setmipcallbackfunc()`, `getmipcallbackfunc()` will store NULL in this variable.

getcallbackinfo

getcallbackinfo() is a function that returns information about the callback function.

Syntax:

```
INTARG getcallbackinfo (lpinfo, wherefrom, infonumber, result_p)
HPROBLEM lpinfo;
INTARG wherefrom, infonumber;
void *result_p;
```

Parameters:

<table>
<tr><th>Parameter Name</th><th>Parameter Description</th></tr>
<tr><td>lpinfo</td><td>Handle to the current LP or MIP problem. This is the value passed to your callback function.</td></tr>
<tr><td>wherefrom</td><td>This argument identifies the callback type; it must be the value of the argument passed to your callback function.</td></tr>
<tr><td>infonumber</td><td>The options below select the specific information to be returned in the result_p argument:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>CBINFO_PRIMAL_OBJ</td><td>The current objective function value (double).</td></tr>
<tr><td>CBINFO_PRIMAL_INFMEAS</td><td>The sum of the current infeasibilities (double).</td></tr>
<tr><td>CBINFO_PRIMAL_FEAS</td><td>= 1 if current solution is feasible, = 0 if not (long).</td></tr>
<tr><td>CBINFO_ITCOUNT</td><td>Number of iterations so far in the LP problem (long).</td></tr>
<tr><td>CBINFO_NODE_COUNT</td><td>Number of branches so far in the MIP problem (long).</td></tr>
<tr><td>CBINFO_MIP_ITERATIONS</td><td>Total number of LP pivots/iterations so far. (long).</td></tr>
<tr><td>CBINFO_BEST_INTEGER</td><td>Objective value of the “incumbent,” the best integer solution found so far (double).</td></tr>
</table></td></tr>
</table>

`result_p`	The address of a variable to receive the requested information; this variable must be of the data type implied by the infonumber argument, e.g. a pointer to a double for a call with `infonumber = CBINFO_PRIMAL_OBJ`, or a pointer to a long integer for `infonumber = CBINFO_PRIMAL_ITCOUNT`.

Parameter Management Functions

Click to navigate to the desired topic:

- setintparam
- getintparam
- infointparam
- setdblparam
- getdblparam
- infodblparam
- setdefaults

setintparam

setintparam() is a function that sets the value of a specified integer parameter in a problem.

`setintparam()` returns 0 if successful, 1 if `whichparam` isn't one of the valid codes for integer parameters, or 2 if `newvalue` is outside the range of valid values for the chosen parameter.

Syntax:

```
INTARG setintparam (lp, whichparam, newvalue)
HPROBLEM lp;
INTARG whichparam;
INTARG newvalue;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem.
whichparam	One of the symbolic names or codes for integer parameters: `PARAM_ARGCK`, `PARAM_ARRAY`, `PARAM_USERP`, `PARAM_IISBND`, `PARAM_ITLIM`, `PARAM_NDLIM`, `PARAM_MIPLIM`, `PARAM_SAMPSZ`, `PARAM_NOIMP`, `PARAM_SCAIND`, `PARAM_CRAIND`, `PARAM_LINVAR`, `PARAM_DERIV`, `PARAM_ESTIM`, `PARAM_DIREC`, `PARAM_RELAX`, `PARAM_PREPRO`, `PARAM_OPTFIX`, `PARAM_REORDR`, `PARAM_IMPBND`, `PARAM_REQBND`.
newvalue	The desired new value for the chosen parameter.

getintparam

getintparam() is a function that retrieves the current value of a specified integer parameter in a problem.

`getintparam()` returns 0 if successful, or 1 if `whichparam` isn't one of the valid codes for integer parameters.

Syntax:

```
INTARG getintparam (lp, whichparam, value_p)
HPROBLEM lp;
INTARG whichparam;
LPINTARG value_p;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem.
whichparam	One of the symbolic names or codes for integer parameters: `PARAM_ARGCK`, `PARAM_ARRAY`, `PARAM_USERP`, `PARAM_IISBND`, `PARAM_ITLIM`, `PARAM_NDLIM`, `PARAM_MIPLIM`, `PARAM_SAMPSZ`, `PARAM_NOIMP`, `PARAM_SCAIND`, `PARAM_CRAIND`, `PARAM_LINVAR`, `PARAM_DERIV`, `PARAM_ESTIM`, `PARAM_DIREC`, `PARAM_RELAX`, `PARAM_PREPRO`, `PARAM_OPTFIX`, `PARAM_REORDR`, `PARAM_IMPBND`, `PARAM_REQBND`.
value_p	A pointer to the location where the current value of the parameter will be stored.

infointparam

infointparam() is a function that retrieves the minimum, maximum, and default values of a specified integer parameter in a problem.

`infointparam()` returns 0 if successful, or 1 if `whichparam` isn't one of the valid codes for integer parameters.

Syntax:

```
INTARG infointparam (lp, whichparam, defvalue_p, minvalue_p,
                maxvalue_p)
HPROBLEM lp;
INTARG whichparam;
LPINTARG defvalue_p, minvalue_p, maxvalue_p;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem. This argument is ignored and may be NULL.
whichparam	One of the symbolic names or codes for integer parameters: `PARAM_ARGCK`, `PARAM_ARRAY`, `PARAM_USERP`, `PARAM_IISBND`, `PARAM_ITLIM`, `PARAM_NDLIM`, `PARAM_MIPLIM`, `PARAM_SAMPSZ`, `PARAM_NOIMP`, `PARAM_SCAIND`, `PARAM_CRAIND`, `PARAM_LINVAR`, `PARAM_DERIV`, `PARAM_ESTIM`, `PARAM_DIREC`, `PARAM_RELAX`, `PARAM_PREPRO`, `PARAM_OPTFIX`, `PARAM_REORDR`, `PARAM_IMPBND`, `PARAM_REQBND`.
defvalue_p	A pointer to the location where the default value of the parameter will be stored.
minvalue_p	A pointer to the location where the minimum value of the parameter will be stored.
maxvalue_p	A pointer to the location where the maximum value of the parameter will be stored.

Parameters' Values:

Symbolic Name	Default	Minimum	Maximum
PARAM_ARGCK	**0**	**0**	**1**
PARAM_ARRAY	**0**	**0**	**1**
PARAM_USERP	**0**	**0**	**2**
PARAM_IISBND	**0**	**0**	**1**
PARAM_ITLIM	**2147483647**	**0**	**2147483647**
PARAM_NDLIM	**2147483647**	**0**	**2147483647**

PARAM_MIPLIM	2147483647	0	2147483647
PARAM_SAMPSZ	0	0	2147483647
PARAM_NOIMP	30	0	2147483647
PARAM_SCAIND	0	-1	1
PARAM_CRAIND	0	0	1
PARAM_LINVAR	0	0	1
PARAM_DERIV	0	0	3
PARAM_ESTIM	0	0	1
PARAM_DIREC	0	0	1
PARAM_RELAX	0	0	1
PARAM_PREPRO	0	0	1
PARAM_OPTFIX	0	0	1
PARAM_REORDR	0	0	1
PARAM_IMPBND	0	0	1
PARAM_REQBND	1	0	1

setdblparam

setdblparam() is a function that sets the value of a specified double parameter in a problem.

`setdblparam()` returns 0 if successful, 1 if `whichparam` isn't one of the valid codes for integer parameters, or 2 if `newvalue` is outside the range of valid values for the chosen parameter.

Syntax:

```
INTARG setdblparam (lp, whichparam, newvalue)
HPROBLEM lp;
INTARG whichparam;
REALARG newvalue;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem.
whichparam	One of the symbolic names or codes for double parameters: `PARAM_TILIM`, `PARAM_EPOPT`, `PARAM_EPPIV`, `PARAM_EPSOL`, `PARAM_EPRHS`, `PARAM_EPGAP`, `PARAM_CUTLO`, `PARAM_CUTHI`, `PARAM_EPNEWT`, `PARAM_EPCONV` or `PARAM_MUTATE`.
newvalue	The desired new value for the chosen parameter.

getdblparam

getdblparam() is a function that retrieves the value of a specified double parameter in a problem.

`getdblparam()` returns 0 if successful, or 1 if `whichparam` isn't one of the valid codes for double parameters.

Syntax:

```
INTARG getdblparam (lp, whichparam, value_p)
HPROBLEM lp;
INTARG whichparam;
LPREALARG value_p;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem.
whichparam	One of the symbolic names or codes for double parameters: `PARAM_TILIM`, `PARAM_EPOPT`, `PARAM_EPPIV`, `PARAM_EPSOL`, `PARAM_EPRHS`, `PARAM_EPGAP`, `PARAM_CUTLO`, `PARAM_CUTHI`, `PARAM_EPNEWT`, `PARAM_EPCONV` or `PARAM_MUTATE THI`.
value_p	A pointer to the location where the current value of the parameter will be stored.

infodblparam

infodblparam() is a function that retrieves the minimum, maximum, and default values of a specified double parameter in a problem.

`infodblparam()` returns 0 if successful, or 1 if `whichparam` isn't one of the valid codes for double parameters.

Syntax:

```
INTARG infodblparam (lp, whichparam, defvalue_p, minvalue_p,
                maxvalue_p)
HPROBLEM lp;
INTARG whichparam;
LPREALARG defvalue_p, minvalue_p, maxvalue_p;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem. This argument is ignored and may be NULL.
whichparam	One of the symbolic names or codes for double parameters: `PARAM_TILIM`, `PARAM_EPOPT`, `PARAM_EPPIV`, `PARAM_EPSOL`, `PARAM_EPRHS`, `PARAM_EPGAP`, `PARAM_CUTLO`, `PARAM_CUTHI`, `PARAM_EPNEWT`, `PARAM_EPCONV` or `PARAM_MUTATE`.
defvalue_p	A pointer to the location where the default value of the parameter will be stored.
minvalue_p	A pointer to the location where the minimum value of the parameter will be stored
maxvalue_p	A pointer to the location where the maximum value of the parameter will be stored.

Parameters' Values:

Symbolic Name	Default	Minimum	Maximum
PARAM_TILIM	65535.0	0.0	65535.0
PARAM_EPOPT	1.0E-5	1.0E-9	1.0E-4
PARAM_EPPIV	1.0E-6	1.0E-9	1.0E-4
PARAM_EPSOL	1.0E-4	1.0E-9	1.0E-4
PARAM_EPRHS	1.0E-8	1.0E-9	1.0E-4
PARAM_EPGAP	0.0	0.0	1.0
PARAM_CUTLO	-2.0E+30	-2.0E+30	2.0E+30
PARAM_CUTHI	2.0E+30	-2.0E+30	2.0E+30

PARAM_EPNEWT	1.0E-6	1.0E-9	1.0E-4
PARAM_EPCONV	1.0E-4	0.0	1.0
PARAM_MUTATE	0.075	0.0	1.0

setdefaults

setdefaults() is a function that sets all integer and double parameters for a specified problem to their default values.

The return value of `setdefaults()` is always 0.

To see the parameters' default values, refer to the Parameters' Values tables for functions `infointparam()` and `infodblparam()`.

Syntax:

```
INTARG setdefaults (lp)
HPROBLEM lp;
```

Parameters:

Parameter Name	Parameter Description
lp	Handle to the current problem.

Solver Parameters

Click to navigate to the desired topic:

- Integer Parameters (General)
- Integer Parameters (Solver Engines)
- Integer Parameters (Mixed-Integer Problems)
- Double Parameters

Integer Parameters (General)

The General parameters have (long) integer values and control some general features of the Solver DLL such as array argument passing conventions and the IIS finder.

Parameters' Codes and Usage:

Symbolic Name	Code	Usage in Solver DLLs
PARAM_ARGCK	**990**	On argument errors: `1`- MsgBox, `0` - retval only.
PARAM_ARRAY	**995**	`0` - Use C-style arrays, `1` - Use VB-style OLE SAFEARRAYs.
PARAM_USERP	**997**	`0` - Eval/Test mode, `1` - automatic reports, `2` - user-controlled reports.
PARAM_IISBND	**999**	`0` - Eliminate bounds, `1` - Don't eliminate bounds in finding IIS.
PARAM_ITLIM	**1020**	Limit on Solver Engine iterations.

Parameters Detailed Description:

Parameter Name	Description
PARAM_ARGCK	This parameter is unique to the Solver DLL and is used for debugging: When it is set to 1, if your program calls a Solver DLL function which detects an invalid argument, that function will display a Windows message box describing the error. This allows you to see the problem immediately, without having to write code to test the value returned by the DLL function.
PARAM_ARRAY	This parameter affects all of the Solver DLL functions which take arrays as arguments. If this parameter is `0` (the default), all array arguments supplied to the Solver, and all array arguments of callback functions (i.e. `funceval()` and `jacobian()`) are C-style arrays: The actual arguments are pointers to the base of the block of memory holding array values. If this parameter is 1, all array arguments are Visual Basic or OLE-style SAFEARRAYs: The actual arguments are pointers to pointers to SAFEARRAY descriptors, which contain pointers to the base of the block of memory holding array values.
PARAM_USERP	This parameter controls the reporting of "use" information in certain licensed versions of the Solver DLL. The default setting is 0, which means that the Solver DLL operates in "Evaluation/Test mode:" It will not keep track of the number of uses, or store anything in the Registry, but once every 10 minutes, it will display a message box on the system console.

Once the DLL is placed into service, this parameter should be set to `1` or `2`: In this case, the Solver DLL will count uses (calls to `optimize()` or `mipoptimize()`) and will store this information in the Registry. When `PARAM_USERP` is `1`, the DLL will automatically send an updated report of the cumulative number of uses by email to Frontline Systems (info@frontsys.com) each time it is run in a new calendar month. When `PARAM_USERP` is `2`, the DLL will not send any reports automatically, but reports may be generated and written to a text file or emailed to Frontline Systems, under control of the user's application, through calls to the `reportuse()` function.

PARAM_IISBND

Setting this parameter to 1 causes the Solver DLL's IIS finder to skip the steps (and computing time) required to eliminate variable bounds from an Irreducibly Infeasible Set of constraints. When `PARAM_IISBND` is `1`, any IIS which is found will include all of the variable bounds defined (in the `lb` and `ub` arrays) for the problem. When this parameter is `0` (the default), as many variable bounds as possible will be eliminated from the IIS.

PARAM_ITLIM

This is the maximum number of iterations (pivots) that the Solver will perform on an NLP, LP or QP problem (or a subproblem of a MIP problem). When this limit is exceeded, the `optimize()` function will return and the `solution()` function's `pstat` argument will be set to `PSTAT_IT_LIM_FEAS` or `PSTAT_IT_LIM_INFEAS`, depending on whether a feasible solution has been found.

Integer Parameters (Solver Engines)

The Solver Engine parameters have (long) integer values and control various features of the Nonlinear, Linear and Quadratic Solver Engines.

Parameters' Codes and Usage

Symbolic Name	Code	Usage in Solver DLLs
PARAM_SCAIND	**1033**	Scaling: -1-none, 0-normal, 1-aggressive.
PARAM_CRAIND	**1007**	Crashing: 0-none, 1-crash initial basis.
PARAM_LINVAR	**5010**	NLP linear variables: 0 - ignore, 1 - recognize linear variables.
PARAM_DERIV	**5011**	NLP derivatives: 0 - Forward, 1 - Central differencing; 2 - Use supplied `jacobian()` function, 3 - Use and check supplied `jacobian()`.
PARAM_ESTIM	**5012**	NLP estimates of basic variables: 0 - Tangent, 1 – Quadratic.
PARAM_DIREC	**5013**	NLP search direction: 0 - Quasi-Newton, 1 - Conjugate Gradient.
PARAM_SAMPSZ	**6001**	Evolutionary Solver population (sample) size.
PARAM_NOIMP	**6002**	Evolutionary Solver max time with no improvement.

Parameters Detailed Description

Parameter Name	Description
PARAM_SCAIND	The scaling option determines how the LP matrix, variable and constraint bounds are (re)scaled during the solution process. A value of -1 means no scaling, 0 means "normal" scaling, and 1 means "aggressive" scaling (used only by the Large-Scale LP/MIP Solver DLL). The default value of 0 will usually yield good results, including more accurate solutions and fewer total iterations.
PARAM_CRAIND	The "crashing" option is used only by the Large-Scale LP/MIP Solver DLL. A value of 1 indicates that the Solver should start by constructing an advanced basis, in which so-called "artificial" variables are eliminated in favor of "structural" variables wherever possible. This often results in fewer total iterations. A value of 0 means that the Solver should start from a "normal" basis.
PARAM_LINVAR	This parameter determines whether the nonlinear Solver Engine will attempt to recognize variables occurring linearly in all of the problem

functions, and then save time by assuming that first partial derivatives with respect to these variables are constant and need not be recomputed. This is an "aggressive" strategy which should be used only when you know that the nonlinearly occurring variables will not appear to change linearly over the intervals around their starting values.

PARAM_DERIV

This parameter controls the method used by the Solver to compute approximate first partial derivatives as follows:

Value	Meaning
0	"Forward differencing" will be used, perturbing variable values in one direction from the current point.
1	"Central differencing" will be used, perturbing variable values in two opposing directions from the current point. Central differencing takes more computing time, but can yield better results and fewer total iterations, especially if the Solver's path to the solution lies close to one or more constraint boundaries.
2	The `jacobian()` function, which must be supplied as an argument to `loadnlp()`, will be called to compute partial derivatives.
3	Used for debugging purposes: it means that the Solver will compute partial derivatives via forward differencing, then call the `jacobian()` function and compare the results. If the derivatives do not match within a small tolerance, the Solver will stop and return `PSTAT_FLOAT_ERROR`; however if `PARAM_ARGCK` is 1, the Solver will display Windows message boxes reporting each mismatching derivative, and will return `PSTAT_FLOAT_ERROR` only if you click Cancel in a message box.

PARAM_ESTIM

This parameter controls the method used to estimate initial values for the basic variables at the beginning of each one-dimensional line search. Zero value means that the Solver will use linear extrapolation from the line tangent to the reduced objective function. Value of 1 means that the Solver will extrapolate to the minimum (or maximum) of a quadratic fitted to the reduced objective at its current point.

PARAM_DIREC

This parameter controls the method used by the Solver to determine a direction vector for the variables being changed on a given iteration. 0 means that the Solver will use a Quasi-Newton method, maintaining an approximate Hessian matrix for the reduced objective function. Value of 1 means that the Solver will use a Conjugate Gradient method, which does not require the Hessian matrix

PARAM_SAMPSZ

This parameter determines the number of points in the Evolutionary Solver's population of candidate solutions. The default value is 0, which

instructs the Evolutionary Solver to select a "reasonable" population size for the problem (currently 10 times the number of variables in the problem, but no more than 200). In general, a larger population size will permit more diversity among candidate solutions, but may slow down convergence towards an accepted solution.

PARAM_NOIMP

This parameter controls one of the stopping criteria used by the Evolutionary Solver Engine: If the Solver does not find an improved solution within the time (in seconds) specified by this parameter, it will stop and return from the call to `optimize()` with the best solution found so far. An "improved solution" is a feasible solution where the relative improvement in the objective, compared to the best solution found previously, exceeds the `PARAM_EPGAP` setting.

Integer Parameters (Mixed-Integer Problems)

Parameters' Codes and Usage

Symbolic Name	Code	Usage in Solver DLLs
PARAM_NDLIM	**2017**	Limit on Branch & Bound nodes explored.
PARAM_MIPLIM	**2015**	Limit on Branch & Bound IP solutions found.
PARAM_RELAX	**2501**	Solve relaxation: `0` - normal problem, `1` - ignore integer variables.
PARAM_PREPRO	**2502**	P&P Probing: `0` - Off, `1` - On.
PARAM_OPTFIX	**2503**	P&P Optimality Fixing: `0` - Off, `1` - On.
PARAM_REORDR	**2504**	P&P Branch Var Reordering: `0` - Off, `1` - On.
PARAM_IMPBND	**2505**	P&P Bounds Improvement: `0` - Off, `1` - On.

Parameters Detailed Description

Parameter Name	Description
PARAM_NDLIM	This is the maximum number of nodes, or LP subproblems, that the Branch & Bound algorithm will fathom on a MIP problem. When this limit is exceeded, the `mipoptimize()` function will return and the `solution()` function's `pstat` argument will be set to `PSTAT_MIP_NODE_LIM_FEAS` or `PSTAT_MIP_NODE_LIM_INFEAS`, depending on whether a feasible integer solution has been found.
PARAM_MIPLIM	This is the maximum number of feasible integer solutions that the Branch & Bound algorithm will find on a MIP problem. When this limit is exceeded, the `mipoptimize()` function will return and the `solution()` function's `pstat` argument will be set to `PSTAT_MIP_SOL_LIM` (there is always a feasible integer solution at this point).
PARAM_RELAX	Setting this parameter to 1 causes the Solver DLL to ignore any integer variables defined by a call to `loadctype()`, and instead to solve the "relaxation" of the mixed-integer programming problem. It is effective for both linear and nonlinear problems. When this parameter is set to 0 (the default), the integer variables are taken into account and the Branch & Bound algorithm is used to find an integer optimal solution.
PARAM_PREPRO	Setting this parameter to 1 activates the Probing strategy for MIP problems (in the Solver DLL Plus). The Probing strategy allows the Solver to derive values for certain binary integer variables based on the settings of others, prior to actually solving the problem. When the Branch & Bound method

creates a subproblem with an additional (tighter) bound on a binary integer variable, this causes the variable to be fixed at 0 or 1. In many problems, this has implications for the values of other binary integer variables which can be discovered through Probing. For example, your model may have a constraint such as $x_1 + x_2 + x_3 + x_4 + x_5, \ 1$ where x_1 through x_5 are all binary integer variables. Whenever one of these variables is fixed at 1, all of the others are forced to be 0; Probing allows the Solver to determine this before solving the problem. In some cases, the Feasibility tests performed as part of Probing will determine that the subproblem is infeasible, so it is unnecessary to solve it at all. (This is a special case of a "clique" or "Special Ordered Set" (SOS) constraint; the Solver recognizes these constraints in their most general form.)

PARAM_OPTFIX

Setting this parameter to 1 activates the Optimality Fixing strategy for MIP problems (in the Solver DLL Plus). The Optimality Fixing strategy is another way to fix the values of binary integer variables before the subproblem is solved, based on the signs of the coefficients of these variables in the objective and the constraints. Optimality Fixing can lead to further opportunities for Probing and Bounds Improvement, and vice versa.

PARAM_REORDR

Setting this parameter to 1 activates the Variable Reordering strategy for MIP problems (in the Solver DLL Plus). The Variable Reordering strategy attempts to improve the order in which the Branch & Bound algorithm chooses integer variables to "branch" upon, based on the relative magnitudes of their coefficients in the objective and constraints. The goal is to choose integer variables to "branch" upon which have a large impact on the values which may be assumed by other variables in the problem. For example, you might have a binary integer variable which indicates whether or not a new plant will be built, and other variables which then determine whether certain manufacturing lines will be started up. You would like the Solver to "branch" upon the plant-building variable as early as possible, since its setting will eliminate many other possibilities which would otherwise have to be considered during the solution of each subproblem. If this parameter is set to 0, the selection of integer variables to branch upon is guided by the order of the variables in the arrays defining your problem.

PARAM_IMPBND

Setting this parameter to 1 activates the Bounds Improvement strategy for MIP problems (in the Solver DLL Plus). The Bounds Improvement strategy allows the Solver to tighten the bounds on variables which are not 0-1 or binary integer variables, based on the values which have been derived for the binary variables, before the problem is solved. Tightening the bounds usually reduces the effort required by the Simplex or other Solver Engine to find the optimal solution, and it some cases it leads to an immediate determination that the subproblem is infeasible and need not be solved.

Double Parameters

These parameters have double values. Parameters whose names begin with EP are used to set tolerances in the LP, QP and MIP solution algorithms.

Parameters' Codes and Usage

Symbolic Name	Code	Usage in Solver DLLs
PARAM_TILIM	**1038**	Time limit for `optimize()`/`mipoptimize()`.
PARAM_EPOPT	**1014**	LP optimality tolerance.
PARAM_EPPIV	**1091**	LP pivot tolerance.
PARAM_EPSOL	**1092**	Large-Scale LP solution tolerance.
PARAM_EPRHS	**1016**	LP feasibility tolerance.
PARAM_EPGAP	**2009**	Integer tolerance/MIP gap.
PARAM_CUTLO	**2006**	Known incumbent for max MIP.
PARAM_CUTHI	**2007**	Known incumbent for min MIP.
PARAM_EPNEWT	**5001**	NLP constraint satisfaction tolerance.
PARAM_EPCONV	**5002**	NLP tolerance for slowly changing objective.
PARAM_MUTATE	**6010**	Evolutionary Solver mutation probability.

Parameters Detailed Description

Parameter Name	Description
PARAM_TILIM	This is the maximum number of seconds that the Solver spend on the overall LP or MIP problem. When this limit is exceeded, the `optimize()` function will return and the `solution()` function's `pstat` argument will be set to `PSTAT_TIME_LIM_FEAS` or `PSTAT_TIME_LIM_INFEAS` for an LP problem, depending on whether a feasible solution has been found. On a MIP problem, the `pstat` argument will be set to `PSTAT_MIP_TIME_LIM_FEAS` or `PSTAT_MIP_TIME_LIM_INFEAS`, depending on whether a feasible integer solution has been found.
PARAM_EPOPT	This is the "optimality tolerance" or "reduced cost tolerance" for LP problems. Variables whose reduced cost is less than the negative of this tolerance are candidates for entering the basis.

PARAM_EPPIV	This is the "pivot tolerance" for LP problems. LP matrix elements must have an absolute value greater than or equal to this tolerance to be candidates for pivoting.
PARAM_EPSOL	This is the "solution tolerance" in the Large-Scale LP/MIP Solver DLL. Any calculated basic variable whose absolute value is less than this tolerance is treated as zero.
PARAM_EPRHS	This is the "feasibility tolerance," within which constraints are considered satisfied and values for decision variables are treated as integer in MIP problems.
PARAM_EPGAP	This is the (relative) integer objective "gap tolerance" for MIP problems. On such problems, it often happens that the Branch & Bound algorithm will find a good solution fairly quickly, but will require a great deal of time to find (or verify that it has found) the optimal integer solution. The integer gap tolerance setting may be used to tell the Solver to stop if the best solution so far is "close enough." The Solver computes the absolute value of the difference between the objective value for the best integer solution so far (the "incumbent") and the "best bound" on the objective value found so far by the Branch & Bound process, divided by the best bound's objective value. If this relative difference is less than the integer gap tolerance, the Solver stops and returns the incumbent as the solution. Initially, the objective value of the LP "relaxation" of the problem (ignoring the integer constraints) serves as the best bound, since the all-integer solution can be no better than this. During the Branch & Bound process, the best bound is updated based on the subproblems that have already been explored. By default, the integer gap tolerance is zero, which means that the Solver will continue searching until all alternatives have been explored and the optimal integer solution has been found. You can often save significant solution time by setting an integer gap tolerance, greater than zero, which is sufficient for your application.
PARAM_CUTLO	This is the "integer objective cutoff value" for MIP maximization problems. In the Branch & Bound algorithm, any subproblem whose objective is less than this value will not be fathomed. If you have an objective value for a known integer solution to this problem, for example from a previous call to the Solver, you can use it to set this parameter and save on solution time.
PARAM_CUTHI	This is the "integer objective cutoff value" for MIP minimization problems. In the Branch & Bound algorithm, any subproblem whose objective is greater than this value will not be fathomed. If you have an objective value for a known integer solution to this problem, for example from a previous call to the Solver, you can use it to set this parameter and save on solution time.
PARAM_EPNEWT	This is the tolerance within which constraints will be considered "binding," or satisfied with equality, during the solution process.
PARAM_EPCONV	This is the tolerance used in the nonlinear GRG Solver's and the

Evolutionary Solver's test for a "slowly changing objective:" In the GRG Solver, if the relative change in the objective function is less than this value for the last five iterations, the Solver stops and returns `PSTAT_FRAC_CHANGE` in the `pstat` argument of the `solution()` function. In the Evolutionary Solver, if 99% of the population members have "fitness values" that differ by less than this value, the Solver stops and returns `PSTAT_FRAC_CHANGE`.

PARAM_MUTATE

This is the probability that the Evolutionary Solver Engine, on one of its major iterations, will attempt to generate a new trial point by "mutating" or altering one or more variable values of a current point in the population of candidate solutions.

Optimization Problems Reference and Code Examples

This section contains examples of programmatically implemented problems of the basic optimization problem types organized in individual topics as follows:

1. Linear Problems
2. Quadratic Problems
3. Nonlinear Problems
4. Quadratically Constrained Problems
5. Nonsmooth Problems

From these general problem types, furher types can be constructed by adding special constraints on the variables, like being requirements for integer, binary, or alldifferent. The resulting optimization sub-types are not covered explicitly in this section. You can find information about them in the topics where the particular properties implementing these constraints are dicussed.

Each topic contains:

1. The exact mathematical definition of the problem type.
2. An example problem described in algebraic notation.
3. The same problem implemented in the Solver SDK Platform in the following programming languages:
    a. VB.NET
    b. C++
    c. C#
    d. Procedural C

All topics in this section >

Linear Problems

Definition

Linear programming (LP) problems, or "linear problems" for short, are optimization problems in which the objective function and the constraints are all linear. Mathematically, linear problems look like this:

$$\begin{aligned} &\text{Minimize/Maximize:} && cx \\ &\text{subject to:} && l \leq Ax \leq u \\ & && b \leq x \leq ub \end{aligned}$$

Examples >

Linear Problem Examples

Example Problem in Algebraic Notation

$$\begin{aligned} &\text{Minimize/Maximize:} && 10\,x_1 + 15\,x_2 + 10\,x_3 + 15\,x_4 \\ &\text{subject to:} && 4\,x_1 + 4\,x_2 + 4\,x_3 + 4\,x_4 \leq 120 \\ & && 4\,x_1 + 2\,x_2 \leq 80 \\ & && 2\,x_1 + 5\,x_2 \leq 60 \\ & && 5\,x_3 + 3\,x_4 \leq 60 \\ & && 5\,x_3 + 3\,x_4 \leq 60 \end{aligned}$$

$$\begin{aligned} &\text{where} && 0 \leq x_1 \leq 10 \\ & && 0 \leq x_2 \leq 9 \\ & && 0 \leq x_3 \leq 1 \\ & && 0 \leq x_4 \leq 13 \end{aligned}$$

(Solution: $x_1 = 9.17$
$x_2 = 8.33$
$x_3 = 0$
$x_4 = 12.5$
objective = 404.15)

The exmple problem in VB.NET follows, click the respective link to navigate to the same problem in C++, C#, and VB.NET.

The Example Problem in VB.NET

```
Private Sub linear_example()

   ' The matrix is passed row-wise, dense.

   Dim nvars As Integer, ncons As Integer
   nvars = 4    ' 4 decision variables.
   ncons = 5    ' 5 constraint functions.

   Dim rhsValue(4) As Double
   rhsValue(0) = 120
   rhsValue(1) = 80
   rhsValue(2) = 60
   rhsValue(3) = 60
   rhsValue(4) = 75

   Dim rhs(3) As Double
   rhs(0) = 10
   rhs(1) = 9
   rhs(2) = 1
   rhs(3) = 13

   Dim fcnObj As New DoubleMatrix
   With fcnObj
      .Create(Rows:=1, Columns:=4)
      .Value(0, 0) = 10
      .Value(0, 1) = 15
      .Value(0, 2) = 10
      .Value(0, 3) = 15
   End With
   Dim fcnConstr As New DoubleMatrix
   With fcnConstr
      Create(Rows:=5, Columns:=4)
      ' This will create a sparse matrix,  so we don't set zero
      ' values  in  the  matrix.  Although  this  approach  is
      ' perfectly valid, for a very big matrix, calling Value()
      ' repeatedly can become  slow.  In that case,  using  the
      ' InitSparse()  method,  although a bit more work for the
      ' user, will be faster.
      .Value(0, 0) = 4
      .Value(0, 1) = 4
      .Value(0, 2) = 4
```

```
    .Value(0, 3) = 4
    .Value(1, 0) = 4
    .Value(1, 1) = 2
    '.Value(1, 2) = 0
    '.Value(1, 3) = 0
    .Value(2, 0) = 2
    .Value(2, 1) = 5
    '.Value(2, 2) = 0
    '.Value(2, 3) = 0
    '.Value(3, 0) = 0
    '.Value(3, 1) = 0
    .Value(3, 2) = 5
    .Value(3, 3) = 3
    '.Value(4, 0) = 0
    '.Value(4, 1) = 0
    .Value(4, 2) = 5
    .Value(4, 3) = 6
End With

Dim prob As SolverPlatform.Problem = Nothing

Try

   prob = New SolverPlatform.Problem   ' Creating a Problem.

   prob.VarDecision.Size = nvars       ' 4 variables.
   prob.FcnConstraint.Size = ncons     ' 5 constraints.

   ' Right-hand sides:
   prob.FcnConstraint.UpperBound.Array = rhsValue

   prob.VarDecision.NonNegative()    ' Variables ≥ 0.

   prob.VarDecision.UpperBound.Array = rhs

   ' Linear model; passing the coefficients for the objective
   ' function(s):
   prob.Model.AllLinear(Function_Type.Objective,
           Variable_Type.Decision) = fcnObj
```

```
        ' Linear model; passing the coefficients for the
        ' constraint function(s):
        prob.Model.AllLinear(Function_Type.Constraint,
                Variable_Type.Decision) = fcnConstr


        prob.Solver.SolverType = Solver_Type.Maximize


        prob.Solver.Optimize()            ' Solving the model.


        Dim status As Optimize_Status
        ' Optimization status:
        status = prob.Solver.OptimizeStatus


        Dim varvalue() As Double
        ' Final variable values:
        varvalue = prob.VarDecision.FinalValue.Array


        Dim objValue() As Double
        objValue = prob.FcnObjective.FinalValue.Array


        ' Note   that   the   FcnConstraint,   VarDecision,   and
        ' FcnObjective  properties  all  point  to  the  "current"
        ' constraint function block, variable block, and objective
        ' block. In this example,  we only have one block of each.


        textOut.Text = textOut.Text & "Optimize Status = " &
                status & Chr(13) & Chr(10)


        textOut.Text = textOut.Text & "x1 = " & varvalue(0) &
                Chr(13) & Chr(10)
        textOut.Text = textOut.Text & "x2 = " & varvalue(1) &
                Chr(13) & Chr(10)
        textOut.Text = textOut.Text & "x3 = " & varvalue(2) &
                Chr(13) & Chr(10)
        textOut.Text = textOut.Text & "x4 = " & varvalue(3) &
                Chr(13) & Chr(10)


        textOut.Text = textOut.Text & "Obj = " & objValue(0) &
                Chr(13) & Chr(10)

    Catch ex As SolverException
```

```
            textOut.Text = textOut.Text & ex.Message & Chr(13) &
                    Chr(10)
            textOut.Text = textOut.Text & ex.ExceptionType & Chr(13) &
                    Chr(10)

        End Try

        If Not prob Is Nothing Then
            prob.Dispose()     ' Freeing up the license the right way.
        End If

        fcnObj = Nothing
        fcnConstr = Nothing

    End Sub
```

The Example Problem in C++

```
int linear_example()
    {
        try
        {
            int nvars = 4, ncons = 5;
            double objcoeff[] = { 10, 15, 10, 15 };

            // Passing the coefficients row-wise:
            double concoeff[] = { 4, 4, 4, 4, 4, 2, 0, 0, 2, 5, 0,
                  0, 0, 0, 5, 3, 0, 0, 5, 6 };

            double rhsvalue[] = { 120, 80, 60, 60, 75 };
            double rhs[] = { 13, 13 , 13, 13 };

            CProblem problem;

            // 4 decision variables:
            problem.VarDecision.Size = nvars;

            // 5 constraint functions:
            problem.FcnConstraint.Size = ncons;

            problem.Solver.SolverType = Solver_Type_Maximize;
```

```
// Constraints right-hand sides:
problem.FcnConstraint.UpperBound = rhsvalue;


problem.VarDecision.UpperBound = rhs;


problem.VarDecision.NonNegative(); // Variables ≥ 0.


/* Linear model; Passing the coefficients for the
   objective function(s):                          */
problem.Model.AllLinear[Function_Type_Objective] =
     CDoubleMatrix(Array_Order_ByRow, 1, nvars,
     objcoeff);


/* Linear model; Passing the coefficients for the
   constraint function(s):                         */
problem.Model.AllLinear[Function_Type_Constraint] =
     CDoubleMatrix (Array_Order_ByRow, ncons, nvars,
     concoeff);


/* By calling the AllLinear functions, the problem type
   has been automatically updated to "linear"       */


problem.Solver.Optimize();         //Solving the model.


// Optimization status:
_Optimize_Status status =
     problem.Solver.OptimizeStatus;


// Final variable values:
const double* varvalue =
     problem.VarDecision.FinalValue;


cout << "linear_example" << endl;
cout << "Status = " << status << endl;
cout << "x1 = " << varvalue[0] << endl;
cout << "x2 = " << varvalue[1] << endl;
cout << "x3 = " << varvalue[2] << endl;
cout << "x4 = " << varvalue[3] << endl;
cout << "Obj = " << problem.FcnObjective.FinalValue[0]
     << endl;
cout << endl;
```

```
      } catch(CSolverException e)
      {
         // Throwing in an exception in case of an error:
         cout << "linear_example" << endl;
         cout << "Throws Exception" << endl;
         cout << "Code = " << e.ErrorCode << endl;
         cout << "Text = " << ToCStr(e.Description).c_str() <<
               endl;
         return 1;
      }
      return 0;
   }
```

The Example Problem in C#

```
private void linear_example()
{
   try
   {
      int nvars = 4;       // 4 decision variables.
      int ncons = 5;       // 5 constraint functions.

      double [] rhsValue = { 120, 80, 60, 60, 75 };
      double [] rhs = { 10, 9, 1, 13 };

      DoubleMatrix fcnObj = new DoubleMatrix(1, nvars);
      fcnObj[0,0] = 10;
      fcnObj[0,1] = 15;
      fcnObj[0,2] = 10;
      fcnObj[0,3] = 15;

      DoubleMatrix fcnConstr =  new DoubleMatrix(ncons,nvars);
      fcnConstr[0,0] = 4;
      fcnConstr[0,1] = 4;
      fcnConstr[0,2] = 4;
      fcnConstr[0,3] = 4;
      fcnConstr[1,0] = 4;
      fcnConstr[1,1] = 2;
      fcnConstr[1,2] = 0;
      fcnConstr[1,3] = 0;
```

```
fcnConstr[2,0] = 2;
fcnConstr[2,1] = 5;
fcnConstr[2,2] = 0;
fcnConstr[2,3] = 0;
fcnConstr[3,0] = 0;
fcnConstr[3,1] = 0;
fcnConstr[3,2] = 5;
fcnConstr[3,3] = 3;
fcnConstr[4,0] = 0;
fcnConstr[4,1] = 0;
fcnConstr[4,2] = 5;
fcnConstr[4,3] = 6;

// Creating a Problem:
using(Problem problem = new Problem())
{
   problem.VarDecision.Size = nvars;
   problem.FcnConstraint.Size = ncons;

   problem.FcnConstraint.UpperBound.Array = rhsValue;
   problem.VarDecision.NonNegative();
   problem.VarDecision.UpperBound.Array = rhs;

   problem.Model.AllLinear[Function_Type.Objective] =
         fcnObj;
   problem.Model.AllLinear[Function_Type.Constraint] =
         fcnConstr;

   problem.Solver.SolverType = Solver_Type.Maximize;

   problem.Solver.Optimize();     // Solving the model.

   // Optimization status:
   Optimize_Status status = problem.Solver.OptimizeStatus;

   // Final variable values:
   double [] varvalue =
         problem.VarDecision.FinalValue.Array;

   double [] objValue =
         problem.FcnObjective.FinalValue.Array;
```

```
            /* Note that the FcnConstraint, VarDecision, and
               FcnObjective properties all point to the "current"
               constraint function block, variable block, and
               objective block. In this example, we only have one
               block of each.                                    */

            m_Log.Dump("Optimize Status = " + status);

            m_Log.Dump("x1 = " + varvalue[0]);
            m_Log.Dump("x2 = " + varvalue[1]);
            m_Log.Dump("x3 = " + varvalue[2]);
            m_Log.Dump("x4 = " + varvalue[3]);

            m_Log.Dump("Obj = " + objValue[0]);
         }
      }

      catch(SolverException ex)
      {
         m_Log.Dump("Exception " + ex.ExceptionType);
         m_Log.Dump("Exception " + ex.Message);
      }
   }
```

The Example Problem in Procedural C

```
void linear_example()
{
   // 4 decision variables, 5 constraint functions.
   int nvars = 4, ncons = 5;
   double objcoeff[] = { 10, 15, 10, 15 };
   // Passing the coeffiecients row-wise:
   double concoeff[] = { 4, 4, 4, 4, 4, 2, 0, 0, 2, 5, 0, 0, 0,
               0, 5, 3, 0, 0, 5, 6 };
   double rhsvalue[] = { 120, 80, 60, 60, 75 };
   double rhs[] = { 10, 9, 1, 13 };

   double objval;
   double x[4];
   void * prob;
```

```
// Creating a Problem:
SolverProblem(&prob);

// Setting maximizing objective:
SolverProbTypeSet(prob, Solver_Type_Maximize,
            Problem_Type_NA);
// Creating a variable block:
SolverVariableAdd(prob, L"Vars", Variable_Type_Decision,
            nvars);
// Creating a Constraint block:
SolverFunctionAdd(prob, L"Constraints",
            Function_Type_Constraint, ncons);
// Creating an Objective block
SolverFunctionAdd(prob, L"Objective",
            Function_Type_Objective,  1);

/* Passing the right-hand sides of the constraints (There are
   two ways to do it: we can pass the lower and upper bounds
   on the functions, as we do here, or we can use the
   Relation method.):                                        */
SolverFcnUpperBoundSet(prob, L"Constraints", rhsvalue);

/* Varaibles have to be non-negative. Note that VarDecision
   points to the "current" block of variables. In this
   example we only have one block of variables, so pointing
   to all variables is OK:                                   */
SolverVarNonNegative(prob, L"Vars");
SolverVarUpperBoundSet(prob,L"Vars",rhs);

// The matrix is dense and passed row-wise:
SolverModArrayOrder(prob, Array_Order_ByRow);
/* Linear model; passing the coefficients for the objective
   function(s):                                              */
SolverModAllLinearSet(prob, Function_Type_Objective,
            Variable_Type_All , objcoeff);
/* Linear model; passing the coefficients for the constraint
   function(s):                                              */
SolverModAllLinearSet(prob, Function_Type_Constraint,
            Variable_Type_All, concoeff);

/* By calling the AllLinear functions, the problem type has
   been automatically updated to linear.                     */

_Optimize_Status status;
```

```
    /* Solving the model (Note that we have not picked an Engine;
       the Solver will determine best Engine by checking the
       problem type; we get the status of the optimization; the
       main status values are typically "optimal", "infeasible",
       or "unbounded", but many others are possible, too):     */
    SolverOptimize(prob, &status, &objval);

    // Getting the final values of the variables:
    SolverVarFinalValue(prob, L"Vars", x);

    // Releasing the memory used by freeing the Problem:
    SolverProbFree(&prob);

    printf("linear_example\n");
    printf("x[0] = %g\n", x[0]);
    printf("x[1] = %g\n", x[1]);
    printf("x[2] = %g\n", x[2]);
    printf("x[3] = %g\n", x[3]);

    printf("Obj = %g\n", objval);
    printf("Status = %d\n", status);

    return;
}
```

Quadratic Problems

Definition

Quadratic programming (QP) problems, or "quadratic problems" for short, are optimization problems which mathematically look like this:

$$\begin{aligned} &\text{Minimize} && xQx + cx \\ &\text{subject to:} && l \leq Ax \leq u \\ & && lb \leq x \leq ub \end{aligned}$$

Examples >

Quadratic Problem Examples

⬇ Example 1 (Using Sparse Matrix (Except for Procedural C)) ⬇

The Example Problem in Algebraic Notation

$$\begin{aligned} &\text{Minimize:} && x_1 * x_1 - 4x_1 - 2x_2 \\ &\text{subject to:} && x_1 + x_2 \leq 4 \\ & && 2x_1 + x_2 \leq 5 \\ & && x_1 - 4x_2 \leq -2 \\ &\text{where} && 0 \leq x_1 \\ & && 0 \leq x_2 \end{aligned}$$

(Solution: $x_1 = 1$
$x_2 = 3$
objective = -9)

The Example Problem in VB.NET

```
Private Sub quadratic_example1()

   Dim nvars As Integer, ncons As Integer
   nvars = 2      ' 2 decision variables.
   ncons = 3      ' 3 constraint functions.

   Dim rhsValue(2) As Double
   rhsValue(0) = 4
   rhsValue(1) = 5
```

```
rhsValue(2) = -2
Dim fcnObj As New DoubleMatrix
With fcnObj
   .Create(Rows:=1, Columns:=2)
   .Value(0, 0) = -4
   .Value(0, 1) = -2
End With

' The matrix is passed sparse. (Note that a Q matrix
' (QMatVal) is typically symmetrical, so it does not matter
' if you pass the coefficents row-wise or column-wise).

Dim QMatVal As New DoubleMatrix

' We have only x1*x1 in the objective function, with
' coefficent 1 in front, so for presenting the QMatVal, we
' will have only the line .Value(0, 0) = 1, because the
' matrix is sparse and we omit the zeroes.
'          x1   x2
'    x1    1    0
'    x2    0    0

With QMatVal
   .Create(Rows:=2, Columns:=2)
   .Value(0, 0) = 1
  ' The other values are zeroes. We could add them, but this
  ' would serve no purpose because the matrix is sparse.

End With

Dim fcnConstr As New DoubleMatrix
With fcnConstr
   .Create(Rows:=3, Columns:=2)
   .Value(0, 0) = 1
   .Value(0, 1) = 1
   .Value(1, 0) = 2
   .Value(1, 1) = 1
   .Value(2, 0) = 1
   .Value(2, 1) = -4
End With

Dim prob As SolverPlatform.Problem = Nothing
```

```
Try

    prob = New SolverPlatform.Problem    ' Creating a Problem.

    prob.VarDecision.Size = nvars        ' 2 variables.
    prob.FcnConstraint.Size = ncons      ' 3 constraints.

    ' The Right-hand sides:
    prob.FcnConstraint.UpperBound.Array = rhsValue

    prob.VarDecision.NonNegative()       ' Variables ≥ 0.

    ' Linear model; Passing the coefficients for the objective
    ' function(s):
    prob.Model.AllLinear(Function_Type.Objective,
            Variable_Type.Decision) = fcnObj

    ' Passing quadratic coefficient for  the  objective
    ' function; by calling this method,  the problem is
    ' automatically cahnged  from linear to quadratic:
    prob.Model.FcnQuadratic(Function_Type.Objective, 0,
            Variable_Type.Decision) = QMatVal

    ' Linear model; Passing the coefficients for the
    ' constraint function(s):
    prob.Model.AllLinear(Function_Type.Constraint,
            Variable_Type.Decision) = fcnConstr

    prob.Solver.SolverType = Solver_Type.Minimize

    prob.Solver.Optimize()              ' Solving the model.

    Dim status As Optimize_Status
    status = prob.Solver.OptimizeStatus ' Optimization status.

    Dim varvalue() As Double
    ' Final variable values:
    varvalue = prob.VarDecision.FinalValue.Array

    Dim objValue() As Double
    objValue = prob.FcnObjective.FinalValue.Array
```

```
        ' Note that the FcnConstraint, VarDecision, and
        ' FcnObjective properties all point to the "current"
        ' constraint function block, variable block, and objective
        ' block. In this example, we only have one block of each.

        Console.WriteLine("Optimize Status = " & status )

        Console.WriteLine( "x1 = " & varvalue(0) )
        Console.WriteLine( "x2 = " & varvalue(1) )
        Console.WriteLine( "Obj = " & objValue(0) )

    Catch ex As SolverException

        Console.WriteLine( ex.Message )
        Console.WriteLine( ex.ExceptionType )
    End Try

    If Not prob Is Nothing Then
        prob.Dispose()      ' Freeing up the license the right way.
    End If

    fcnObj = Nothing
    fcnConstr = Nothing

    End Sub
```

The Example problem in C++

```
int quadratic_example1()
{
    try
    {
        /* 2 decision variables,
           3 constraint functions:                                  */
        int nvars = 2, ncons = 3;
        double objcoeff[] = { -4, -2 };

    /* The matrix is passed sparse. (Note that a Q matrix
       (QMatVal) is typically symmetrical, so it does not matter
       if you pass the coefficents row-wise or column-wise).
```

```
   We have only x1*x1 in the objective function, with
   coefficent 1 in front, so for presenting the QMatVal, we
   will have only the line qmatval.Value[0][0]= 1, because
   the matrix is sparse and we omit the zeroes.
                              x1   x2
                        x1     1    0
                        x2     0    0
          */

CDoubleMatrix qmatval =
           CDoubleMatrix(Array_Order_ByRow,nvars,nvars);
qmatval.Value[0][0]= 1;
   /* The other values are zeroes. We could add them, but
      this would serve no purpose because the matrix is
      sparse.                                            */

   // Note that the coefficients are passed row-wise:
   double concoeff[] = { 1, 1, 2, 1, 1, -4 };
   double rhsvalue[] = { 4, 5, -2 };

   /* One of the possible ways to create a Problem. The type,
      the number of variables and the number of constraints
      are passed in the constructor:                     */
   CProblem problem(Solver_Type_Minimize, nvars, ncons);

   // Constraint right-hand sides:
   problem.FcnConstraint.UpperBound = rhsvalue;

   problem.VarDecision.NonNegative();>     // Variables ≥ 0.

   /* The Q matrix specifies the covariance between each pair
      of assets                                          */

   /* Linear model; Passing the coefficients for the
      objective function(s):                             */
   problem.Model.AllLinear[Function_Type_Objective] =
           CDoubleMatrix(Array_Order_ByRow, 1, nvars,
           objcoeff);

   /* Linear model; Passing the coefficients for the
      constraint function(s):                            */
   problem.Model.AllLinear[Function_Type_Constraint] =
           CDoubleMatrix (Array_Order_ByRow, ncons, nvars,
           concoeff);
```

```
      /* By calling the AllLinear functions, the Problem Type
         has been automatically updated to linear.          */

      /* Passing quadratic coefficient for the objective
         function; by calling this method, the problem is now
         automatically changed from linear to quadratic:     */
      problem.Model.FcnQuadratic[Function_Type_Objective][proble
            m.get_ObjectiveIndex()][Variable_Type_Decision] =
            qmatval;
      problem.Solver.Optimize();
      _Optimize_Status status = problem.Solver.OptimizeStatus;

      // Final variable values:
      const double* varvalue = problem.VarDecision.FinalValue;

      cout << "quadratic_example" << endl;
      cout << "Status = " << status << endl;
      cout << "x1 = " << varvalue[0] << endl;
      cout << "x2 = " << varvalue[1] << endl;

      cout << "Obj = " << problem.FcnObjective.FinalValue[0] <<
            endl;
      cout << endl;

   } catch(CSolverException e)
   {
      cout << "quadratic_example" << endl;
      cout << "Throws Exception" << endl;
      cout << "Code = " << e.ErrorCode << endl;
      cout << "Text = " << ToCStr(e.Description).c_str() <<
            endl;
      return 1;
   }
   return 0;
}
```

The Example problem in C#

```
Private void quadratic_example1()

   try
   {
```

```
int nvars = 2;      // 2 decision variables.
int ncons = 3;      // 3 constraint functions.

double [] rhsValue = { 4, 5, -2 };

DoubleMatrix fcnObj = new DoubleMatrix(1, nvars);
fcnObj[0,0] = -4;
fcnObj[0,1] = -2;


DoubleMatrix fcnConstr = new DoubleMatrix(ncons,nvars);
fcnConstr[0,0] = 1;
fcnConstr[0,1] = 1;
fcnConstr[1,0] = 2;
fcnConstr[1,1] = 1;
fcnConstr[2,0] = 1;
fcnConstr[2,1] = -4;

DoubleMatrix qmatval = new DoubleMatrix(nvars,nvars);

/* The matrix is passed sparse. (Note that a Q matrix
   (QMatVal) is typically symmetrical, so it does not
   matter if you pass the coefficents row-wise or column-
   wise).
   We have only x1*x1 in the objective function, with
   coefficent 1 in front, so for presenting the QMatVal,
   we will have only the line qmatval[0][0]= 1, because
   the matrix is sparse and we omit the zeroes.
          x1   x2
     x1    1    0
     x2    0    0
*/

qmatval[0,0] = 1
/* The other values are zeroes. We could add them, but
   this would serve no purpose because the matrix is
   sparse.                                              */

/* One of the possible ways to create a Problem. The type,
   the number of variables and the number of constraints
   are passed in the constructor:                       */
using(Problem problem = new Problem(Solver_Type.Minimize,
        nvars, ncons))
{
```

```
// Constraint right-hand sides:
problem.FcnConstraint.UpperBound.Array = rhsValue;
problem.VarDecision.NonNegative();    // Variables ≥ 0.

/* Linear model; Passing the coefficients for the
   objective function(s):                          */
problem.Model.AllLinear[Function_Type.Objective] =
     fcnObj;

/* Passing quadratic coefficient for the objective
   function; by calling this method, the problem is now
   automatically cahnged from linear to quadratic:   */
problem.Model.FcnQuadratic[Function_Type.Objective,
     problem.ObjectiveIndex] = qmatval;

/* Linear model; Passing the coefficients for the
   constraint function(s):                         */
problem.Model.AllLinear[Function_Type.Constraint] =
     fcnConstr;

problem.Solver.Optimize();        // Solving the model.

// Optimization status:
Optimize_Status status = problem.Solver.OptimizeStatus;

// Final variable values:
double [] varvalue =
     problem.VarDecision.FinalValue.Array;

double [] objValue =
     problem.FcnObjective.FinalValue.Array;

/* Note that the FcnConstraint, VarDecision, and
   FcnObjective properties all point to the "current"
   constraint function block, variable block, and
   objective block. In this example, we only have one
   block of each.                                  */

Console.WriteLine("Optimize Status = " + status);

Console.WriteLine("x1 = " + varvalue[0]);
Console.WriteLine("x2 = " + varvalue[1]);
```

```
            Console.WriteLine("Obj = " + objValue[0]);
        }
    }

    catch(SolverException ex)
    {
        Console.WriteLine("Exception " + ex.ExceptionType);
        Console.WriteLine("Exception " + ex.Message);
    }
}
```

The Example Problem in Procedural C

```
void quadratic_example1()
{
    /* 2 decision variables,
       3 constraint functions:                                    */
    int nvars = 2, ncons = 3;
    double objcoeff[] = { -4, -2 };

    /* The matrix is passed row-wise, dense (sparse matrixes are
       difficult to create in Procedural C). Note that a Q matrix
       (QMatVal) is typically symmetrical, so it does not matter
       if you pass the coefficents row-wise or column-wise. We
       have only x1*x1 in the objective function, with coefficent
       1 in front, so in the QMatVal, we will have 1 in
       qmatval[0][0] and the rest will be 0.
                              x1   x2
                         x1    1    0
                         x2    0    0                            */

    double qmatval[] = { 1, 0,
                         0, 0, };

    // Note that the coefficients are passed row-wise:
    double concoeff[] = { 1, 1, 2,  1, 1, -4 };
    double rhsvalue[] = { 4, 5, -2 };

    double objval;
    double x[2];
    void * prob;

    // Creating a Problem:
```

```
SolverProblem(&prob);

// Setting maximizing objective:
SolverProbTypeSet(prob, Solver_Type_Minimize,
            Problem_Type_NA);

// Creating a variable block:
SolverVariableAdd(prob, L"Vars", Variable_Type_Decision,
            nvars);

// Creating a Constraint block:
SolverFunctionAdd(prob, L"Constraints",
            Function_Type_Constraint, ncons);

// Creating an Objective block:
SolverFunctionAdd(prob, L"Objective",
            Function_Type_Objective,  1);

/* Passing the right-hand sides of the constraints (There are
   ways two do it: we can either pass the lower and upper
   bounds on the functions, as we do here, or we can use the
   Relation method.):                                        */
SolverFcnUpperBoundSet(prob, L"Constraints", rhsvalue);

/* Varaibles have to be non-negative. Note that VarDecision
   points to the "current" block of variables. In this
   example we only have one block of variables, so pointing
   to all variables is OK:                                   */
SolverVarNonNegative(prob, L"Vars");

// The matrix is dense and passed row-wise:
SolverModArrayOrder(prob, Array_Order_ByRow);

/* Linear model; passing the coefficients for the objective
   function(s):                                              */
SolverModAllLinearSet(prob, Function_Type_Objective,
            Variable_Type_Decision , objcoeff);

/* Passing quadratic coefficient for the objective function
   (By calling this method, the problem is now automatically
   cahnged from linear to quadratic.):                       */
SolverModFcnQuadraticSet(prob, Function_Type_Objective,
            Variable_Type_Decision, 0, qmatval);
```

```
    /* Linear model; passing the coefficients for the constraint
       function(s):                                                */
    SolverModAllLinearSet(prob, Function_Type_Constraint,
                Variable_Type_Decision, concoeff);

    /* By calling the AllLinear functions, the problem type has
       been automatically updated to linear.                       */

    _Optimize_Status status;

    /* Solving the model (Note that we have not picked an Engine;
       the Solver will determine best Engine by checking the
       problem type; we get the status of the optimization; the
       main status values are typically "optimal", "infeasible",
       or "unbounded", but many others are possible, too):     */
    SolverOptimize(prob, &status, &objval);

    // Getting the final values of the variables:
    SolverVarFinalValue(prob, L"Vars", x);

    // Releasing the memory used by freeing the Problem:
    SolverProbFree(&prob);

    printf("quadratic_example\n");
    printf("x[0] = %g\n", x[0]);
    printf("x[1] = %g\n", x[1]);
    printf("Obj = %g\n", objval);
    printf("Status = %d\n", status);

    return;
}
```

⬇ Example 2 (Using Dense Matrix) ⬇

The Example Problem in Algebraic Notation

Minimize: $4x_1 * x_1 + 2x_1 * x_2 + 5x_1 * x_2 + 1.5x_1 - 2x_2$

subject to:
$$2x_1 - x_2 \leq -2$$
$$-x_1 + 2x_2 \leq 6$$
$$x_1 \leq 20$$

where $0 \leq x_1$
$0 \leq x_2$

(Solution: $x_1 = 0.762499$
$x_2 = 0.475002$
objective = 4.371875)

The Example Problem in VB.NET

```
Private Sub quadratic_example2()

   Dim nvars As Integer, ncons As Integer
   nvars = 2  '2 decision variables
   ncons = 3  '3 constraint functions

   Dim rhsValue(2) As Double
   rhsValue(0) = -2
   rhsValue(1) = 6
   rhsValue(2) = 20
   Dim fcnObj As New DoubleMatrix
   With fcnObj
      .Create(Rows:=1, Columns:=2)
      .Value(0, 0) = 1.5
      .Value(0, 1) = -2
   End With

   ' The matrix is passed row-wise, dense.(Note that a Q matrix
   ' (QMatVal) is typically symmetrical,  so it does not matter
   ' if you pass the coefficents row-wise or column-wise).

   Dim QMatVal As New DoubleMatrix
```

```
'        x1    x2
'   x1     4     1
'   x2     1     5

With QMatVal
   .Create(Rows:=2, Columns:=2)
   .Value(0, 0) = 4
   .Value(1, 0) = 1
   .Value(0, 1) = 1
   .Value(1, 1) = 5
End With

Dim fcnConstr As New DoubleMatrix
With fcnConstr
   .Create(Rows:=3, Columns:=2)
   .Value(0, 0) = -2
   .Value(0, 1) = -1
   .Value(1, 0) = -1
   .Value(1, 1) = 2
   .Value(2, 0) = 1
   .Value(2, 1) = 0
End With

Dim prob As SolverPlatform.Problem = Nothing

Try

   prob = New SolverPlatform.Problem     ' Creating a Problem.

   prob.VarDecision.Size = nvars         ' 2 variables.
   prob.FcnConstraint.Size = ncons       ' 3 constraints.

   ' Constraint right-hand sides:
   prob.FcnConstraint.UpperBound.Array = rhsValue

   prob.VarDecision.NonNegative()        ' Variables ≥ 0.

   ' Linear model; Passing the coefficients for the objective
   ' function(s):
   prob.Model.AllLinear(Function_Type.Objective,
           Variable_Type.Decision) = fcnObj
```

```
        ' Passing quadratic coefficient for  the  objective
        ' function; by calling this method,  the problem is
        ' automatically cahnged  from linear to quadratic:
        prob.Model.FcnQuadratic(Function_Type.Objective, 0,
                Variable_Type.Decision) = QMatVal


        ' Linear model; Passing the coefficients for the
        ' constraint function(s):
        prob.Model.AllLinear(Function_Type.Constraint,
                Variable_Type.Decision) = fcnConstr


        prob.Solver.SolverType = Solver_Type.Minimize


        prob.Solver.Optimize()              ' Solving the model.


        Dim status As Optimize_Status
        status = prob.Solver.OptimizeStatus ' Optimization status.


        Dim varvalue() As Double
        ' Final variable values:
        varvalue = prob.VarDecision.FinalValue.Arrays


        Dim objValue() As Double
        objValue = prob.FcnObjective.FinalValue.Array


        ' Note   that   the   FcnConstraint,   VarDecision,   and
        ' FcnObjective  properties  all  point  to  the  "current"
        ' constraint function block, variable block, and objective
        ' block. In this example, we only have one block of each.


        Console.WriteLine( "Optimize Status = " & status )


        Console.WriteLine( "x1 = " & varvalue(0) )
        Console.WriteLine( "x2 = " & varvalue(1) )
        Console.WriteLine( "Obj = " & objValue(0) )

    Catch ex As SolverException


        Console.WriteLine( ex.Message )
        Console.WriteLine( ex.ExceptionType )
    End Try
```

```
    If Not prob Is Nothing Then
        prob.Dispose()  ' Freeing up the license the right way.
    End If

    fcnObj = Nothing
    fcnConstr = Nothing

End Sub
```

The Example Problem in C++

```
int quadratic_example2()
{
    try
    {
        // 2 decision variables.
        // 3 constraint functions.
        int nvars = 2, ncons = 3;
        double objcoeff[] = { 1.5 , -2 };

        /* The matrix is passed row-wise, dense.(Note that a Q
           matrix (QMatVal) is typically symmetrical, so it does
           not matter if you pass the coefficents row-wise or
           column-wise).
                    x1   x2
              x1     4    1
              x2     1    5
*/

        double qmatval[] = { 4, 1,
                             1, 5, };

        //Passing the coefficients row-wise:
        double concoeff[] = { -2, -1, -1, 2, 1, 0 };
        double rhsvalue[] = { -2, 6, 20 };

        CProblem problem;

        problem.VarDecision.Size = nvars;       // 2 variables.
        problem.FcnConstraint.Size = ncons;     // 3 constraints.
```

```
problem.Solver.SolverType = Solver_Type_Minimize;

// Constraint right-hand sides:
problem.FcnConstraint.UpperBound = rhsvalue;

problem.VarDecision.NonNegative();    // Variables ≥ 0.

/* The Q matrix specifies the covariance between each pair
   of assets.                                              */

/* Linear model; Passing the coefficients for the
   objective function(s):                                  */
problem.Model.AllLinear[Function_Type_Objective] =
        CDoubleMatrix(Array_Order_ByRow, 1, nvars,
        objcoeff);

/* Linear model; Passing the coefficients for the
   constraint function(s):                                 */
problem.Model.AllLinear[Function_Type_Constraint] =
        CDoubleMatrix (Array_Order_ByRow, ncons, nvars,
        concoeff);
/* By calling the AllLinear functions, the problem is
   automatically cahnged from linear to quadratic.     */

problem.Model.FcnQuadratic[Function_Type_Objective][proble
        m.get_ObjectiveIndex()][Variable_Type_Decision]
          = CDoubleMatrix(Array_Order_ByRow, nvars,
        nvars, qmatval);

problem.Solver.Optimize();          // Solving the model.

// Optimization status:
_Optimize_Status status = problem.Solver.OptimizeStatus;

// Final variable values:
const double* varvalue = problem.VarDecision.FinalValue;

cout << "quadratic_example1" << endl;
cout << "Status = " << status << endl;
cout << "x1 = " << varvalue[0] << endl;
cout << "x2 = " << varvalue[1] << endl;
```

```
      cout << "Obj = " << problem.FcnObjective.FinalValue[0] <<
              endl;
      cout << endl;

    } catch(CSolverException e)
    {
      cout << "quadratic_example1" << endl;
      cout << "Throws Exception" << endl;
      cout << "Code = " << e.ErrorCode << endl;
      cout << "Text = " << ToCStr(e.Description).c_str() <<
              endl;
      return 1;
    }
    return 0;
}
```

The Example Problem in C#

```
private void quadratic_example2()
{
   try
   {
      int nvars = 2;            // 2 decision variables.
      int ncons = 3;            // 3 constraint functions.

      double [] rhsValue = { -2, 6, 20 };

      DoubleMatrix fcnObj = new DoubleMatrix(1, nvars);
      fcnObj[0,0] = 1.5;
      fcnObj[0,1] = -2;


      DoubleMatrix fcnConstr =  new DoubleMatrix(ncons,nvars);
      fcnConstr[0,0] = -2;
      fcnConstr[0,1] = -1;
      fcnConstr[1,0] = -1;
      fcnConstr[1,1] = 2;
      fcnConstr[2,0] = 1;
      fcnConstr[2,1] = 0;

      DoubleMatrix qmatval =  new DoubleMatrix(nvars,nvars);
```

```
/* The matrix is passed row-wise, dense.(Note that a Q
   matrix (QMatVal) is typically symmetrical, so it does
   not matter if you pass the coefficents row-wise or
   column-wise).
                x1   x2
          x1     4    1
          x2     1    5


qmatval[0,0] = 4;
qmatval[0,1] = 1;
qmatval[1,0] = 1;
qmatval[1,1] = 5;

// Creating a Problem:
using(Problem problem = new Problem())
{
   problem.VarDecision.Size = nvars;     // 2 variables.
   problem.FcnConstraint.Size = ncons;   // 3 constraints.

   // Constraint right-hand sides:
   problem.FcnConstraint.UpperBound.Array = rhsValue;
   problem.VarDecision.NonNegative();    // Variables ≥ 0.

   /* Linear model; Passing the coefficients for the
      objective function(s):                          */
   problem.Model.AllLinear[Function_Type.Objective] =
        fcnObj;

   /* Passing quadratic coefficient for the objective
      function (By calling the AllLinear functions, the
      problem is automatically cahnged  from linear to
      quadratic.):                                    */
   problem.Model.FcnQuadratic[Function_Type.Objective,
        problem.ObjectiveIndex]   = qmatval;

   /* Linear model; Passing the coefficients for the
      constraint function(s):                         */
   problem.Model.AllLinear[Function_Type.Constraint] =
        fcnConstr;

   problem.Solver.SolverType = Solver_Type.Minimize;
```

```
            problem.Solver.Optimize();          // Solving the model.

            // Optimization status:
            Optimize_Status status = problem.Solver.OptimizeStatus;

            // Final variable values:
            double [] varvalue =
                  problem.VarDecision.FinalValue.Array;

            double [] objValue =
                  problem.FcnObjective.FinalValue.Array;

            /* Note that the FcnConstraint, VarDecision, and
               FcnObjective properties all point to the "current"
               constraint function block, variable block, and
               objective block. In this example, we only have one
               block of each.                                    */

            Console.WriteLine("Optimize Status = " + status);

            Console.WriteLine("x1 = " + varvalue[0]);
            Console.WriteLine("x2 = " + varvalue[1]);

            Console.WriteLine("Obj = " + objValue[0]);
         }
      }

      catch(SolverException ex)
      {
         Console.WriteLine("Exception " + ex.ExceptionType);
         Console.WriteLine("Exception " + ex.Message);
      }
   }
```

The Example Problem in Procedural C

```
void quadratic_example2()
{
   //2 decision variables, 3 constraint functions:
   int nvars = 2, ncons = 3;
   double objcoeff[] = { 1.5, -2 };
```

```
/* The matrix is passed row-wise, dense.(Note that a Q matrix
   (QMatVal) is typically symmetrical, so it does not matter
   if you pass the coefficents row-wise or column-wise).
                x1   x2
          x1     4    1
          x2     1    5
   */

double qmatval[] = { 4, 1,
                     1, 5, };
//Passing the coefficients row-wise:
double concoeff[] = { -2, -1, -1, 2, 1, 0 };
double rhsvalue[] = { -2, 6, 20 };

double objval;
double x[2];
void * prob;

// Creating a Problem:
SolverProblem(&prob);

// Setting maximizing objective:
SolverProbTypeSet(prob, Solver_Type_Minimize,
           Problem_Type_NA);
// Creating a variable block:
SolverVariableAdd(prob, L"Vars", Variable_Type_Decision,
           nvars);
// Creating a Constraint block:
SolverFunctionAdd(prob, L"Constraints",
           Function_Type_Constraint, ncons);
// Creating an Objective block:
SolverFunctionAdd(prob, L"Objective",
           Function_Type_Objective,  1);

/* Passing the right-hand sides of the constraints (There are
   two ways to do it: by either passing the lower and upper
   bounds on the functions, as we do here, or by using use
   the Relation method.):                                  */
SolverFcnUpperBoundSet(prob, L"Constraints", rhsvalue);

/* Varaibles have to be non-negative. Note that the
   FcnConstraint, VarDecision, and FcnObjective properties
   all point to the "current" constraint function block,
```

```
   variable block, and objective block. In this example, we
   only have one block of each.                                */
SolverVarNonNegative(prob, L"Vars");

/* Note that we use a dense matrix, and pass the elements by
   row:                                                        */
SolverModArrayOrder(prob, Array_Order_ByRow);

/* Linear model; Passing the coefficients for the objective
   function(s):                                                */
SolverModAllLinearSet(prob, Function_Type_Objective,
           Variable_Type_Decision , objcoeff);

// Passing quadratic coefficient for the objective function:
SolverModFcnQuadraticSet(prob, Function_Type_Objective,
           Variable_Type_Decision, 0, qmatval);

/* Linear model; Passing the coefficients for the constraint
   function(s):                                                */
SolverModAllLinearSet(prob, Function_Type_Constraint,
           Variable_Type_Decision, concoeff);
/* By calling the AllLinear functions, the problem is
   automatically cahnged from linear to quadratic.             */

_Optimize_Status status;

/* Now we solve the model. Note that we do not specify an
   engine; the Solver will determine best Engine by checking
   the problem type. We get the status of the optimization.
   The main status values are typically "optimal",
   "infeasible", or "unbounded". But many others are
   possible, too.                                              */

SolverOptimize(prob, &status, &objval);

// Retrieving final values of the variables:
SolverVarFinalValue(prob, L"Vars", x);

// Releasing the memory used, by freeing the Problem:
SolverProbFree(&prob);

printf("quadratic_example1\n");
printf("x[0] = %g\n", x[0]);
```

```
    printf("x[1] = %g\n", x[1]);
    printf("Obj = %g\n", objval);
    printf("Status = %d\n", status);

    return;
}
```

Nonlinear Problems

Definition

Nonlinear Programs/Problems (NLP) are programs in which at least one of the functions (objective or constraint) is not affine on X. (Usually, it is the rule of the function that classifies it as nonlinear. In particular, a linear integer program is not generally considered an NLP.) Mathematically, NLPs look like this:

Minimize $f(x)$
subject to: $l \leq g(x) \leq u$
$lb \leq x \leq ub$
where $f(x)$ and/or $g(x)$ are smooth nonlinear functions.

Examples >

Nonlinear Problem Examples

The Example Problem in Algebraic Notation

Minimize: $-4x_1^3 + 3x_1 - 6x_2$
subject to: $x_1 + x_2 \leq 4$
$2x_1 + x_2 \leq 5$
$x_1 - 4x_2 \leq -2$

(Solution: $x_1 = 2$
$x_2 = 1$
objective = -32)

The Example Problem in VB.NET

```
Private Function NLPEvaluator1_OnEvaluate(ByVal evaluator As
                SolverPlatform.Evaluator) As
                SolverPlatform.Engine_Action Handles
                NLPEvaluator1.OnEvaluate
' This Evaluator computes the values of the objective  and
' constraint functions for a given set of variable values.

   Dim prob As SolverPlatform.Problem
   prob = Evaluator.Problem

   Dim Var() As Double
```

```
    ' Retrieving the current variable values:
    Var = prob.VarDecision.Value.Array

    ' Storing the computed function values in an array, Fcn:
    Dim Fcn(2) As Double

    Fcn(0) = Var(0) + Var(1)
    Fcn(1) = 2 * Var(0) + Var(1)
    Fcn(2) = Var(0) - 4 * Var(1)


    Dim obj As Double

    ' Computing the objective:
    obj = -4 * Var(0) * Var(0) * Var(0) + 3 * Var(0) - 6 * Var(1)

    ' Passing the function values to the Solver:
    prob.FcnConstraint.Value.Array = Fcn

    ' Passing the objective value to the Solver:
    prob.FcnObjective.Value(prob.ObjectiveIndex) = obj

    NLPEvaluator1_OnEvaluate = Engine_Action.Continue

End Function

Private Sub nonlinear_example()

    NLPEvaluator1 = New SolverPlatform.Evaluator
    EvalIterator = New SolverPlatform.Evaluator

    Dim nvars As Integer, ncons As Integer
    nvars = 2                           '2 decision variables.
    ncons = 3                           '3 constraint functions.

    Dim rhsValue(2) As Double
    rhsValue(0) = 4
    rhsValue(1) = 5
    rhsValue(2) = -2
```

```
Dim rhs(1) As Double
rhs(0) = 5
rhs(1) = 5

Dim prob As SolverPlatform.Problem = Nothing

Try

   prob = New SolverPlatform.Problem  ' Creating a Problem:

   prob.VarDecision.Size = nvars      ' 2 variables.
   prob.FcnConstraint.Size = ncons    ' 3 constraints.

   ' Constraint right-hand sides:
   prob.FcnConstraint.UpperBound.Array = rhsValue

   prob.VarDecision.NonNegative()     ' Variables ≥ 0.

   prob.VarDecision.UpperBound.Array = rhs

   prob.Solver.SolverType = Solver_Type.Minimize

   prob.ProblemType = Problem_Type.OptNLP

   prob.Engine = prob.Engines.Item("GRG Nonlinear")

   prob.Engine.Params("MultiStart").Value = 1

   ' Setting the Evaluator:
   prob.Evaluators.Item(Eval_Type.Function) = NLPEvaluator1()

   prob.Solver.Optimize()              ' Solving the model.

   Dim convalue() As Double

   ' Final constraint values:
   convalue = prob.FcnConstraint.FinalValue.Array

   Dim varvalue() As Double
```

```
        ' Final variable values:
        varvalue = prob.VarDecision.FinalValue.Array

        Dim objValue As Double

        ' Final objective value:
        objValue = prob.FcnObjective.FinalValue(0)

        Dim status As Optimize_Status
        status = prob.Solver.OptimizeStatus ' Optimization status.

        Console.WriteLine( "Optimize Status = " & status )
        Console.WriteLine( "x1 = " & varvalue(0) )
        Console.WriteLine( "x2 = " & varvalue(1) )
        Console.WriteLine( "Obj = " & objValue )

    Catch ex As SolverException

        Console.WriteLine( ex.Message )
        Console.WriteLine( ex.ExceptionType )

    End Try

    If Not prob Is Nothing Then
        prob.Dispose()      ' Freeing up the license the right way.
    End If

    NLPEvaluator1 = Nothing

    prob = Nothing

    End Sub
```

The Example Problem in C++

```
class CNLPEvaluator1 : public IEvaluator
{
/* This Evaluator computes the values of the objective and
   constraint functions for a given set of variable values.   */
public:
```

```
   virtual _Engine_Action Evaluate(CEvaluator& evaluator)
   {
      // Retrieving the current variable values:
      const double* pVar = evaluator.Problem.VarDecision.Value;

      cout << "Evaluation x1 = " << pVar[0] << ", x2 = " <<
              pVar[1] << ", Iteration = " <<
         evaluator.Problem.Engine.Stat.FunctionEvals<< endl;

      /* Geting a convenient pointer to the constraint
         functions:                                        */
      double* pFcn = evaluator.Problem.FcnConstraint.Value;

      pFcn[0] = pVar[0] + pVar[1];      // x1 + x2
      pFcn[1] = 2*pVar[0] + pVar[1];    // 2x1 +x2
      pFcn[2] = pVar[0] - 4*pVar[1];    // x1 - 4x2
      // Computing the objective (-4x13 + 3x1 - 6x2) :
      evaluator.Problem.FcnObjective.Value[evaluator.Problem.get
              _ObjectiveIndex()] =  -4*pVar[0] * pVar[0]*
              pVar[0] + 3 * pVar[0] - 6* pVar[1];

      return Engine_Action_Continue;
   }
};

int nonlinear_example()
{
   try
   {
      int nvars = 2, ncons = 3;

      double rhsvalue[] = { 4, 5, -2 };

      double rhs[] = { 5, 5 };

      //Creating a Problem:
      CProblem problem(Solver_Type_Minimize, nvars, ncons);

      // Constraint right-hand sides:
      problem.FcnConstraint.UpperBound = rhsvalue;
```

```
        problem.VarDecision.UpperBound = rhs;

        problem.VarDecision.NonNegative();       // Variables ≥ 0.

        problem.ProblemType = Problem_Type_OptNLP;
        problem.Engine = problem.Engines[CEngine::GRGName];
        problem.Engine.Params[L"MultiStart"].Value = 1;

        CNLPEvaluator1 evalNLP;          // Creating an Evaluator.

        /* Passing the Evaluator (Note that this Evaluator is
           required.):                                          */
        problem.Evaluators[Eval_Type_Function].Evaluator = &
                evalNLP;

        problem.Solver.Optimize();
        _Optimize_Status status = problem.Solver.OptimizeStatus;

        const double* convalue = problem.FcnConstraint.FinalValue;
        const double* varvalue = problem.VarDecision.FinalValue;

        cout << "nonlinear_example" << endl;
        cout << "Status = " << status << endl;
        cout << "x1 = " << varvalue[0] << endl;
        cout << "x2 = " << varvalue[1] << endl;
        cout << "Obj = " << problem.FcnObjective.FinalValue[0] <<
                endl;
        cout << endl;

    } catch(CSolverException e)
    {
        cout << "nonlinear_example" << endl;
        cout << "Throws Exception" << endl;
        cout << "Code = " << e.ErrorCode << endl;
        cout << "Text = " << ToCStr(e.Description).c_str() <<
                endl;
        return 1;
    }
    return 0;
}
```

The Example Problem in C#

```
private Engine_Action NLPEvaluator1_Evaluate(Evaluator
            evaluator)
{
   Console.WriteLine("Evaluation x1 = " +
            evaluator.Problem.VarDecision.Value[0] +
      ", x2 = " + evaluator.Problem.VarDecision.Value[1] + ",
            Eval = " +
      evaluator.Problem.Engine.Stat.FunctionEvals);

   evaluator.Problem.FcnConstraint.Value[0] =
      evaluator.Problem.VarDecision.Value[0] +
      evaluator.Problem.VarDecision.Value[1];

   evaluator.Problem.FcnConstraint.Value[1] =
      2*evaluator.Problem.VarDecision.Value[0] +
       evaluator.Problem.VarDecision.Value[1];

   evaluator.Problem.FcnConstraint.Value[2] =
      evaluator.Problem.VarDecision.Value[0] -
      4*evaluator.Problem.VarDecision.Value[1];


   evaluator.Problem.FcnObjective.Value[evaluator.Problem.Object
            iveIndex] =
      -4*evaluator.Problem.VarDecision.Value[0] *
      evaluator.Problem.VarDecision.Value[0] *
            evaluator.Problem.VarDecision.Value[0] +
      3*evaluator.Problem.VarDecision.Value[0] -
      6*evaluator.Problem.VarDecision.Value[1];

   return Engine_Action.Continue;
}

private void nonlinear_example()
{
   try
   {
      int nvars = 2, ncons = 3;

      double [] rhsValue = { 4, 5, -2 };
```

```
        double [] rhs = { 5, 5 };

        using(Problem problem = new Problem(Solver_Type.Minimize,
                nvars, ncons))
        {
            problem.FcnConstraint.UpperBound.Array  = rhsValue;

            problem.VarDecision.NonNegative();
            problem.VarDecision.UpperBound.Array = rhs;

            problem.ProblemType = Problem_Type.OptNLP;

            problem.Engine = problem.Engines[Engine.GRGName];

            problem.Engine.Params["MultiStart"].Value = 1;

            problem.Evaluators[Eval_Type.Function].OnEvaluate +=
                  new EvaluateEventHandler(NLPEvaluator1_Evaluate);

            problem.Solver.Optimize();

            Optimize_Status status = problem.Solver.OptimizeStatus;

            Console.WriteLine("\r\nnonlinear_example");
            Console.WriteLine("Status = " + status);
            Console.WriteLine("x1 = " +
                  problem.VarDecision.FinalValue[0]);
            Console.WriteLine("x2 = " +
                  problem.VarDecision.FinalValue[1]);
            Console.WriteLine("Obj = " +
                  problem.FcnObjective.FinalValue[0]);
        }
    }
    catch(SolverException ex)
    {
        Console.WriteLine("Exception " + ex.ExceptionType);
        Console.WriteLine("Exception " + ex.Message);
    }
}
```

The Example Problem in Procedural C

```
_Engine_Action Evaluate1(void* prob, void* refuser, _Eval_Type
               evaltype )
{

   /* This Evaluator computes the values of the objective and
      constraint functions for a given set of variable values.*/

   double var[2];
   double fcn[3];
   double obj[1];

   /* Retrieving the current variable values and storing the
      computed function values in an array, Fcn:              */
   SolverVarValue(prob, L"Vars", var);
   fcn[0] = var[0] + var[1];
   fcn[1] = 2*var[0] + var[1];
   fcn[2] = var[0] - 4*var[1];

   // Computing the objective:
   obj[0] = -4*var[0]*var[0]*var[0] + 3*var[0]-6*var[1];

   // Passing the function values to the Solver:
   SolverFcnValueSet(prob, L"Constraints", fcn);

   // Passing the objective value to the Solver:
   SolverFcnValueSet(prob, L"Objective", obj);
   return Engine_Action_Continue;
}

void nonlinear_example()
{
   void * prob;
   _Optimize_Status status;
   double objval;
   double rhsvalue[] = {4, 5, -2};
   double rhs[] = { 5, 5 };
   double x[2];

   SolverProblem(&prob);
```

```
   SolverProbTypeSet(prob,
              Solver_Type_Minimize,Problem_Type_OptNLP);
   SolverVariableAdd(prob, L"Vars", Variable_Type_Decision, 2);
   SolverFunctionAdd(prob, L"Constraints",
              Function_Type_Constraint, 3);
   SolverFunctionAdd(prob, L"Objective",
              Function_Type_Objective, 1);


   /* Passing the right-hand sides of the constraints (There are
      two ways to do it: we can pass the lower and upper bounds
      on the functions, as we do here, or we can use the
      Relation method.):                                        */
   SolverFcnUpperBoundSet(prob, L"Constraints", rhsvalue);


   /* Varaibles have to be non-negative. Note that VarDecision
      points to the "current" block of variables. In this
      example we only have one block of variables, so pointing
      to all variables is OK:                                   */
   SolverVarNonNegative(prob, L"Vars");
   SolverVarUpperBoundSet(prob, L"Vars", rhs);


   SolverProbEngineSet(prob, L"GRG Nonlinear");


   SolverEngParamSet(prob, L"Multistart", 1);


   SolverEvaluatorAdd(prob, Evaluate1, NULL,
              Eval_Type_Function);


   SolverOptimize(prob, &status, &objval);
   SolverVarFinalValue(prob, L"Vars", x);


   SolverProbFree(&prob);


   printf("nonlinear_example\n");
   printf("x[0] = %g\n", x[0]);
   printf("x[1] = %g\n", x[1]);
   printf("Obj = %g\n", objval);
   printf("Status = %d\n", status);
}
```

Quadratically Constrained Problems

Definition

Quadratically Constrained Programs (or “Problems”) – "QCP" for short – represent a special case of the NLP in which all the nonlinearities are required to be quadratic. Mathematically, QCPs look like this:

$$\begin{aligned} \text{Minimize} \quad & xQ1x + cx \\ \text{subject to:} \quad & l1 \le Ax \le u1 \\ & l2 \le xQ2x \le u2 \\ & lb \le x \le ub \end{aligned}$$

Examples >

Quadratically Constrained Problem Examples

The Example Problem in Algebraic Notation

$$\begin{aligned} \text{Minimize:} \quad & x_1 * x_1 + x_2 * x_{2+} - 2x_1 \\ \text{subject to:} \quad & x_1 * x_1 + 6x_2 - 36 \le 0 \\ & x_2 * x_2 - 4x_1 - 2x_2 \le 0 \\ \text{where} \quad & 0 \le x_1 \\ & 0 \le x_2 \end{aligned}$$

(Solution: $x_1 = 1$
$x_2 = 0$
objective = -1)

The Example Problem in VB.NET

```
Private Sub quadratically_constrained()

   Dim nvars As Integer, ncons As Integer
   nvars = 2        ' 2 decision variables.
   ncons = 2        ' 2 constraint functions.

   ' In the constraint "x1*x1 + 6x2 - 36 ≤ 0", we move 36 to the
   ' right-hand  side and the right-hand side value becomes 36.
   Dim rhsValue(2) As Double
```

```
rhsValue(0) = 36
rhsValue(1) = 0

Dim fcnObj As New DoubleMatrix
With fcnObj
   .Create(Rows:=1, Columns:=2)
   .Value(0, 0) = -2
   .Value(0, 1) = 0
End With

' The matrix is passed row-wise, dense.(Note that a Q matrix
' (QMatVal) is typically symmetrical,  so it does not matter
' if we pass the coefficents row-wise or column-wise.)

Dim QMatVal1 As New DoubleMatrix

'       x1   x2
' x1     1    0
' x2     0    1

With QMatVal1
   .Create(Rows:=2, Columns:=2)
   .Value(0, 0) = 1
   .Value(1, 0) = 0
   .Value(0, 1) = 0
   .Value(1, 1) = 1
End With

Dim fcnConstr As New DoubleMatrix
With fcnConstr
   .Create(Rows:=2, Columns:=2)
   .Value(0, 0) = 0
   .Value(0, 1) = 6
   .Value(1, 0) = -4
   .Value(1, 1) = -2
End With

Dim QMatVal2 As New DoubleMatrix
```

```
'       x1   x2
' x1     1    0
' x2     0    0

With QMatVal2
   .Create(Rows:=2, Columns:=2)
   .Value(0, 0) = 1
   .Value(1, 0) = 0
   .Value(0, 1) = 0
   .Value(1, 1) = 0
End With

Dim QMatVal3 As New DoubleMatrix

'       x1   x2
' x1     0    0
' x2     0    1

With QMatVal3
   .Create(Rows:=2, Columns:=2)
   .Value(0, 0) = 0
   .Value(1, 0) = 0
   .Value(0, 1) = 0
   .Value(1, 1) = 1
End With

Dim prob As SolverPlatform.Problem = Nothing

Try

   prob = New SolverPlatform.Problem    ' Creating a Problem.

   prob.VarDecision.Size = nvars        ' 2 variables.
   prob.FcnConstraint.Size = ncons      ' 2 constraints.

   ' Constraint right-hand sides:
   prob.FcnConstraint.UpperBound.Array = rhsValue

   prob.VarDecision.NonNegative()       ' Variables ≥ 0
```

```
' Linear model; Passing the coefficients for the objective
' function(s):
prob.Model.AllLinear(Function_Type.Objective,
        Variable_Type.Decision) = fcnObj


' Passing quadratic coefficient for  the  objective
' function; by calling this method,  the problem is
' automatically cahnged from linear to quadratic:
prob.Model.FcnQuadratic(Function_Type.Objective, 0,
        Variable_Type.Decision) = QMatVal


' Linear model; Passing the coefficients for the
' constraint function(s):
prob.Model.AllLinear(Function_Type.Constraint,
        Variable_Type.Decision) = fcnConstr


' Passing quadratic coefficient for the first constraint
' function:
prob.Model.FcnQuadratic(Function_Type.Constraint, 0,
        Variable_Type.Decision) = QMatVal2


' Passing quadratic coefficient for the second constraint
' function
prob.Model.FcnQuadratic(Function_Type.Constraint, 1,
        Variable_Type.Decision) = QMatVal3


prob.Solver.SolverType = Solver_Type.Minimize


' Specifying the problem type:
prob.ProblemType = Problem_Type.OptQCP


Dim engine As Engine = Nothing
' Specifying the Engine to solve the problem (A call to an
' external  DLL  is  made  because the Engine required for
' solving this type of problem - the MOSEK Engine - is not
' available internally.):
engine = New Engine("name",
        "C:\\Platform\\Src\\MOSEKeng.dll")
prob.Engine = engine


prob.Solver.Optimize()              ' Solving the model.


Dim status As Optimize_Status
status = prob.Solver.OptimizeStatus ' Optimization status.
```

```
        Dim varvalue() As Double

       ' Final variable values:
        varvalue = prob.VarDecision.FinalValue.Array

        Dim objValue() As Double
        objValue = prob.FcnObjective.FinalValue.Array

        ' Note that the FcnConstraint, VarDecision, and
        ' FcnObjective properties all point to the "current"
        ' constraint function block, variable block, and objective
        ' block. In this example, we only have one block of each.

        Console.WriteLine( "Optimize Status = " & status )
        Console.WriteLine( "x1 = " & varvalue(0) )
        Console.WriteLine( "x2 = " & varvalue(1) )
        Console.WriteLine( "Obj = " & objValue(0) )
    Catch ex As SolverException

        Console.WriteLine( ex.Message )
        Console.WriteLine( ex.ExceptionType )
    End Try

    If Not prob Is Nothing Then
        prob.Dispose()      ' Freeing up the license the right way.
    End If

    fcnObj = Nothing
    fcnConstr = Nothing

End Sub
```

The Example Problem in C++

```
int quadratically_constrained()
{
    try
    {
        /* 2 decision variables,
           2 constraint functions:                                */
```

```
      int nvars = 2, ncons = 2;
      double objcoeff[] = { -2, 0 };

      /* The matrix is passed row-wise, dense. (Note that a Q
         matrix (QMatVal) is typically symmetrical, so it does
         not matter if you pass the coefficents row-wise or
         column-wise).
                          x1  x2
                     x1    1   0
                     x2    0   1
         */

      double qmatval1[] = { 1, 0,
                            0, 1, };

      // Note that the coefficients are passed row-wise:
      double concoeff[] = { 0, 6, -4, -2 };

      /*          x1  x2
            x1     1   0
            x2     0   0
*/

      double qmatval2[] = { 1, 0,
                            0, 0, };

      /*          x1  x2
            x1     0   0
            x2     0   1
*/

      double qmatval3[] = { 0, 0,
                           0, 1, };

      /* In the constraint "x1*x1 + 6 x2 - 36 ≤ 0", we move 36
         to the right-hand side and the right-hand side value
         becomes 36:                                          */
      double rhsvalue[] = { 36, 0 };

      CProblem problem;

      problem.VarDecision.Size = nvars;    // 2 variables.
      problem.FcnConstraint.Size = ncons;  // 2 constraints.
```

```
problem.Solver.SolverType = Solver_Type_Minimize;

// Constraint right-hand sides:
problem.FcnConstraint.UpperBound = rhsvalue;

problem.VarDecision.NonNegative();   // Variables ≥ 0.

/* The Q matrix specifies the covariance between each pair
   of assets                                              */

/* Linear model; Passing the coefficients for the
   objective function(s):                                 */
problem.Model.AllLinear[Function_Type_Objective] =
        CDoubleMatrix(Array_Order_ByRow, 1, nvars,
        objcoeff);

/* Linear model; Passing the coefficients for the
   constraint function(s):                                */
problem.Model.AllLinear[Function_Type_Constraint] =
        CDoubleMatrix (Array_Order_ByRow, ncons, nvars,
        concoeff);

/* By calling the AllLinear functions, the Problem Type
   has been automatically updated to linear.              */

/* Passing quadratic coefficient for the objective
   function; by calling this method, the problem is now
   automatically cahnged from linear to quadratic:        */
problem.Model.FcnQuadratic[Function_Type_Objective][proble
        m.get_ObjectiveIndex()][Variable_Type_Decision] =
        CDoubleMatrix(Array_Order_ByRow, nvars, nvars,
        qmatval1);

/* Passing quadratic coefficient for the first constraint
   function:                                              */
problem.Model.FcnQuadratic[Function_Type_Constraint][0][Va
        riable_Type_Decision]   =
        CDoubleMatrix(Array_Order_ByRow, nvars, nvars,
        qmatval2);

/* Passing quadratic coefficient for the second constraint
   function:                                              */
problem.Model.FcnQuadratic[Function_Type_Constraint][1][Va
        riable_Type_Decision]   =
        CDoubleMatrix(Array_Order_ByRow, nvars, nvars,
        qmatval3);
```

```
      // Specifying the Problem Type:
      problem.ProblemType = Problem_Type_OptQCP;

      /* Specifying the Engine to solve the problem (A call to
         an external DLL is made because the Engine required for
         solving this type of problem - the MOSEK Engine - is
         not available internally.):                          */
      CEngine engine( L"name",
              L"C:\\Platform\\Src\\MOSEKeng.dll" );
      problem.Engine = engine;
      problem.Solver.Optimize();           // Solving the model.

      // Optimization status:
      _Optimize_Status status = problem.Solver.OptimizeStatus;

      // Final variable values:
      const double* varvalue = problem.VarDecision.FinalValue;

      cout << "quadratically_constrained" << endl;
      cout << "Status = " << status << endl;
      cout << "x1 = " << varvalue[0] << endl;
      cout << "x2 = " << varvalue[1] << endl;

      cout << "Obj = " << problem.FcnObjective.FinalValue[0] <<
              endl;
      cout << endl;

   } catch(CSolverException e)
   {
      cout << "quadratically_constrained" << endl;
      cout << "Throws Exception" << endl;
      cout << "Code = " << e.ErrorCode << endl;
      cout << "Text = " << ToCStr(e.Description).c_str() <<
              endl;
      return 1;
   }
   return 0;
}
```

The Example Problem in C#

```
private void quadratically_constrained()
{
   try
   {
      int nvars = 2;          // 2 decision variables.
      int ncons = 2;          // 2 constraint functions.

      /* In the constraint "x1*x1 + 6x2 - 36 ≤ 0", we move 36 to
         the right-hand side and the right-hand side value
         becomes 36:                                          */
      double [] rhsValue = { 36, 0 };

      DoubleMatrix fcnObj = new DoubleMatrix(1, nvars);
      fcnObj[0,0] = -2;
      fcnObj[0,1] = 0;


      DoubleMatrix fcnConstr =  new DoubleMatrix(ncons,nvars);
      fcnConstr[0,0] = 0;
      fcnConstr[0,1] = 6;
      fcnConstr[1,0] = -4;
      fcnConstr[1,1] = -2;

      DoubleMatrix qmatval1 =  new DoubleMatrix(nvars,nvars);

      /* The matrix is passed row-wise, dense. (Note that a Q
         matrix (QMatVal) is typically symmetrical, so it does
         not matter if you pass the coefficents row-wise or
         column-wise).
                x1  x2
            x1   1   0
            x2   0   1
*/

      qmatval1[0,0] = 1;
      qmatval1[0,1] = 0;
      qmatval1[1,0] = 0;
      qmatval1[1,1] = 1;

      DoubleMatrix qmatval2 =  new DoubleMatrix(nvars,nvars);
```

```
      /*          x1   x2
            x1     1    0
            x2     0    0
*/

      qmatval2[0,0] = 1;
      qmatval2[0,1] = 0;
      qmatval2[1,0] = 0;
      qmatval2[1,1] = 0;

      DoubleMatrix qmatval3 =  new DoubleMatrix(nvars,nvars);

      /*          x1   x2
            x1     0    0
            x2     0    1
*/

      qmatval3[0,0] = 0;
      qmatval3[0,1] = 0;
      qmatval3[1,0] = 0;
      qmatval3[1,1] = 1;

      // Creating a Problem:
      using(Problem problem = new Problem())
      {
         problem.VarDecision.Size = nvars;
         problem.FcnConstraint.Size = ncons;

         problem.FcnConstraint.UpperBound.Array = rhsValue;
         problem.VarDecision.NonNegative();

         /* Linear model; Passing the coefficients for the
            objective function(s):                          */
         problem.Model.AllLinear[Function_Type.Objective] =
               fcnObj;

         /* Passing quadratic coefficient for the objective
            function; by calling this method, the problem is now
            automatically cahnged from linear to quadratic:   */
         problem.Model.FcnQuadratic[Function_Type.Objective,
               problem.ObjectiveIndex]   = qmatval1;
```

```
/* Linear model; Passing the coefficients for the
   constraint function(s):                          */
problem.Model.AllLinear[Function_Type.Constraint] =
     fcnConstr;

/* Passing quadratic coefficient for the first
   constraint function                              */
problem.Model.FcnQuadratic[Function_Type.Constraint,
     0]   = qmatval2;

/* Passing quadratic coefficient for the second
   constraint function:                             */
problem.Model.FcnQuadratic[Function_Type.Constraint,
     1]   = qmatval3;

problem.Solver.SolverType = Solver_Type.Minimize;

// Specifying the Problem Type:
problem.ProblemType = Problem_Type.OptQCP;

/* Specifying the Engine to solve the problem (A call
   to an external DLL is made because the Engine
   required for solving this type of problem - the
   MOSEK Engine - is not available internally.):    */
Engine engine = new Engine ( "name",
     "C:\\Platform\\Src\\MOSEKeng.dll" );
problem.Engine = engine;

problem.Solver.Optimize();        // Solving the model.

// Optimization status:
Optimize_Status status = problem.Solver.OptimizeStatus;

// Final variable values:
double [] varvalue =
     problem.VarDecision.FinalValue.Array;

double [] objValue =
     problem.FcnObjective.FinalValue.Array;

/* Note that the FcnConstraint, VarDecision, and
   FcnObjective properties all point to the "current"
   constraint function block, variable block, and
```

```
            objective block. In this example, we only have one
            block of each.                                          */

            Console.WriteLine("Optimize Status = " + status);

            Console.WriteLine("x1 = " + varvalue[0]);
            Console.WriteLine("x2 = " + varvalue[1]);

            Console.WriteLine("Obj = " + objValue[0]);
        }
    }

    catch(SolverException ex)
    {
        Console.WriteLine("Exception " + ex.ExceptionType);
        Console.WriteLine("Exception " + ex.Message);
    }
}
```

The Example Problem in Procedural C

```
void quadratically_constrained()
{
      /* 2 decision variables,
         2 constraint functions:                                   */
    int nvars = 2, ncons = 2;
    double objcoeff[] = { -2, 0 };

    /* The matrix is passed row-wise, dense. (Note that a Q
       matrix (QMatVal) is typically symmetrical, so it does not
       matter if you pass the coefficents row-wise or column-
       wise).
                  x1   x2
            x1     1    0
            x2     0    1
*/

    double qmatval1[] = { 1, 0,
                          0, 1, };
    // Note that the coefficients are passed row-wise:
    double concoeff[] = { 0, 6, -4, -2 };
```

```
   /*            x1  x2
            x1    1   0
            x2    0   0
*/

   double qmatval2[] = { 1, 0,
                         0, 0, };

   /*            x1  x2
            x1    0   0
            x2    0   1
*/

   double qmatval3[] = { 0, 0,
                         0, 1, };

   /* In the constraint "x1* x1 + 6x2 - 36 ≤ 0", we move 36 to
      the right-hand side and the right-hand side value becomes
      36:                                                      */
   double rhsvalue[] = { 36, 0 };
   double objval;
   double x[2];
   void * prob;

   // Creating a problem:
   SolverProblem(&prob);

   // Setting maximizing objective:
   SolverProbTypeSet(prob, Solver_Type_Minimize,
               Problem_Type_OptQCP);

   // Creating a variable block:
   SolverVariableAdd(prob, L"Vars", Variable_Type_Decision,
               nvars);

   // Creating a constraint block:
   SolverFunctionAdd(prob, L"Constraints",
               Function_Type_Constraint, ncons);

   // Creating an objective block:
   SolverFunctionAdd(prob, L"Objective",
               Function_Type_Objective,  1);
```

```
/* Passing the right-hand sides of the constraints (There are
   ways two do it: we can either pass the lower and upper
   bounds on the functions, as we do here, or we can use the
   Relation method.):                                        */
SolverFcnUpperBoundSet(prob, L"Constraints", rhsvalue);

/* Varaibles have to be non-negative. Note that VarDecision
   points to the "current" block of variables. In this
   example we only have one block of variables, so pointing
   to all variables is OK:                                   */
SolverVarNonNegative(prob, L"Vars");

// The matrix is dense and passed row-wise:
SolverModArrayOrder(prob, Array_Order_ByRow);

/* Linear model; passing the coefficients for the objective
   function(s):                                              */
SolverModAllLinearSet(prob, Function_Type_Objective,
           Variable_Type_Decision , objcoeff);

// Passing quadratic coefficient for the objective function:
SolverModFcnQuadraticSet(prob, Function_Type_Objective,
           Variable_Type_Decision, 0, qmatval1);

/* Linear model; passing the coefficients for the constraint
   function(s):                                              */
SolverModAllLinearSet(prob, Function_Type_Constraint,
           Variable_Type_Decision, concoeff);

/* By calling the AllLinear functions, the problem type has
   been automatically updated to linear.                     */

SolverModFcnQuadraticSet(prob, Function_Type_Constraint,
           Variable_Type_Decision, 0, qmatval2);

SolverModFcnQuadraticSet(prob, Function_Type_Constraint,
           Variable_Type_Decision, 1, qmatval3);

/* Specifying the Engine to solve the problem (A call to an
   external DLL is made because the Engine required for
   solving this type of problem - the MOSEK Engine - is not
   available internally.):                                   */
SolverEngineAdd(prob,L"MyEngine",L"C:\\Platform\\Src\\MOSEKEn
           g.dll");
```

```
    _Optimize_Status status;

    /* Solving the model (Note that we have not picked an Engine;
       the Solver will determine best Engine by checking the
       problem type; we get the status of the optimization; the
       main status values are typically "optimal", "infeasible",
       or "unbounded", but many others are possible, too):     */
    SolverOptimize(prob, &status, &objval);

    // Getting the final values of the variables:
    SolverVarFinalValue(prob, L"Vars", x);

    // Releasing the memory used by freeing the Problem:
    SolverProbFree(&prob);

    printf("quadratically_constrained\n");
    printf("x[0] = %g\n", x[0]);
    printf("x[1] = %g\n", x[1]);
    printf("Obj = %g\n", objval );
    printf("Status = %d\n", status);

    return;
}
```

Nonsmooth Problems

Definition

Nonsmooth Programs (NSP) (or "Problems") are programs in which at least one of the functions (objective or constraint) is a nonsmooth nonlinear function. Mathematically, NSPs look like this:

Minimize $f(x)$
subject to: $l \le g(x) \le u$
$lb \le x \le ub$
where $f(x)$ and/or $g(x)$ are non-smooth nonlinear functions.

Examples >**Nonsmooth Problem** Examples

The Example Problem in Algebraic Notation

Minimize: $2x_1 + x_2 - 5\log(x_1) * \sin(x_2)$
subject to: $x_1 * x_1 \le 10$
$|x_1 - x_2| \le 2$
where $0.1 \le x_1 \le 5$
$0.1 \le x_2 \le 3$

(Solution: $x_1 = 0.1$
$x_2 = 1.6617$
objective = 13.3271)

The Example Problem in VB.NET

```
Private Function NLPEvaluator2_OnEvaluate(ByVal evaluator As
                SolverPlatform.Evaluator) As
                SolverPlatform.Engine_Action Handles
                NLPEvaluator2.OnEvaluate
' This Evaluator computes the values of the objective  and
' constraint functions for a given set of variable values.

   Dim prob As SolverPlatform.Problem
   prob = evaluator.Problem

   Dim Var() As Double
```

```
    ' Retrieving the current variable values:
    Var = prob.VarDecision.Value.Array

    ' Storing the computed function values in an array, Fcn:
    Dim Fcn(2) As Double

    Fcn(0) = Var(0) * Var(1)                ' x1 + x2
    Fcn(1) = Math.Abs(Var(0) - Var(1))      ' | x1 - x2 |

    Dim obj As Double
    ' Computing the objective, 2*x1 + x2 - 5*log(x1)*sin(x2):
    obj = 2 * Var(0) + Var(1) - 5 * Math.Log(Var(0)) *
              Math.Sin(Var(1))

    ' Passing the function values to the Solver:
    prob.FcnConstraint.Value.Array = Fcn

    ' Passing the objective value to the Solver:
    prob.FcnObjective.Value(prob.ObjectiveIndex) = obj

    NLPEvaluator2_OnEvaluate = Engine_Action.Continue

End Function

Private Sub nonsmooth_nonlinear_example()

    NLPEvaluator2 = New SolverPlatform.Evaluator
    EvalIterator = New SolverPlatform.Evaluator

    Dim nvars As Integer, ncons As Integer
    nvars = 2                           ' 2 decision variables.
    ncons = 2                           ' 3 constraint functions.

    Dim rhsValue(1) As Double
    rhsValue(0) = 10
    rhsValue(1) = 2

    Dim rhs(1) As Double
    rhs(0) = 5
    rhs(1) = 3
```

```
Dim lhs(1) As Double
lhs(0) = 0.1
lhs(1) = 0.1

Dim prob As SolverPlatform.Problem = Nothing

Try

   prob = New SolverPlatform.Problem  ' Creating a Problem:

   prob.VarDecision.Size = nvars      ' 2 variables.
   prob.FcnConstraint.Size = ncons    ' 2 constraints.

   ' Constraint right-hand sides:
   prob.FcnConstraint.UpperBound.Array = rhsValue

   prob.VarDecision.LowerBound.Array = lhs ' Variables ≥ 0.1.
   prob.VarDecision.UpperBound.Array = rhs ' x1 ≤ 5, x2 ≤ 3.

   prob.Solver.SolverType = Solver_Type.Maximize

   ' Setting the Problem Type:
   prob.ProblemType = Problem_Type.OptNLP

   Dim engine As Engine = Nothing
   ' Specifying the Engine to solve the problem (A call to an
   ' external  DLL  is  made  because the Engine required for
   ' solving this  type  of  problem  -  the  Large-Scale GRG
   ' Engine - is not available internally.):
   engine = New Engine("name",
            "C:\\Platform\\Src\\LSGRGeng.dll")
   prob.Engine = engine

   ' Setting the Evaluator:
   prob.Evaluators.Item(Eval_Type.Function) = NLPEvaluator2()

   prob.Solver.Optimize()                ' Solving the model.

   Dim convalue() As Double
```

```
        ' Final constraint values:
        convalue = prob.FcnConstraint.FinalValue.Array

        Dim varvalue() As Double

        ' Final variable values:
        varvalue = prob.VarDecision.FinalValue.Array

        Dim objValue As Double

        ' Final objective value:
        objValue = prob.FcnObjective.FinalValue(0)

        Dim status As Optimize_Status
        status = prob.Solver.OptimizeStatus ' Optimization status.

        Console.WriteLine( "Optimize Status = " & status )
        Console.WriteLine( "x1 = " & varvalue(0) )
        Console.WriteLine( "x2 = " & varvalue(1) )
        Console.WriteLine( "Obj = " & objValue )

    Catch ex As SolverException

        Console.WriteLine( ex.Message )
        Console.WriteLine( ex.ExceptionType )

    End Try

    If Not prob Is Nothing Then
        prob.Dispose()      ' Freeing up the license the right way.
    End If

    NLPEvaluator2 = Nothing

    prob = Nothing

End Sub
```

The Example Problem in C++

```
class CNLPEvaluator2 : public IEvaluator
{

/* This Evaluator computes the values of the objective and
   constraint functions for a given set of variable values.   */

public:
   virtual _Engine_Action Evaluate(CEvaluator& evaluator)
   {
      // Retrieving the current variable values:
      const double* pVar = evaluator.Problem.VarDecision.Value;


      cout << "Evaluation x1 = " << pVar[0] << ", x2 = " <<
              pVar[1] << ", Iteration = " <<
         evaluator.Problem.Engine.Stat.FunctionEvals<< endl;


      /* Geting a convenient pointer to the constraint
         functions:                                          */
      double* pFcn = evaluator.Problem.FcnConstraint.Value;


      pFcn[0] = pVar[0] * pVar[1];          // x1 + x2
      pFcn[1] = abs(pVar[0] - pVar[1]);     // | x1 - x2 |


      // Computing the objective, 2*x1 + x2 - 5*log(x1)*sin(x2):
      evaluator.Problem.FcnObjective.Value[evaluator.Problem.get
              _ObjectiveIndex()] =  2*pVar[0] + pVar[1] -
              5*log( pVar[0] )*sin(pVar[1]);


      return Engine_Action_Continue;
   }
};


int nonsmooth_nonlinear_example()
{
   try
   {
      /* 2 decision variables,
         2 constraint functions:                             */
      int nvars = 2, ncons = 2;


      double rhsvalue[] = { 10, 2 };
```

```
double rhs[] = { 5, 3 };          // Variables upper bound.

double lhs[] = { 0.1, 0.1 } ;     // Variables lower bound.

// Creating the Problem:
CProblem problem(Solver_Type_Maximize, nvars, ncons);

// Constraint right-hand sides:
problem.FcnConstraint.UpperBound = rhsvalue;

problem.VarDecision.UpperBound = rhs;    // x1 ≤ 5, x2 ≤ 3

problem.VarDecision.LowerBound = lhs;    // Variables ≥ 0.1

// Specifying the Problem Type:
problem.ProblemType = Problem_Type_OptNLP;

/* Specifying the Engine to solve the problem (A call to
   an external DLL is made because the Engine required for
   solving this type of problem - the  Large-Scale GRG
   Engine - is not available internally.):              */
CEngine engine( L"name1",
        L"C:\\Platform\\Src\\LSGRGeng.dll" );
problem.Engine = engine;

CNLPEvaluator2 evalNLP;           // Creating an Evaluator.

/* Passing the Evaluator (Note that this evaluator is
   required.):                                          */
problem.Evaluators[Eval_Type_Function].Evaluator = &
        evalNLP;

problem.Solver.Optimize();
_Optimize_Status status = problem.Solver.OptimizeStatus;

const double* convalue = problem.FcnConstraint.FinalValue;
const double* varvalue = problem.VarDecision.FinalValue;

cout << "nonsmooth_nonlinear_example" << endl;
cout << "Status = " << status << endl;
```

```
      cout << "x1 = " << varvalue[0] << endl;
      cout << "x2 = " << varvalue[1] << endl;
      cout << "Obj = " << problem.FcnObjective.FinalValue[0] <<
              endl;
      cout << endl;

   } catch(CSolverException e)
   {
      cout << "nonsmooth_nonlinear_example" << endl;
      cout << "Throws Exception" << endl;
      cout << "Code = " << e.ErrorCode << endl;
      cout << "Text = " << ToCStr(e.Description).c_str() <<
              endl;
      return 1;
   }
   return 0;
}
```

The Example Problem in C#

```
private Engine_Action NLPEvaluator2_Evaluate(Evaluator
              evaluator)
{

/* This Evaluator computes the values of the objective and
   constraint functions for a given set of variable values.   */

   Console.WriteLine("Evaluation x1 = " +
              evaluator.Problem.VarDecision.Value[0] +
      ", x2 = " + evaluator.Problem.VarDecision.Value[1] + ",
              Eval = " +
      evaluator.Problem.Engine.Stat.FunctionEvals);

   evaluator.Problem.FcnConstraint.Value[0] =
      evaluator.Problem.VarDecision.Value[0] *
      evaluator.Problem.VarDecision.Value[1];      // x1 + x2

   evaluator.Problem.FcnConstraint.Value[1] =
      Math.Abs( evaluator.Problem.VarDecision.Value[0] -
      evaluator.Problem.VarDecision.Value[1] );    // | x1 -
               x2 |
```

```
      // Computing the objective, 2*x1 + x2 - 5*log(x1)*sin(x2):
   evaluator.Problem.FcnObjective.Value[evaluator.Problem.Object
            iveIndex] =
      2*evaluator.Problem.VarDecision.Value[0]  +
      evaluator.Problem.VarDecision.Value[1] -
      5*Math.Log(evaluator.Problem.VarDecision.Value[0])*Math.Si
            n(evaluator.Problem.VarDecision.Value[1]);

   return Engine_Action.Continue;
}

private void nonsmooth_nonlinear_example()
{
   try
   {
      /* 2 decision variables,
         2 constraint functions:                                  */
      int nvars = 2, ncons = 2;

      double [] rhsValue = { 10, 2 };

      double [] rhs = { 5, 3 };        // Variables upper bound.

      double [] lhs = { 0.1, 0.1 };    // Variables lower bound.

      using(Problem problem = new Problem(Solver_Type.Maximize,
            nvars, ncons))
      {
         // Constraint right-hand sides:
         problem.FcnConstraint.UpperBound.Array = rhsValue;

         // x1 ≤ 5, x2 ≤ 3:
         problem.VarDecision.UpperBound.Array = rhs ;

         // Variables ≥ 0.1:
         problem.VarDecision.LowerBound.Array = lhs;

         // Specifying the Problem Type:
         problem.ProblemType = Problem_Type.OptNLP;
```

```
            /* Specifying the Engine to solve the problem (A call
               to an external DLL is made because the Engine
               required for solving this type of problem - the
               Large-Scale GRG Engine - is not available
               internally.):                                    */
            Engine engine = new Engine ( "name",
                 "C:\\Platform\\Src\\LSGRGeng.dll" );
            problem.Engine = engine;


            /* Creating and passing the Evaluator (Note that this
               evaluator is required.):                         */
               problem.Evaluators[Eval_Type.Function].OnEvaluate +=
                  new EvaluateEventHandler(NLPEvaluator2_Evaluate);


            problem.Solver.Optimize();


            Optimize_Status status = problem.Solver.OptimizeStatus;


            Console.WriteLine("\r\nnonsmooth_nonlinear_example");
            Console.WriteLine("Optimize Status = " + status);


            Console.WriteLine("x1 = " + varvalue[0]);
            Console.WriteLine("x2 = " + varvalue[1]);


            Console.WriteLine("Obj = " + objValue[0]);
         }
      }


      catch(SolverException ex)
      {
         Console.WriteLine("Exception " + ex.ExceptionType);
         Console.WriteLine("Exception " + ex.Message);
      }
   }
```

The Example Problem in Procedural C

```
_Engine_Action Evaluate2(void* prob, void* refuser, _Eval_Type
               evaltype )
{

/* This Evaluator computes the values of the objective and
   constraint functions for a given set of variable values.   */
```

```
   double var[2];
   double fcn[2];
   double obj[1];

   /* Retrieving the current variable values (the computed
      function values are stored in an array, fcn.):          */
   SolverVarValue(prob, L"Vars", var);
   fcn[0] = var[0] * var[1];          // x1 + x2
   fcn[1] = abs(var[0] - var[1]);     // | x1 - x2 |

   // Computing the objective, 2*x1 + x2 - 5*log(x1)*sin(x2):
   obj[0] = 2*var[0] + var[1] - 5*log( var[0] )*sin( var[1] );

   // Passing the function values to the Solver:
   SolverFcnValueSet(prob, L"Constraints", fcn);

   // Passing the objective value to the Solver:
   SolverFcnValueSet(prob, L"Objective", obj);
    return Engine_Action_Continue;
}

void nonsmooth_nonlinear_example()
{
   void * prob;
   _Optimize_Status status;
   double objval;
   double rhsvalue[] = { 10, 2 };
   double rhs[] = { 5, 3 };         // Variables upper bound.
   double lhs[] = { 0.1, 0.1 };     // Variables lower bound.
   double x[2];

   SolverProblem(&prob);
   SolverProbTypeSet(prob,
              Solver_Type_Maximize,Problem_Type_OptNLP);
   SolverVariableAdd(prob, L"Vars", Variable_Type_Decision, 2);
   SolverFunctionAdd(prob, L"Constraints",
              Function_Type_Constraint, 3);
   SolverFunctionAdd(prob, L"Objective",
              Function_Type_Objective, 1);
```

```
   /* Passing the right-hand sides of the constraints (There are
      ways two do it: we can either pass the lower and upper
      bounds on the functions, as we do here, or we can use the
      Relation method.):                                        */
   SolverFcnUpperBoundSet(prob, L"Constraints", rhsvalue);

   // Variables ≥ 0.1
   SolverVarLowerBoundSet(prob, L"Vars",lhs);

   // x1 ≤ 5, x2 ≤ 3
   SolverVarUpperBoundSet(prob, L"Vars", rhs);

   /* Specifying the Engine to solve the problem (A call to an
      external DLL is made because the Engine required for
      solving this type of problem - the  Large-Scale GRG Engine
      - is not available internally.):                          */
   SolverEngineAdd(prob,L"MyEngine",L"C:\\Platform\\Src\\LSGRGen
               g.dll");

   SolverEvaluatorAdd(prob, Evaluate2, NULL,
               Eval_Type_Function);

   SolverOptimize(prob, &status, &objval);
   SolverVarFinalValue(prob, L"Vars", x);

   SolverProbFree(&prob);

   printf("nonsmooth_nonlinear_example\n");
   printf("x[0] = %g\n", x[0]);
   printf("x[1] = %g\n", x[1]);
   printf("Obj = %g\n", objval);
   printf("Status = %d\n", status);
}
```

Definition of Monte Carlo Simulation

"Monte Carlo simulation" is a generic term for algorithms using random numbers to solve various kinds of computational problems by (or more often pseudo-random numbers), as opposed to deterministic algorithms.

The Monte Carlo simulation is the basis for probabilistic risk analysis as it replicates real life occurrences by mathematically modeling a projected event. It uses pre-defined probability distributions of risk variables to perform random modeling over many "simulations" or computer trials.

With the Solver SDK Platform, Monte Carlo simulation is implemented by calling the Simulation Engine.

Examples >**Simulation Problem** Example

The Example Problem in Algebraic Notation

This example demonstrates calling the Simulation Engine to perform the following Monte Carlo simulation:

Given two uncertain variables X and Y with distributions:

x = PsiUniform(0, 2) and y = PsiNormal(1, 1)

Perform 10 trials and calculate the functions:

$x + y$
$x * y$

The Example Problem in VB 6

```
Private Sub Example_Click()

   TextOut.Text = ""    ' Clears the output window.

   Dim nvars As Integer
   Dim nfcns As Integer

   nvars = 2
   nfcns = 2
```

```
Dim correl(3) As Double

correl(0) = 1#
correl(1) = 0.5
correl(2) = 0.5
correl(3) = 1#

Dim prob As SolverPlatform.Problem

Set prob = New SolverPlatform.Problem

prob.Solver.SolverType = Solver_Type_Simulate

prob.VarUncertain.Size = nvars
prob.FcnUncertain.Size = nfcns

Set SimEvaluator = New Evaluator

prob.Evaluators(Eval_Type_Sample) = SimEvaluator

prob.Solver.NumTrials = 10

prob.Model.Distribution 0, "PsiUniform(0,2)"
prob.Model.Distribution 1, "PsiNormal(1,1)"

Dim matDistCor As SolverPlatform.DoubleMatrix

Set matDistCor = New SolverPlatform.DoubleMatrix

matDistCor.ArrayOrder = Array_Order_ByRow
matDistCor.InitDense nvars, nvars, correl

prob.Model.DistCorrelations = matDistCor

prob.Solver.Simulate

TextOut.Text = TextOut.Text & "Status = " &
           prob.Solver.SimulateStatus & Chr(13) & Chr(10)
```

```
Dim i As Integer

For i = 0 To nvars - 1
   TextOut.Text = TextOut.Text & Chr(13) & Chr(10)
   TextOut.Text = TextOut.Text & "Variable " & i & Chr(13) &
           Chr(10)
   TextOut.Text = TextOut.Text & "Minimum  = " &
           prob.VarUncertain.Statistics.Minimum(i) & Chr(13)
           & Chr(10)
   TextOut.Text = TextOut.Text & "Maximum  = " &
           prob.VarUncertain.Statistics.Maximum(i) & Chr(13)
           & Chr(10)
   TextOut.Text = TextOut.Text & "Mean     = " &
           prob.VarUncertain.Statistics.Mean(i) & Chr(13) &
           Chr(10)
   TextOut.Text = TextOut.Text & "StdDev   = " &
           prob.VarUncertain.Statistics.StdDev(i) & Chr(13)
           & Chr(10)
   TextOut.Text = TextOut.Text & "Variance = " &
           prob.VarUncertain.Statistics.Variance(i) &
           Chr(13) & Chr(10)
   TextOut.Text = TextOut.Text & "Skewness = " &
           prob.VarUncertain.Statistics.Skewness(i) &
           Chr(13) & Chr(10)
   TextOut.Text = TextOut.Text & "Kurtosis = " &
           prob.VarUncertain.Statistics.Kurtosis(i) &
           Chr(13) & Chr(10)
   TextOut.Text = TextOut.Text & "10th Percentile = " &
           prob.VarUncertain.Percentiles(i, 9) & Chr(13) &
           Chr(10)
   TextOut.Text = TextOut.Text & "90th Percentile = " &
           prob.VarUncertain.Percentiles(i, 89) & Chr(13) &
           Chr(10)
Next

TextOut.Text = TextOut.Text & Chr(13) & Chr(10)
TextOut.Text = TextOut.Text & "Statistics on Uncertain
           Functions" & Chr(13) & Chr(10)

For i = 0 To nfcns - 1
   TextOut.Text = TextOut.Text & Chr(13) & Chr(10)
   TextOut.Text = TextOut.Text & "Function " & i & Chr(13) &
           Chr(10)
   TextOut.Text = TextOut.Text & "Minimum  = " &
           prob.FcnUncertain.Statistics.Minimum(i) & Chr(13)
           & Chr(10)
```

```
            TextOut.Text = TextOut.Text & "Maximum  = " &
                    prob.FcnUncertain.Statistics.Maximum(i) & Chr(13)
                    & Chr(10)
            TextOut.Text = TextOut.Text & "Mean     = " &
                    prob.FcnUncertain.Statistics.Mean(i) & Chr(13) &
                    Chr(10)
            TextOut.Text = TextOut.Text & "StdDev   = " &
                    prob.FcnUncertain.Statistics.StdDev(i) & Chr(13)
                    & Chr(10)
            TextOut.Text = TextOut.Text & "Variance = " &
                    prob.FcnUncertain.Statistics.Variance(i) &
                    Chr(13) & Chr(10)
            TextOut.Text = TextOut.Text & "Skewness = " &
                    prob.FcnUncertain.Statistics.Skewness(i) &
                    Chr(13) & Chr(10)
            TextOut.Text = TextOut.Text & "Kurtosis = " &
                    prob.FcnUncertain.Statistics.Kurtosis(i) &
                    Chr(13) & Chr(10)
            TextOut.Text = TextOut.Text & "10th Percentile = " &
                    prob.FcnUncertain.Percentiles(i, 9) & Chr(13) &
                    Chr(10)
            TextOut.Text = TextOut.Text & "90th Percentile = " &
                    prob.FcnUncertain.Percentiles(i, 89) & Chr(13) &
                    Chr(10)
        Next

        Set SimEvaluator = Nothing
        Set prob = Nothing

    End Sub
```

The Example Problem in VB.NET

```
    Private Sub btnExample_Click(ByVal sender As System.Object,
                ByVal e As System.EventArgs) Handles
                btnExample.Click

        m_Log.Clear()

        Dim nvars As Integer = 2
        Dim nfcns As Integer = 2

        Dim correl() As Double = New Double() {1.0, 0.5, 0.5, 1.0}
```

```
Dim problem As Problem = New Problem(Solver_Type.Simulate,
            nvars, nfcns)

SimEvaluator = New Evaluator

problem.Evaluators(Eval_Type.Sample) = SimEvaluator

problem.Solver.NumTrials = 10

problem.Model.Distribution(0, "PsiUniform(0,2)")
problem.Model.Distribution(1, "PsiNormal(1,1)")

problem.Model.DistCorrelations = New
            DoubleMatrix(Array_Order.ByRow, nvars, nvars,
            correl)

problem.Solver.Simulate()

m_Log.Dump("Status = " +
            problem.Solver.SimulateStatus.ToString())
m_Log.Dump("")
m_Log.Dump("Statistics on Uncertain Variables")

Dim i As Integer

For i = 0 To nvars - 1
   m_Log.Dump("")
   m_Log.Dump("Variable " + i.ToString())
   m_Log.Dump("Minimum  = " +
            problem.VarUncertain.Statistics.Minimum(i).ToStri
            ng())
   m_Log.Dump("Maximum  = " +
            problem.VarUncertain.Statistics.Maximum(i).ToStri
            ng())
   m_Log.Dump("Mean     = " +
            problem.VarUncertain.Statistics.Mean(i).ToString(
            ))
   m_Log.Dump("StdDev   = " +
            problem.VarUncertain.Statistics.StdDev(i).ToStrin
            g())
   m_Log.Dump("Variance = " +
            problem.VarUncertain.Statistics.Variance(i).ToStr
            ing())
```

```
   m_Log.Dump("Skewness = " +
            problem.VarUncertain.Statistics.Skewness(i).ToStr
            ing())
   m_Log.Dump("Kurtosis = " +
            problem.VarUncertain.Statistics.Kurtosis(i).ToStr
            ing())
   m_Log.Dump("10th Percentile = " +
            problem.VarUncertain.Percentiles(i, 9).ToString()
            )
   m_Log.Dump("90th Percentile = " +
            problem.VarUncertain.Percentiles(i, 89).ToString(
            ))
Next

m_Log.Dump("")
m_Log.Dump("Statistics on Uncertain Functions")

For i = 0 To nfcns - 1
   m_Log.Dump("")
   m_Log.Dump("Function " + i.ToString())
   m_Log.Dump("Minimum  = " +
            problem.FcnUncertain.Statistics.Minimum(i).ToStri
            ng())
   m_Log.Dump("Maximum  = " +
            problem.FcnUncertain.Statistics.Maximum(i).ToStri
            ng())
   m_Log.Dump("Mean     = " +
            problem.FcnUncertain.Statistics.Mean(i).ToString(
            ))
   m_Log.Dump("StdDev   = " +
            problem.FcnUncertain.Statistics.StdDev(i).ToStrin
            g())
   m_Log.Dump("Variance = " +
            problem.FcnUncertain.Statistics.Variance(i).ToStr
            ing())
   m_Log.Dump("Skewness = " +
            problem.FcnUncertain.Statistics.Skewness(i).ToStr
            ing())
   m_Log.Dump("Kurtosis = " +
            problem.FcnUncertain.Statistics.Kurtosis(i).ToStr
            ing())
   m_Log.Dump("10th Percentile = " +
            problem.FcnUncertain.Percentiles(i, 9).ToString()
            )
   m_Log.Dump("90th Percentile = " +
            problem.FcnUncertain.Percentiles(i, 89).ToString(
            ))
```

```
    Next

    SimEvaluator = Nothing
    problem = Nothing

    End Sub

    Private Function SimEvaluator_OnEvaluate(ByVal evaluator As S
                olverPlatform.Evaluator) As SolverPlatform.Engine
                _Action Handles SimEvaluator.OnEvaluate

    ' Retrieving the current uncertain variable values:
    Dim Var() As Double =
                evaluator.Problem.VarUncertain.Value.Array

    m_Log.Dump("Evaluation x1 = " + Var(0).ToString() + ", x2 = "
                 + Var(1).ToString() + _
      ", Trial = " + evaluator.Problem.Engine.Stat.FunctionEvals.
                ToString())

    ' Computing the function values given the current variable
    ' values:

    ' X+Y:
    evaluator.Problem.FcnUncertain.Value(0) = Var(0) + Var(1)

    ' X*Y:
    evaluator.Problem.FcnUncertain.Value(1) = Var(0) * Var(1)

    SimEvaluator_OnEvaluate = Engine_Action.Continue

    End Function
```

The Example Problem in C++

```
void example()
{
    _Simulate_Status status;
    HProblem prob;
    int nvars = 2, nfcns = 2;
    double correl[] = { 1.0, 0.5, 0.5, 1.0 };
```

```
int numvalues, numerrors;

double minimum[2];
double maximum[2];
double mean[2];
double stddev[2];
double variance[2];
double skewness[2];
double kurtosis[2];
double mode[2];
double percentiles[2*100];

int i;

SolverProblem(&prob);   //Creates a new Problem.
SolverProbTypeSet(prob, Solver_Type_Simulate,
           Problem_Type_NA);

SolverVariableAdd(prob, L"Vars", Variable_Type_Uncertain,
           nvars);
SolverFunctionAdd(prob, L"Funcs", Function_Type_Uncertain,
           nfcns);

SolverNumTrialsSet(prob, 10);

SolverModDistInit(prob, Dist_Var, 0, L"PsiUniform(0,2)" );
SolverModDistInit(prob, Dist_Var, 1, L"PsiNormal(1,1)" );

SolverModArrayOrder(prob, Array_Order_ByCol);

SolverModDistCorrelationsSet(prob, 4, NULL, NULL, correl);

SolverEvaluatorAdd(prob, EvalSim, NULL, Eval_Type_Sample);

SolverSimulate(prob, & status);

printf("Example 17\n");
printf("Status = %d\n", status);
printf("\nStatistics on Uncertain Variables\n");
```

```
SolverVarStatistics(prob, L"Vars", & numvalues, & numerrors,
            minimum, maximum, mean, stddev, variance,
            skewness, kurtosis, mode);

SolverVarPercentiles(prob, L"Vars", Array_Order_ByRow,
            percentiles);

for(i = 0; i < nvars; i ++)
{
   printf("Variable %d\n", i);
   printf("Minimum  = %g\n", minimum[i]);
   printf("Maximum  = %g\n", maximum[i]);
   printf("Mean     = %g\n", mean[i]);
   printf("StdDev   = %g\n", stddev[i]);
   printf("Variance = %g\n", variance[i]);
   printf("Skewness = %g\n", skewness[i]);
   printf("Kurtosis = %g\n", kurtosis[i]);

   printf("10th Percentile %g\n", percentiles[i * nvars +
            9]);
   printf("90th Percentile = %g\n", percentiles[i * nvars +
            89]);
}

printf("\nStatistics on Uncertain Functions\n");

SolverFcnStatistics(prob, L"Funcs", & numvalues, & numerrors,
            minimum, maximum, mean, stddev, variance,
            skewness, kurtosis, mode);

SolverFcnPercentiles(prob, L"Funcs", Array_Order_ByRow,
            percentiles);

for (i = 0; i < nfcns; i++)
{
   printf("Function %d\n", i);
   printf("Minimum  = %g\n", minimum[i]);
   printf("Maximum  = %g\n", maximum[i]);
   printf("Mean     = %g\n", mean[i]);
   printf("StdDev   = %g\n", stddev[i]);
   printf("Variance = %g\n", variance[i]);
   printf("Skewness = %g\n", skewness[i]);
```

```
      printf("Kurtosis = %g\n", kurtosis[i]);
      printf("10th Percentile = %g\n", percentiles[i * nvars +
            9]);
      printf("90th Percentile = %g\n", percentiles[i * nvars +
            89]);
   }

   SolverProbFree(&prob);
}
```

The Example Problem in C#

```
private void btnExample_Click(object sender, System.EventArgs e)
{
   m_Log.Clear();

   try
   {
      int nvars = 2, nfcns = 2;
      double [] correl = new double [] { 1.0, 0.5, 0.5, 1.0 };

      using(Problem problem = new Problem(Solver_Type.Simulate,
            nvars, nfcns))
      {

         problem.Evaluators[Eval_Type.Sample].OnEvaluate += new
               EvaluateEventHandler(SimEvaluator_OnEvaluate);
         problem.Solver.NumTrials = 10;

         problem.Model.Distribution( 0, "PsiUniform(0,2)" );
         problem.Model.Distribution( 1, "PsiNormal(1,1)" );

         problem.Model.DistCorrelations = new
               DoubleMatrix(Array_Order.ByRow, nvars, nvars,
               correl);

         problem.Solver.Simulate();

         m_Log.Dump("Status = " +
               problem.Solver.SimulateStatus);
         m_Log.Dump("\r\nStatistics on Uncertain Variables");
```

```
for (int i = 0; i < nvars; i++)
{
   m_Log.Dump("\r\nVariable " + i);
   m_Log.Dump("Minimum  = " +
     problem.VarUncertain.Statistics.Minimum[i]);
   m_Log.Dump("Maximum  = " +
     problem.VarUncertain.Statistics.Maximum[i]);
   m_Log.Dump("Mean     = " +
     problem.VarUncertain.Statistics.Mean[i]);
   m_Log.Dump("StdDev   = " +
     problem.VarUncertain.Statistics.StdDev[i]);
   m_Log.Dump("Variance = " +
     problem.VarUncertain.Statistics.Variance[i]);
   m_Log.Dump("Skewness = " +
     problem.VarUncertain.Statistics.Skewness[i]);
   m_Log.Dump("Kurtosis = " +
     problem.VarUncertain.Statistics.Kurtosis[i]);
   m_Log.Dump("10th Percentile = " +
     problem.VarUncertain.Percentiles[i, 9]);
   m_Log.Dump("90th Percentile = " +
     problem.VarUncertain.Percentiles[i, 89]);
}

m_Log.Dump("\r\nStatistics on Uncertain Functions");

for (int i = 0; i < nfcns; i++)
{
   m_Log.Dump("\r\nFunction " + i);
   m_Log.Dump("Minimum  = " +
     problem.FcnUncertain.Statistics.Minimum[i]);
   m_Log.Dump("Maximum  = " +
     problem.FcnUncertain.Statistics.Maximum[i]);
   m_Log.Dump("Mean     = " +
     problem.FcnUncertain.Statistics.Mean[i]);
   m_Log.Dump("StdDev   = " +
     problem.FcnUncertain.Statistics.StdDev[i]);
   m_Log.Dump("Variance = " +
     problem.FcnUncertain.Statistics.Variance[i]);
   m_Log.Dump("Skewness = " +
     problem.FcnUncertain.Statistics.Skewness[i]);
   m_Log.Dump("Kurtosis = " +
     problem.FcnUncertain.Statistics.Kurtosis[i]);
```

```
            m_Log.Dump("10th Percentile = " +
               problem.FcnUncertain.Percentiles[i,9]);
            m_Log.Dump("90th Percentile = " +
               problem.FcnUncertain.Percentiles[i,89]);
         }
      }
   }
   catch(SolverException ex)
   {
      m_Log.Dump("Exception " + ex.ExceptionType);
      m_Log.Dump("Exception " + ex.Message);
   }
}

private Engine_Action SimEvaluator_OnEvaluate(Evaluator
               evaluator)
{
   // Requesting the current uncertain variable values:
   double [] Var = evaluator.Problem.VarUncertain.Value.Array;

   m_Log.Dump("Evaluation x1 = " + Var[0] + ", x2 = " + Var[1] +
      ", Trial = " +
               evaluator.Problem.Engine.Stat.FunctionEvals);

   /* Computing the function values given the current variable
      values:                                                  */

   // X+Y:
   evaluator.Problem.FcnUncertain.Value[0] = Var[0] + Var[1];

   // X*Y:
   evaluator.Problem.FcnUncertain.Value[1] = Var[0] * Var[1];

   return Engine_Action.Continue;
}
```

The Example Problem in Procedural C

```
void example()
{
```

```
_Simulate_Status status;
HProblem prob;
int nvars = 2, nfcns = 2;
double correl[] = { 1.0, 0.5, 0.5, 1.0 };

int numvalues, numerrors;

double minimum[2];
double maximum[2];
double mean[2];
double stddev[2];
double variance[2];
double skewness[2];
double kurtosis[2];
double mode[2];
double percentiles[2*100];

int i;

SolverProblem(&prob);   // Creates a new Problem.
SolverProbTypeSet(prob, Solver_Type_Simulate,
           Problem_Type_NA);

SolverVariableAdd(prob, L"Vars", Variable_Type_Uncertain,
           nvars);
SolverFunctionAdd(prob, L"Funcs", Function_Type_Uncertain,
           nfcns);

SolverNumTrialsSet(prob, 10);

SolverModDistribution(prob, 0, L"PsiUniform(0,2)" );
SolverModDistribution(prob, 1, L"PsiNormal(1,1)" );

SolverModArrayOrder(prob, Array_Order_ByCol);

SolverModDistCorrelationsSet(prob, 4, NULL, NULL, correl);

SolverEvaluatorAdd(prob, EvalSim, NULL, Eval_Type_Sample);

SolverSimulate(prob, & status);
```

```
printf("Example 17\n");
printf("Status = %d\n", status);
printf("\nStatistics on Uncertain Variables\n");

SolverVarStatistics(prob, L"Vars", & numvalues, & numerrors,
           minimum, maximum, mean, stddev, variance,
           skewness, kurtosis, mode);

SolverVarPercentiles(prob, L"Vars", Array_Order_ByRow,
           percentiles);

for(i = 0; i < nvars; i ++)
{
   printf("Variable %d\n", i);
   printf("Minimum  = %g\n", minimum[i]);
   printf("Maximum  = %g\n", maximum[i]);
   printf("Mean     = %g\n", mean[i]);
   printf("StdDe    = %g\n", stddev[i]);
   printf("Variance = %g\n", variance[i]);
   printf("Skewness = %g\n", skewness[i]);
   printf("Kurtosis = %g\n", kurtosis[i]);

   printf("10th Percentile %g\n", percentiles[i * nvars +
           9]);
   printf("90th Percentile = %g\n", percentiles[i * nvars +
           89]);
}

printf("\nStatistics on Uncertain Functions\n");

SolverFcnStatistics(prob, L"Funcs", & numvalues, & numerrors,
           minimum, maximum, mean, stddev, variance,
           skewness, kurtosis, mode);

SolverFcnPercentiles(prob, L"Funcs", Array_Order_ByRow,
           percentiles);

for (i = 0; i < nfcns; i++)
{
   printf("Function %d\n", i);
   printf("Minimum  = %g\n", minimum[i]);
```

```
      printf("Maximum  = %g\n", maximum[i]);
      printf("Mean     = %g\n", mean[i]);
      printf("StdDev   = %g\n", stddev[i]);
      printf("Variance = %g\n", variance[i]);
      printf("Skewness = %g\n", skewness[i]);
      printf("Kurtosis = %g\n", kurtosis[i]);
      printf("10th Percentile = %g\n", percentiles[i * nvars +
              9]);
      printf("90th Percentile = %g\n", percentiles[i * nvars +
              89]);
   }

   SolverProbFree(&prob);
}
```

The Example Problem in Java

```
public class Example implements IEvaluator
{

   public int Evaluate(Evaluator evaluator)
   {
      try
      {
         // Retrieving the current uncertain variable values:
         double Var[] =
               evaluator.Problem().VarUncertain().Value();

         System.out.println("Evaluation x1 = " + Var[0] + ", x2
               = " + Var[1] + ", Trial = " +
               evaluator.Problem().Engine().Stat().FunctionEvals
               ());

         /* Getting a convenient pointer to the uncertain
            functions:                                        */

         double Fcn[] =
               evaluator.Problem().FcnUncertain().Value();

         // Computing the function values given the current
            variable values:                                  */
```

```
            Fcn[0] = Var[0] + Var[1];    // x + y
            Fcn[1] = Var[0] * Var[1];    // x * y

            evaluator.Problem().FcnUncertain().Value(Fcn);
         }
         catch(SolverException e)
         {
            System.out.println(
               "SolverException was generated. Exception
                  Description = " + e.Description());

            return Engine_Action.Stop;
         }

         return Engine_Action.Continue;
      }

      private void Start()
      {
         Problem problem = null;

         try
         {
            int nvars = 2, nfcns = 2;
            double [] correl = new double [] { 1.0, 0.5, 0.5, 1.0
                  };

            problem = new Problem(Solver_Type.Simulate, nvars,
                  nfcns);


         problem.Evaluators().Item(Eval_Type.Sample).
                  Evaluator(this);
            problem.Solver().NumTrials(10);

            problem.Model().Distribution( 0, "PsiUniform(0,2)" );
            problem.Model().Distribution( 1, "PsiNormal(1,1)" );

            problem.Model().DistCorrelations(
               new DoubleMatrix(Array_Order.ByRow, nvars, nvars,
                  correl ));
```

```
problem.Solver().Simulate();


System.out.println("Simulate Status = " +
      problem.Solver().SimulateStatus());
System.out.println("");
System.out.println("Statistics on Uncertain
      Variables");


for (int i = 0; i < nvars; i++)
{
   System.out.println("");
   System.out.println("Variable " + i);
   System.out.println("Minimum  = " +
     problem.VarUncertain().Statistics().Minimum(i));
   System.out.println("Maximum  = " +
     problem.VarUncertain().Statistics().Maximum(i));
   System.out.println("Mean     = " +
     problem.VarUncertain().Statistics().Mean(i));
   System.out.println("StdDev   = " +
     problem.VarUncertain().Statistics().StdDev(i));
   System.out.println("Variance = " +
     problem.VarUncertain().Statistics().Variance(i));
   System.out.println("Skewness = " +
     problem.VarUncertain().Statistics().Skewness(i));
   System.out.println("Kurtosis = " +
     problem.VarUncertain().Statistics().Kurtosis(i));
   System.out.println("10th Percentile = " +
     problem.VarUncertain().Percentiles().Value(i,
     9));
   System.out.println("90th Percentile = " +
     problem.VarUncertain().Percentiles().Value(i,
     89));
}


System.out.println("");
System.out.println("Statistics on Uncertain
      Functions");


for (int i = 0; i < nfcns; i++)
{
   System.out.println("");
   System.out.println("Function " + i);
```

```
            System.out.println("Minimum  = " +
               problem.FcnUncertain().Statistics().Minimum(i));
            System.out.println("Maximum  = " +
               problem.FcnUncertain().Statistics().Maximum(i));
            System.out.println("Mean     = " +
               problem.FcnUncertain().Statistics().Mean(i));
            System.out.println("StdDev   = " +
               problem.FcnUncertain().Statistics().StdDev(i));
            System.out.println("Variance = " +
               problem.FcnUncertain().Statistics().Variance(i));
            System.out.println("Skewness = " +
               problem.FcnUncertain().Statistics().Skewness(i));
            System.out.println("Kurtosis = " +
               problem.FcnUncertain().Statistics().Kurtosis(i));
            System.out.println("10th Percentile = " +
               problem.FcnUncertain().Percentiles().Value(i,
               9));
            System.out.println("90th Percentile = " +
               problem.FcnUncertain().Percentiles().Value(i,
               89));
         }
      }
      catch(SolverException e)
      {
         System.out.println(
            "SolverException was generated. Exception
               Description = " + e.Description());
      }

      if(problem != null)
         problem.Free();
   }

   public static void main(String[] args)
   {
      System.out.println("Example ");

      new Example().Start();
   }
}
```

The Example Problem in Object-Oriented Matlab

```
function Example()

nvars = 2;
nfcns = 2;

correl = [1.0 0.5; 0.5 1.0];

problem = 0;

try

   problem = Problem('Simulate', nvars, nfcns);

   problem.Evaluators('Sample').Evaluator = 'SimEvaluator';
   problem.Solver.NumTrials = 10;

   Distribution(problem.Model, 0, 'PsiUniform(0,2)');
   Distribution(problem.Model, 1, 'PsiNormal(1,1)');

   problem.Model.DistCorrelations = correl;

   Simulate(problem.Solver);

   disp(' ');
   disp('Example ');
   disp(sprintf('Status = %d', problem.Solver.SimulateStatus));
   disp('Statistics on Uncertain Variables');

   for i = 1:nvars

      disp(' ');
      disp(sprintf('Variable %d', i));
      disp(sprintf('Minimum  = %g', problem.VarUncertain.Statist
               ics.Minimum(i-1)));
      disp(sprintf('Maximum  = %g', problem.VarUncertain.Statist
               ics.Maximum(i-1)));
      disp(sprintf('Mean     = %g', problem.VarUncertain.Statisti
               cs.Mean(i-1)));
      disp(sprintf('StdDev   = %g', problem.VarUncertain.Statist
               ics.StdDev(i-1)));
```

```
   disp(sprintf('Variance = %g', problem.VarUncertain.Statist
            ics.Variance(i-1)));
   disp(sprintf('Skewness = %g', problem.VarUncertain.Statist
            ics.Skewness(i-1)));
   disp(sprintf('Kurtosis = %g', problem.VarUncertain.Statist
            ics.Kurtosis(i-1)));
   disp(sprintf('10th Percentile = %g', problem.VarUncertain.
            Percentiles.Value(i-1, 9)));
   disp(sprintf('90th Percentile = %g', problem.VarUncertain.
            Percentiles.Value(i-1, 89)));

end;

disp(' ');
disp('Statistics on Uncertain Functions');

for i = 1:nfcns

   disp(' ');
   disp(sprintf('Function %d', i));
   disp(sprintf('Minimum  =
            %g', problem.FcnUncertain.Statistics.Minimum(i-
            1)));
   disp(sprintf('Maximum  =
            %g', problem.FcnUncertain.Statistics.Maximum(i-
            1)));
   disp(sprintf('Mean     =
            %g', problem.FcnUncertain.Statistics.Mean(i-1)));
   disp(sprintf('StdDev   =
            %g', problem.FcnUncertain.Statistics.StdDev(i-
            1)));
   disp(sprintf('Variance =
            %g', problem.FcnUncertain.Statistics.Variance(i-
            1)));
   disp(sprintf('Skewness =
            %g', problem.FcnUncertain.Statistics.Skewness(i-
            1)));
   disp(sprintf('Kurtosis =
            %g', problem.FcnUncertain.Statistics.Kurtosis(i-
            1)));
   disp(sprintf('10th Percentile =
            %g', problem.FcnUncertain.Percentiles.Value(i-
            1,9)));
   disp(sprintf('90th Percentile =
            %g', problem.FcnUncertain.Percentiles.Value(i-
            1,89)));
```

```
   end

catch

   disp('Example ');
      disp('Throws Exception');

   errstr = lasterror;

   disp(errstr.message);

end

if isa(problem, 'Problem')
   Destroy(problem);
end;
```

The Example Problem in Procedural Matlab

```
function Example()

nvars = 2;
nfcns = 2;
correl = [1.0 0.5; 1.0 0.5];

prob = 0;

try

   prob = SolverSDK('Problem');

   SolverSDK('ProbTypeSet', prob, 'Simulate');

   SolverSDK('VariableAdd', prob, 'Vars', 'Uncertain', nvars);
   SolverSDK('FunctionAdd', prob, 'Funcs', 'Uncertain', nfcns);

   SolverSDK('SolverNumTrialsSet', prob, 10);

   SolverSDK('ModDistribution', prob, 0, 'PsiUniform(0,2)');
```

```
SolverSDK('ModDistribution', prob, 1, 'PsiNormal(1,1)');

SolverSDK('ModDistCorrelationsSet', prob, correl);

SolverSDK('EvaluatorAdd', prob, 'EvalSim', '', 'Sample');

status = SolverSDK('SolverSimulate', prob);

disp('Example ');
disp(sprintf('Status = %d', status));
disp('');
disp('Statistics on Uncertain Variables');

[numvalues, numerrors, minimum, maximum, mean, stddev,
           variance, skewness, kurtosis, mode] =
           SolverSDK('VarStatistics', prob, 'Vars');

percentiles = SolverSDK('VarPercentiles', prob, 'Vars');

for i = 1:nvars

   disp(sprintf('Variable %d', i));
   disp(sprintf('Minimum  = %g', minimum(i)));
   disp(sprintf('Maximum  = %g', maximum(i)));
   disp(sprintf('Mean     = %g', mean(i)));
   disp(sprintf('StdDev   = %g', stddev(i)));
   disp(sprintf('Variance = %g', variance(i)));
   disp(sprintf('Skewness = %g', skewness(i)));
   disp(sprintf('Kurtosis = %g', kurtosis(i)));

   disp(sprintf('10th Percentile %g', percentiles(i, 10)));
   disp(sprintf('90th Percentile = %g', percentiles(i, 90)));
end;

disp('');
disp('Statistics on Uncertain Functions');

[numvalues, numerrors, minimum, maximum, mean, stddev,
           variance, skewness, kurtosis, mode] =
           SolverSDK('FcnStatistics', prob, 'Funcs');
```

```
   percentiles = SolverSDK('FcnPercentiles', prob, 'Funcs');

   for i = 1:nvars

      disp(sprintf('Function %d', i));
      disp(sprintf('Minimum  = %g', minimum(i)));
      disp(sprintf('Maximum  = %g', maximum(i)));
      disp(sprintf('Mean     = %g', mean(i)));
      disp(sprintf('StdDev   = %g', stddev(i)));
      disp(sprintf('Variance = %g', variance(i)));
      disp(sprintf('Skewness = %g', skewness(i)));
      disp(sprintf('Kurtosis = %g', kurtosis(i)));

      disp(sprintf('10th Percentile %g', percentiles(i, 10)));
      disp(sprintf('90th Percentile = %g', percentiles(i, 90)));
   end;

catch

   disp('Example ');
   disp('Throws Exception');

   errstr = lasterror;

   disp(errstr.message);

end

% Releasing the memory used by freeing the problem:
if prob ~= 0
   SolverSDK('ProbFree', prob);
end;
```

Appendix

Solver SDK Object-Oriented API Objects Hierarchy

Figure 1 schematically represents the objects hierarchy in the Solver SDK Object-Oriented API.

Figure 2 shows the objects instantiated upon creation of a problem instance. The grayed names represents objects that are not created by default but can be optionally added by the user.

Figure 1 - Objects Hierarchy Diagram

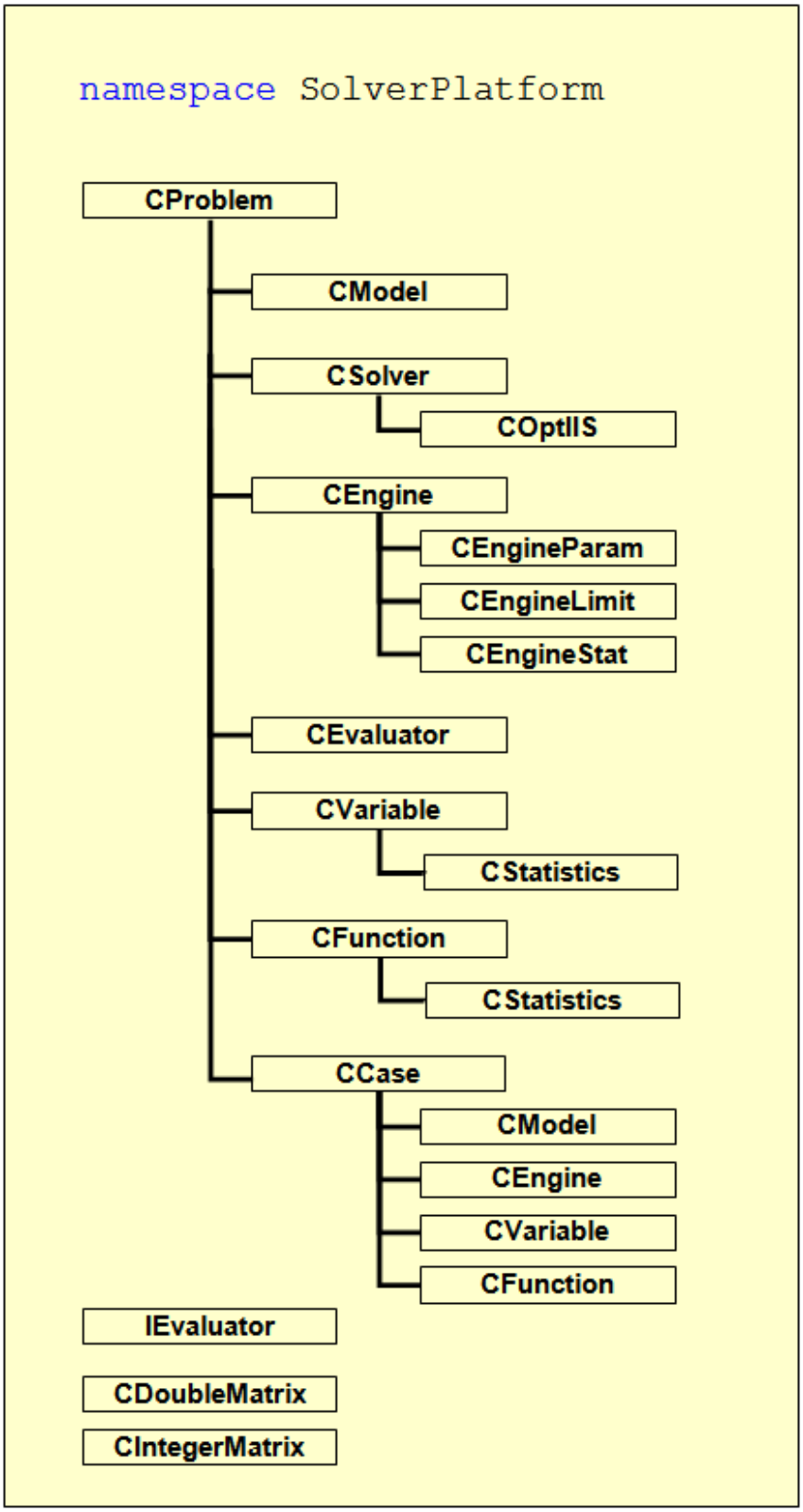


Figure 1 - Objects Hierarchy Diagram

Figure 2 - Objects Hierarchy at problem Creation

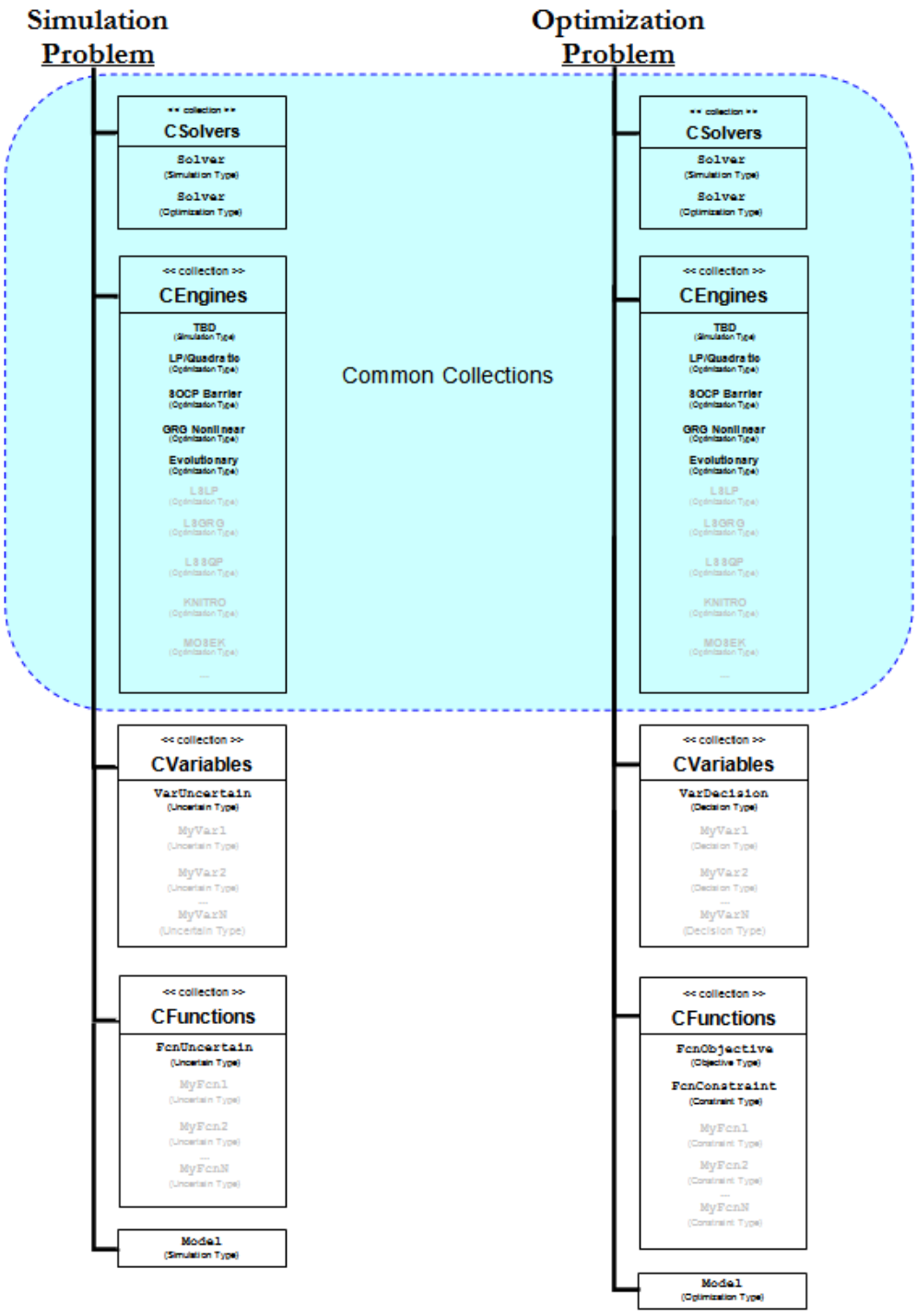


Figure 2 - Objects Hierarchy at Problem Creation

Index

Acronyms Used, iv
APPENDIX, 676
Callback Functions for Nonlinear Problems, 503, 550
class CDependMatrix, 75
class CDistribution, 105
class CDoubleMatrix, 64
class CEngine, 16
class CEngineCollection, 46
class CEngineLimit, 90
class CEngineParam, 101, 119
class CEngineParamCollection, 50
class CEngineStat, 87
class CEnvironment, 42
class CEvaluator, 21
class CEvaluatorCollection, 55
class CFunction, 37
class CFunctionCollection, 57
class CModel, 25
class COptIIS, 86
class CProblem, 6
class CSolver, 11
class CSolverCollection, 52
class CSolverException, 111, 113
class CStatistics, 83
class CVariable, 32
class CVariableCollection, 60
Classes and Enumerations Overview, 2
Code Formatting, ii
Collections Classes, 45
Concepts Formatting, ii
constat, 502, 533
Conventions and Acronyms Used, i
Cross-References, iii
Design Conventions, iii
Diagnostic Functions, 502, 535
Distribution (class), 105
Distribution Arguments Function, 224
Distribution Arguments Functions, 463
Distribution Parameter Function, 223
Distribution Parameter Functions, 456
Double Parameters, 506, 580
DualTolerance SDK parameter, 104
Engine Method Functions, 215, 266
Engine Property Functions, 216, 273
enum APIerr, 145
enum Array_Order, 166
enum Cone_Type, 186
enum Cons_Rel, 188
enum Depend_Type, 154
enum Dist_VarFcn, 197, 199, 201, 203, 205, 207, 209, 210, 212
enum DistFit_Type
  enum DistFit_Type, 174
enum DistParam_Type, 194
enum Engine_Action, 164
enum Eval_Type, 162
enum File_Format, 168, 170, 174
enum Frequency_Type, 177
enum Function_Type, 160
enum IIS_Status, 190, 192, 194, 197, 199, 201, 203, 205, 207, 209, 210, 212
enum Integer_Type, 184
enum License_Loc, 150
enum Optimize_Status, 181
enum Problem_Type, 156
enum Simulate_Status, 179
enum Solver_Type, 152
enum Variable_Type, 158
Enumerations, 3, 143
Environment Method Functions, 222, 425
Evaluator Method Functions, 216, 284
findiis, 503, 536, 537, 539
Fitting Method Functions, 223, 445
Fitting Quality Criteria Functions, 223, 448
Formatting Conventions, ii
funceval, 283, 503, 514, 515, 516, 519, 520, 521, 551, 552, 553
Functions and Parameters Overview, 501
Functions Error Codes, 497
Functions Overview, 502
getcallbackinfo, 504, 560
getdblparam, 504, 568
getiis, 503, 536, 537, 538
getintparam, 504, 564
getlpcallbackfunc, 503, 557
getmipcallbackfunc, 504, 559
getproblimits, 503, 544, 549
getuse, 503, 546, 548

iiswrite, 503, 536, 539
infodblparam, 504, 569, 571
infointparam, 504, 565, 571
Initialization Method Function, 223, 455
Integer Parameters (General), 505, 573
Integer Parameters (Mixed-Integer Problems), 506, 578
Integer Parameters (Solver Engines), 505, 575
interface IEvaluator, 24
jacobian, 503, 514, 515, 516, 551, 553, 575, 576
Legal and Ordering Information, 1
Linear Problems, 584
Linear Problems Definition, 584
Linear Problems Examples, 584
loadctype, 502, 513, 524, 540, 542, 543, 578
loadlp, 502, 509, 511, 513, 514, 516, 519, 522, 524, 525, 526, 529, 530, 536, 537, 539, 540, 541, 542, 543, 544, 546, 547, 548, 556, 557, 558, 559
loadnlp, 502, 509, 513, 514, 516, 517, 519, 520, 522, 524, 525, 526, 531, 533, 536, 537, 540, 546, 547, 548, 551, 553, 554, 576
loadnltype, 502, 514, 515, 517, 519, 554
loaquad, 511
lpread, 503, 540, 542, 543, 544
lpwrite, 503, 539, 540, 541, 542
Mathematical Method Functions, 220, 380
Mathematical Property Functions, 221, 387
Matrix Classes, 64
Method and Property Functions Overview, 214
mipoptimize, 502, 525, 526, 546, 547, 556, 578, 580
Model Method Functions, 216, 287
Model Property Functions, 216, 290
Nonlinear Problems, 617
Nonlinear Problems Definition, 617
Nonlinear Problems Examples, 617
Nonsmooth Problems, 642
Nonsmooth Problems Definition, 642
Nonsmooth Problems Examples, 642
Objects Hierarchy (diagram), 679
Objects Hierarchy at problem Creation (diagram), 679
objsa, 502, 511, 512, 529
Optimization Problems Reference and Examples, 583
optimize, 502, 524, 525, 526, 546, 547, 551, 556, 574, 577, 580
Other Callback Functions, 503, 555
Parameter Management Functions, 504, 562
PrimalTolerance SDK parameter, 104
Primary Classes, 2, 5
Probability Distribution Method Functions, 222, 437
Problem Definition Functions, 502, 508
Problem Method Functions, 214, 225
Problem Property Functions, 214, 232
PsiSemiVar, 370, 372, 373, 374, 375, 377, 378, 379, 410, 411, 412, 413, 414, 416, 417, 418, 419, 420, 421, 422, 423, 424
PsiSimParam, 125
PsiStdDevCI, 377, 378, 379, 416, 417, 418, 419, 420, 421, 422, 423, 424
Quadratic Problems, 595
Quadratic Problems Definition, 595
Quadratic Problems Examples, 595
Quadratically Constrained Problems, 627
Quadratically Constrained Problems Definition, 627
Quadratically Constrained Problems Examples, 627
reportuse, 503, 546, 548
rhssa, 502, 511, 512, 530
Sample Management Function, 224
Sample Management Functions, 468, 473, 491
Secondary Classes, 2, 63
setdblparam, 504, 567
setdefaults, 504, 571
setintparam, 504, 563
setlpcallbackfunc, 503, 556, 557
setmipcallbackfunc, 503, 558, 559
Simulation Definition, 653
Simulation Distribution Arguments Functions, 222, 442
Simulation Problem Examples, 653
Simulation Reference and Code Examples, 653

solution, iii, 2, 15, 21, 23, 32, 37, 152, 160, 162, 181, 182, 215, 220, 257, 258, 283, 384, 502, 505, 506, 512, 515, 517, 524, 525, 526, 528, 536, 539, 560, 574, 575, 576, 577, 578, 579, 580, 581
Solution Functions, 502, 523
Solver Method Functions, 214, 239
Solver Parameters, 505, 572
Solver Parameters Overview, 505
Solver Platform SDK Procedural API Reference, 213
Solver Platform SDK Legacy API Reference, 500
Solver Platform SDK Object-Oriented API Reference, 1
Solver Property Functions, 215, 247
Solver SDK Object-Oriented API Objects Hierarchy, 676
SolverEngCount, 216, 274
SolverEngineAdd, 215, 267
SolverEngineDiscover, 269
SolverEngineRemove, 215, 268, 269
SolverEngLimit, 216, 282
SolverEngName, 216, 275
SolverEngParamCount, 216, 277
SolverEngParamGet, 216, 279
SolverEngParamInfo, 216, 281
SolverEngParamLoad, 215, 270
SolverEngParamName, 216, 278
SolverEngParamReset, 215, 272
SolverEngParamSave, 215, 271
SolverEngParamSet, 216, 280
SolverEngStat, 216, 283
SolverEngTrialPeriod, 433, 434, 435, 436
SolverEngType, 216, 276
SolverEnvFree, 427
SolverEnvInit, 426
SolverEnvLicenseGet, 431
SolverEnvLicenseSet, 432
SolverEnvLockCodeGet, 430
SolverEnvRuntimeIDGet, 428
SolverEnvRuntimeIDSet, 429
SolverEvaluatorAdd, 216, 285
SolverEvaluatorRemove, 216, 286
SolverFcnAllTrials, 221, 400
SolverFcnCount, 221, 388
SolverFcnDualLower, 221, 407
SolverFcnDualUpper, 369, 370, 372, 373, 374, 375, 377, 378, 379, 408, 409, 410, 411, 412, 413, 414, 416, 417, 418, 419, 420, 421, 422, 423, 424
SolverFcnDualValue, 221, 406
SolverFcnFinalValue, 221, 405
SolverFcnGetCorrelation, 220, 383
SolverFcnGetFrequency, 220, 384
SolverFcnLowerBoundGet, 221, 401
SolverFcnLowerBoundSet, 221, 402
SolverFcnNameGet, 221, 389
SolverFcnNameSet, 221, 389, 390
SolverFcnNonNegative, 220, 385
SolverFcnPercentiles, 221, 399
SolverFcnPosition, 221, 397
SolverFcnRelation, 220, 386
SolverFcnSizeGet, 221, 395
SolverFcnSizeSet, 221, 396
SolverFcnStatistics, 221, 398
SolverFcnTypeGet, 221, 391
SolverFcnTypeSet, 221, 392
SolverFcnUpperBoundGet, 221, 403
SolverFcnUpperBoundSet, 221, 404
SolverFcnValueGet, 221, 393
SolverFcnValueSet, 221, 394
SolverFunctionAdd, 220, 381
SolverFunctionCurrent, 215, 259
SolverFunctionIndex, 215, 261
SolverFunctionRemove, 220, 382
SolverFunctionType, 215, 260
SolverModAllGradDependGet, 217, 301
SolverModAllGradDependSet, 217, 302
SolverModAllGradientGet, 217, 303
SolverModAllGradientSet, 217, 304
SolverModAllLinearGet, 217, 297, 299
SolverModAllLinearSet, 217, 298
SolverModAllSpGradientGet, 217, 305
SolverModAllSpGradientSet, 217, 306
SolverModAllSpLinearGet, 217, 297, 299, 305
SolverModAllSpLinearSet, 217, 300
SolverModArrayOrder, 216, 288
SolverModDependTest, 216, 289
SolverModDistArgumentsGet, 218, 320
SolverModDistArgumentsSet, 218, 321
SolverModDistAutoFit, 438
SolverModDistCDF, 439
SolverModDistCDFInv, 440
SolverModDistControlParamsGet, 443
SolverModDistControlParamsSet, 444

SolverModDistCorrelationsGet, 218, 325
SolverModDistCorrelationsSet, 218, 326
SolverModDistFit, 446
SolverModDistFitLimit, 449
SolverModDistFitSigLevel, 450
SolverModDistFitStatistic, 451
SolverModDistFitSuccess, 452
SolverModDistFitTest, 453
SolverModDistIndepTestGet, 454
SolverModDistInit, 455
SolverModDistInitSample, 447
SolverModDistKurtosis, 457
SolverModDistMean, 458
SolverModDistMedian, 459
SolverModDistMode, 460
SolverModDistNameGet, 319
SolverModDistNumControlParams, 464
SolverModDistNumParams, 465
SolverModDistParamsGet, 466
SolverModDistParamsSet, 467
SolverModDistPDF, 441
SolverModDistSampleGet, 469
SolverModDistSampleSet, 470
SolverModDistSizeGet, 471
SolverModDistSizeSet, 472, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 492, 493, 494, 495, 496
SolverModDistSkewness, 461
SolverModDistSpArgumentsGet, 218, 322
SolverModDistSpArgumentsSet, 218, 323, 324
SolverModDistTypeGet, 218, 317
SolverModDistTypeSet, 218, 318, 319
SolverModDistVariance, 462
SolverModFcnHessDependGet, 218, 315
SolverModFcnHessDependSet, 218, 316
SolverModFcnHessianGet, 217, 307
SolverModFcnHessianSet, 217, 308
SolverModFcnIndexName, 218, 328
SolverModFcnQuadraticGet, 217, 311
SolverModFcnQuadraticSet, 218, 312
SolverModFcnSpHessianGet, 217, 309
SolverModFcnSpHessianSet, 217, 310
SolverModFcnSpQuadraticGet, 218, 313
SolverModFcnSpQuadraticSet, 218, 314
SolverModNameFcnIndex, 218, 330
SolverModNameVarIndex, 218, 329
SolverModNumAllGrad, 217, 293
SolverModNumCorrelations, 217, 296
SolverModNumDistArguments, 217, 295
SolverModNumFcnHess, 217, 294
SolverModNumFunctions, 217, 292
SolverModNumVariables, 216, 291
SolverModVarIndexName, 218, 327
SolverNumSimulationsGet, 215, 248
SolverNumSimulationsSet, 215, 249
SolverNumSolutionsGet, 215, 255
SolverNumSolutionsSet, 215, 256
SolverNumTrialsGet, 215, 250, 251
SolverNumTrialsSet, 215, 251
SolverOptIIS, 215, 254
SolverOptIISFind, 214, 242
SolverOptimize, 214, 241
SolverOptimizeStatus, 215, 253
SolverProbEngineGet, 214, 237
SolverProbEngineSet, 214, 238
SolverProbFree, 214, 228
SolverProbInit, 214, 227
SolverProblem, 214, 226, 231, 338, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 441
SolverProbLoad, 214, 229
SolverProbNameGet, 214, 235
SolverProbNameSet, 214, 236
SolverProbSave, 214, 230
SolverProbTypeGet, 214, 233, 234
SolverProbTypeSet, 214, 234
SolverReport, 215, 243, 244, 245, 246, 262, 263, 264, 265
SolverSimulate, 214, 240
SolverSimulateStatus, 215, 252
SolverSolutionsIndexGet, 215, 257
SolverSolutionsIndexSet, 258
SolverVarAllTrialsGet, 219, 349
SolverVarAllTrialsSet, 219, 350
SolverVarConeIndexGet, 220, 361
SolverVarConeIndexSet, 220, 362
SolverVarConeTypeGet, 220, 359
SolverVarConeTypeSet, 220, 360
SolverVarCount, 219, 338
SolverVarDualLower, 220, 367
SolverVarDualUpper, 220, 368

SolverVarDualValue, 220, 366
SolverVarFinalValue, 220, 365
SolverVarGetCorrelation, 219, 334
SolverVarGetFrequency, 219, 335
SolverVarGroupIndexGet, 220, 357
SolverVarGroupIndexSet, 220, 358
SolverVariableAdd, 218, 332
SolverVariableRemove, 218, 333
SolverVarInitialValueGet, 220, 363
SolverVarInitialValueSet, 220, 364
SolverVarIntegerTypeGet, 219, 355
SolverVarIntegerTypeSet, 220, 356
SolverVarLowerBoundGet, 219, 351
SolverVarLowerBoundSet, 219, 352
SolverVarNameGet, 219, 339
SolverVarNameSet, 219, 340
SolverVarNonNegative, 219, 336
SolverVarPercentiles, 219, 348
SolverVarPosition, 219, 346
SolverVarSizeGet, 219, 344
SolverVarSizeSet, 219, 345
SolverVarStatistics, 219, 347
SolverVarTypeGet, 219, 341
SolverVarTypeSet, 219, 342
SolverVarUpperBoundGet, 219, 353
SolverVarUpperBoundSet, 219, 354
SolverVarValue, 219, 343
testnltype, 502, 519, 520, 521
unloadprob, 502, 509, 516, 519, 522, 540
Use Control Functions, 503, 545
Variable Method Functions, 218, 331
Variable Property Functions, 219, 337
varstat, 502, 531