



Solver SDK User's Guide

By Frontline Systems, Inc.

```
#include "solversdk.h"
#include "stdafx.h"

using namespace SolverPlatform;

class CNLPEvaluator: public IEvaluator
{
Public:
    Virtual _Engine_Action Evaluate (EEvaluator& evaluator)
    {
        CProblem& problem = evaluator.Problem;
        Const double* pVar = problem.VarDecision.Value;
        Double* pFcn = problem.FcnConstraint.Value;
        pFcn[0] = pVar[0] + pVar[1];
        pFcn[1] = pVar[0] * pVar[1];
        problem.FcnObjective.Value[problem.ObjectiveIndex] = pVar[0] * pVar[1] +
        pVar[1] * pVar[1];
        return Engine_Action_Continue;
    }
}

int main() /* Solve example constrained nonlinear optimization problem */
{
    int nvars = 2, ncons = 2;
    double rhs[2] = { 1.0, CConstants::PINF };
    double lhs[2] = { 1.0, 0 };
    double lb[2] = { 0, 0 };
    double ub[2] = { 1, 1 };
    CProblem problem(Solver_Type_Minimize, nvars, ncons);
    problem.FcnConstraint.UpperBound = rhs;
    problem.FcnConstraint.LowerBound = lhs;
    problem.VarDecision.UpperBound = ub;
    problem.VarDecision.LowerBound = lb;
    CNLPEvaluator evalNLP;
    problem.Evaluators[Eval_Type_Function].Evaluator = &evalNLP;
    problem.ProblemType = Problem_Type_OptNLP;
    problem.Solver.Optimize
    _Optimize_Status status = problem.Solver.OptimizeStatus;
    const double objvalue = problem.FcnObjective.FinalValue[0];
    const double* varvalue = problem.VarDecision.FinalValue;
```

Copyright

Software copyright 1991-2025 by Frontline Systems, Inc.

User Guide copyright 2025 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™, and Solver SDK™ 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 Ziena Optimization, Inc. MOSEK is a trademark of MOSEK ApS. OptQuest is a trademark of OptTek Systems, Inc. Xpress^{MP} is a trademark of FICO, Inc.

Patent Pending

Systems and Methods for Automated Risk Analysis of Machine Learning Models.

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>

Table of Contents

Table of Contents	iii
Introduction	1
Key Benefits of Solver SDK	1
Solver SDK Platform Overview of Features	2
Loading and Solving Excel Models	2
Exploiting and Controlling Multiple Cores	2
Supports Multiple Users Concurrently	2
Built-In Solver SDK Platform Engines	2
Large-Scale Solver Engines	3
Solver SDK Pro Overview of Features	3
Exploiting and Controlling Multiple Cores	4
Supports Multiple Users Concurrently	4
Built-In Solver SDK Pro Engines	4
How to Use This Guide	4
What's New in Analytic Solver V2025 Q1	5
Usability: New Flexible Dialogs	5
Performance: New Nonlinear Solver Power	5
Standard Excel Formulas, New Solving Speed	6
What's New in Analytic Solver V2025 Q2	7
Faster When Using, Not Just Solving	7
Enhanced Excel Support, Smarter Human Support	7
New Support for Metalog2 Distributions	7
What's New in Analytic Solver V2025 Q4	8
Semantic Search Without Using AI	8
A Shorter Ribbon, and Other Simplifications	9
Easier to Find Distribution Fitting	9
Easier to Find "LP or MPS Model Format"	10
Installation	11
What you need	11
Getting Help	11
Installing the Software	11
Logging in the First Time	14
Uninstalling the Software	15
Licensing the Solver SDK	17
Important Considerations	17
Single User Licenses	17
Multiple User Licenses	18

Runtime Licensing Programs.....	18
Frequently Asked Questions	20
Why do I keep getting the error, "The specified module could not be found"?	20
Why can't the Solver SDK find my license file?.....	20

Designing Your Application 22

Calling the Solver SDK	22
Loading and Solving RASON and Excel Models	22
Analyzing and Solving Excel Models in the SDK Platform.....	23
Input Objects	24
Parameter Objects.....	24
Solving Linear and Quadratic Optimization Problems	24
Solving Nonlinear Optimization Problems	25
Solving Nonsmooth Optimization Problems.....	26
Genetic and Evolutionary Algorithms	28
Optimization Problems in Algebraic Notation.....	29
Saving Models in RASON format	29
Setting the Problem Type	30
Selecting an Optimization Engine	30
Selecting an Engine in the SDK	30
Selecting Large Scale Solver Engines.....	31
Determining Linearity Automatically in Optimization Models	32
Diagnosing Infeasible Optimization Problems.....	33
Solution Properties of Quadratic Optimization Problems.....	34
Solving Simulation Problems.....	34
Passing Sparse Matrices	36
DoubleMatrix Object	36
DoubleMatrix Methods.....	36
Using the Solver SDK in Multi-Threaded Applications	37
Memory Usage.....	38

Solver SDK Evaluators 39

Introduction	39
Using an Evaluator to Check the Progress of an Optimization	39
Optimization Progress Information Evaluators	39
Using a Function Evaluator to Compute the Objective and Constraints.....	41
Complete Evaluator	42
Using a Gradient Evaluator to Provide First Order Partial Derivatives	43
Complete Evaluator	44
Using a Hessian to Provide Second Order Partial Derivatives.....	45
Solving a Quadratic Problem with a Hessian Evalauter.....	45
Complete Evaluator	48
Solving a Quadratically Constrained Problem with a Hessian Evaluator.....	48
Complete Evaluator	51
Solving a Nonlinear Problem with a Hessian Evalauter	52
Complete Evaluator	54
Using an Evaluator to Check the Progress of a Simulation Run	55
Simulation Progress Information Evaluators.....	55
Using an Evaluator to Compute Uncertain Functions in a Simulation Problem.....	56
Complete Evaluator	57
Conclusion.....	57

Calling the Solver SDK from C++ 59

C++ Compilers.....	59

Basic Steps: Solving Three Simple Models	59
Creating a New C++ Project	59
Creating a Linear Model.....	61
Creating a Nonlinear Problem.....	65
Solving a Simulation Problem.....	70
C/C++ Source Code.....	75
Calling the Solver SDK from .NET	79
Introduction.....	79
Creating a New C# Project.....	79
Creating a Linear Model	81
Creating a Nonlinear Problem	86
Creating a Simulation Model.....	91
Creating a New F# Project	97
Creating a Linear Model in F#.....	99
Creating a Nonlinear Problem	101
Creating a Simulation Model.....	103
Creating a New VB.NET Project	105
Creating a Linear Model	107
Creating a Nonlinear Problem	112
Creating a Simulation Model	118
Calling the Solver SDK from JAVA	124
Introduction.....	124
Creating a Linear Model.....	125
Creating a Nonlinear Model.....	128
Solving a Simulation Problem.....	131
Calling the Solver SDK from Python	135
Introduction.....	135
Creating a Linear Model.....	136
Creating a Non-Linear Model	138
Solving a Simulation Problem.....	140
Calling the Solver SDK from R	144
Introduction.....	144
Creating a Linear Model.....	144
Creating a Non-Linear Model	148
Solving a Simulation Problem.....	153
Calling the Solver SDK from C using the Procedural API	159
Introduction.....	159
Creating a New C Project	159
Creating a linear model.....	161
Creating a Nonlinear Problem.....	165
Creating a Simulation Model	169
Calling the Solver SDK from VB6 using the COM API	177
Introduction.....	177
Creating a New VB Project.....	177
Creating a Linear Model.....	179

Creating a Nonlinear Model.....	183
Creating a Simulation Model.....	188
Calling Solver SDK from VBA	194
Visual Basic 6 Source Code.....	194

From Excel to SDK 198

Introduction	198
Optimization Problem Type.....	198
Converting a Linear Example	198
Converting a Nonlinear Example	202
Converting a Simulation Example.....	206

PSI Function Reference 211

Using PSI Functions.....	211
Using PSI Distribution Functions.....	211
Continuous Analytic Distributions.....	212
PsiBeta.....	212
PsiBetaGen.....	214
PsiBetaSubj.....	215
PsiBurr12.....	216
PsiCauchy.....	217
PsiChiSquare.....	218
PsiDagum.....	219
PsiDblTriang.....	220
PsiErf.....	221
PsiErlang.....	222
PsiExponential.....	223
PsiFatigueLife.....	223
PsiFDist.....	224
PsiFrechet.....	225
PsiGamma.....	226
PsiHypSecant.....	227
PsiInvNormal.....	228
PsiJohnsonSB.....	229
PsiJohnsonSU.....	230
PsiKumaraswamy.....	230
PsiLaplace.....	231
PsiLevy.....	232
PsiLaplace.....	233
PsiLogistic.....	234
PsiLogLogistic.....	235
PsiLogNormal.....	236
PsiLogNorm2.....	237
PsiMaxExtreme.....	238
PsiMetalog.....	239
PsiMetalog2.....	240
PsiMetalogFit.....	241
PsiMetalog2Fit.....	242
PsiMetalogSPT.....	243
PsiMinExtreme.....	244
PsiMyerson.....	245
PsiNormal.....	248
PsiPareto.....	248
PsiPareto2.....	249

PsiPearson5	250
PsiPearson6	252
PsiPert	253
PsiRayleigh	254
PsiReciprocal	255
PsiStudent	256
PsiTriangGen	257
PsiTriangular	259
PsiUniform	260
PsiWeibull	261
Discrete Analytic Distributions	263
PsiBernoulli	263
PsiBinomial	264
PsiGeometric	265
PsiHyperGeo	267
PsiIntUniform	268
PsiLogarithmic	269
PsiNegBinomial	270
PsiPoisson	271
Custom Distributions	273
PsiCumul	273
PsiCumulD	274
PsiDiscrete	274
PsiDisUniform	275
PsiGeneral	276
PsiHistogram	277
PsiMetalog	278
PsiMetalogSPT	280
Stochastic Information Packets (SIPs)	281
Creating a SIP/SLURP	281
Creating a SIP/SLURP from a Data Source	281
Creating a SIP/SLURP from a Psi Distribution	285
Dependence and Correlation	287
Measuring Observed Correlation	287
Inducing Correlation Among Uncertain Variables	288
GetCorrelation	289
GetFrequency	290
PSI Statistics Functions	290
CITrials	291
CoeffVar	291
CondValueAtRisk	291
ExpGain	291
ExpGainRatio	291
ExpLoss	292
ExpLossRatio	292
ExpValMargin	292
Kurtosis	292
Maximum	293
Mean	293
MeanAbsDev	293
MeanCI	293
Median	294
Minimum	294
Mode	294
NumErrors	294
NumValues	295

Percentile	295
Range	295
SemiDeviation	295
SemiVariance	295
SigmaCP	296
SigmaCPK	296
SigmaCPKLower	296
SigmaCPKUpper	297
SigmaCPM	297
SigmaDefectPPM	297
SigmaDefectShiftPPM	297
SigmaDefectShiftPPMLower	298
SigmaDefectShiftPPMUpper	298
SigmaK	298
SigmaLowerBound	298
SigmaProbDefectShift	299
SigmaProbDefectShiftLower	299
SigmaProbDefectShiftUpper	299
SigmaSigmaLevel	299
SigmaUpperBound	300
SigmaYield	300
SigmaZLower	300
SigmaZMin	301
SigmaZUpper	301
Skewness	301
StdDev	301
StdDevCI	302
Target	302
ValueAtRisk	302
Variance	303
Distribution Control Parameters	303
SDK Simulation Parameters	304
CensorType	304
Cutoff Type	304
Run Specific Simulation	305
Accessing Statistics from Different Simulations	306
Distribution Fitting	306
AutoFit	307
FitSuccess	307
AutoFitAll	307
Fit	308
IndepTest	308
FitStatistic	308
FitTest	308
FitLimit	308
Risk Solver Engine Options	309
Random Seed	309
CLT Threshold	311
Psi Functions for Analytic Solver	312
Unsupported Excel Functions	312

Solver Result Messages

313

Introduction	313
Simulation Status Result Codes	313
Optimization Status Result codes	314

Problems with Poorly Scaled Models	321
Dealing with Poor Scaling.....	322
Limitations on Smooth Nonlinear Optimization	322
GRG Solver Stopping Conditions.....	322
GRG Solver with Multistart Methods	323
Nonlinear GRG Solver and Integer Constraints.....	324
Limitations on Non-Smooth Optimization.....	324
Effect on the Nonlinear GRG and Simplex Solvers.....	324
Evolutionary Solver Stopping Conditions.....	324

Solver SDK Parameters

327

Setting the Parameters	327
Common SDK Optimization Parameters	327
Iterations	327
Precision.....	328
Use Automatic Scaling.....	328
Run Specific Optimization	329
LP Simplex Solver Options.....	330
Pivot Tolerance.....	330
Reduced Cost Tolerance	330
Solution Tolerance.....	330
LP Simplex Solver MIP Parameters	330
Max Subproblems.....	331
Max Feasible (Integer) Solutions.....	331
Integer Tolerance	331
Integer Cutoff.....	332
Use Dual Simplex for Subproblems.....	332
Preprocessing and Probing	333
Cut Generation	335
Solve Without Integer Constraints.....	336
LP/Quadratic Solver Options	336
Primal Tolerance.....	337
Dual Tolerance	337
Do Presolve	337
Derivatives for the Quadratic Solver.....	337
Solve Without Integer Constraints.....	338
LP/Quadratic Solver MIP Parameters.....	338
Max Subproblems.....	338
Max Feasible (Integer) Solutions.....	338
Integer Tolerance	339
Integer Cutoff.....	339
Preprocessing	340
Cuts.....	340
Heuristics	341
Thread Mode	341
LP/Quadratic Solver Stochastic Decomposition Parameters.....	341
Tau.....	341
Tolerance.....	342
Compute Confidence Interval.....	342
Objective Error	342
Objective Improvement.....	342
Compute Recourse Statistics	343
SOC Barrier Solver Parameters.....	343
Gap Tolerance	343
Step Size Factor.....	343

Feasibility Tolerance	343
Search Direction	344
Power Index	344
SOCP Integer Options	344
Max Subproblems.....	344
Max Feasible (Integer) Solutions.....	345
Integer Tolerance.....	345
Integer Cutoff.....	346
Convergence.....	346
Recognize Linear Variables	347
Derivatives and Other Nonlinear Options	347
Large Scale GRG Nonlinear Solver Parameters.....	349
Convergence.....	349
Recognize Linear Variables	349
Relax Bounds on Variables.....	350
Derivatives and Other Nonlinear Options	350
Multistart Search Parameters	352
Multistart Search	352
Topographic Search.....	353
Require Bounds on Variables.....	353
Population Size.....	353
Random Seed	354
Evolutionary Solver Parameters.....	354
Convergence.....	354
Population Size.....	355
Mutation Rate.....	355
Random Seed	356
Require Bounds on Variables.....	356
Local Search.....	356
Global Search.....	358
Model Based Search	358
Feasibility Pump / FeasibilityPump	359
Evolutionary Parameters for Integer Problems.....	359
Max Subproblems.....	359
Max Feasible Solutions.....	360
Tolerance	360
Max Time without Improvement.....	360
Solve Without Integer Constraints.....	360
Mixed Integer Problem Parameters	361
Max Subproblems.....	361
Max Feasible (Integer) Solutions.....	361
Integer Tolerance.....	361
Integer Cutoff.....	362
Solve Without Integer Constraints.....	362

Using Solver SDK Dedicated Server

367

Introduction	367
Installing the Software Via Installation Program	367
Using the REST API	369
Responses: Solution Status and Results	370
Unauthorized Errors.....	370
Synchronous Quick Calls to REST API Endpoints	371
Asynchronous Rason REST API Endpoints.....	376
Solving a RASON Model using Solver SDK Dedicated Server.....	383
Introduction.....	383

Solving an Optimization	383
RASON Server URL.....	384
Authorization.....	384
Simulation Example.....	392
Setting Up the Rason Model.....	393
Simulation Optimization Example.....	397
Setting Up the Rason Model.....	398
Managing Solver SDK Dedicated Server.....	402
Introduction	402
Using Solver SDK Dedicated Server Administrative Program.....	402
Settings.....	403
Solver SDK Dedicated Server Workflow.....	406
Servers	406
Jobs.....	408
Users	409
Certificates	410
Creating a New Certificate	410
Editing an Existing Certificate.....	411
Server Log.....	412
Client Sessions	412
Client Tasks.....	413
Licenses	413
Engines.....	414
Notable Files and Subfolders.....	414
Environment Variable Example Code	415
More Information	416
Solving a Linear Model Using PHP	416
Passing a Sparse Coefficient Matrix	420

Appendix 421

Setting the Problem Type.....	421
C#	421
Procedural API in C.....	421
COM in VB.....	421
Selecting the Engine	421
C#	421
Procedural API in C.....	421
COM in VB.....	422
Selecting a Large Scale Engine	422
C#	422
Procedural API in C.....	423
COM in VB.....	424
Using Discovery Method	425
C#	425
Procedural API in C.....	425
COM in VB.....	425
Using an Evaluator to Check the Progress of an Optimization	426
C#	426
Procedural API in C.....	426
COM in VB.....	427
Using a Function Evaluator to Compute the Objective and Constraints	427
C#	427
Procedural API in C.....	428
COM in VB.....	428
Using a Gradient Evaluator to Provide First Order Partial Derivatives	429

C#	429
Procedural API in C.....	429
COM in VB.....	430
Using a Hessian Evaluator to Provide Second Order Partial Derivatives	431
Solving a Quadratic Problem with a Hessian Evaluator.....	431
C#	431
Procedural API in C.....	432
COM in VB.....	433
Solving a Quadratically Constrained Problem with a Hessian Evaluator.....	436
C#	436
Procedural API in C.....	437
COM in VB.....	440
Solving a Nonlinear Problem with a Hessian Evaluator	442
C#	442
Procedural API in C.....	443
COM in VB.....	444
Using an Evaluator to Check the Progress of a Simulation Run	445
C#	445
Procedural API in C.....	445
COM in VB.....	445
Using an Evaluator to Compute Uncertain Functions in a Simulation Problem.....	446
C#	446
Procedural API in C.....	446
COM in VB.....	447
Stochastic Information Packets	447
C# - Creating a SIP from a Data Source	447
C# - Creating a SIP inside the Evaluator.....	449
C# - Creating a SIP from a PsiDistribution	450
Procedural API - Creating a SIP from a Data Source	451
Procedural API - Creating a Sip inside the Evaluator	453
Procedural API - Creating a SIP from a PsiDistribution	454
COM - Creating a SIP from a Data Source	456
COM - Creating a SIP from a PsiDistribution.....	458
GetCorrelation	460
C#	460
Procedural API.....	460
COM in VB.....	461
GetFrequency.....	461
C#	461
Procedural API.....	461
COM in VB.....	461
Calling IISFind.....	462
C#	462
Procedural API.....	462
COM.....	463

Introduction

Welcome to **Frontline Systems' Solver SDK Comprehensive** which combines **Solver SDK** and **XLMiner SDK** to provide the tools you need to rapidly prototype and deploy optimization, Monte Carlo simulation, forecasting and data science models on desktops, servers, or in the Cloud.

- This guide covers Solver SDK for solving optimization, simulation, stochastic optimization and simulation optimization models.
- The Solver SDK Data Science User Guide covers forecasting and datamining functionality.

Key Benefits of Solver SDK

Solver SDK comes in three versions: Comprehensive, Platform and Pro. The Solver SDK Platform's four bundled optimization engines handle linear, quadratic, nonlinear, and nonsmooth problems. In addition, the SOCP Barrier Solver solves new second order cone programming (SOCP) problems to obtain robust solutions to problems in finance and engineering design. It can also solve quadratically constrained (QCP) problems that go beyond conventional quadratic programming (QP). The Solver SDK Pro's three bundled optimization engines handle linear, nonlinear and nonsmooth problems. (Quadratic programming only supported in the Solver SDK Platform.)

The powerful Monte Carlo Simulation engine in the both the Solver SDK Platform and the Solver SDK Pro, provides everything you need to define and solve risk analysis problems, with 40 probability distributions and 30 statistics and risk measures, plus percentiles. With the Solver SDK's optimization engines plus its Monte Carlo simulation engine, you can easily define and solve simulation optimization problems.

The SDK can be called from a program you write in any programming language, macro language, or scripting language for 64-bit Microsoft Windows, or Mac OSX that is capable of calling a DLL (Dynamic Link Library). Some of the possibilities that will be covered in this User Guide are Microsoft COM and .NET, as well as C/C++, and Visual Basic. There are many others, including Visual Basic Application Edition in the Microsoft Office application.

In addition to the Object-Oriented API and Procedural API, the Solver SDK includes a Legacy API that is fully upward compatible with the API supported by the former Solver DLL and Solver DLL Platform products.

If your models are larger than the limits of the Solver SDK's bundled engines, or if you want even faster solution times on problems within the bundled engines' limits, you can expand the Solver SDK Platform with field-installable Solver engines. These engines allow the SDK to solve models of virtually unlimited size. We've integrated the world's best Solvers, to handle the largest linear (with millions of variables), quadratic, conic, nonlinear and mixed integer programming problems. Note: Field-installable engines are not supported for Solver SDK Pro.)

Solver SDK also includes the capability to load Excel workbooks with optimization and simulation models (created using one of our Excel products) . Excel is required to create your model, but it is not required on the target system - everything you need is contained within Solver SDK.

Solver SDK makes it easy to build dashboards or interfaces over your optimization or Monte Carlo simulation model to ensure ease of use for your end users. *Visual Studio Wizards* help you get started right away, and *IntelliSense* helps you write correct code quickly instead of searching through manuals.

Solver SDK Platform Overview of Features

The five built-in engines of the Solver SDK Platform are able to solve linear programming (LP), quadratic programming (QP) and mixed integer programming (MIP) problems of up to 8,000 variables (2,000 integers) and 8,000 constraints; quadratically constrained programming (QCP) problems of up to 2,000 decision variables and 8,000 constraints; smooth nonlinear programming (NLP) problems of up to 1,000 variables and 1,000 constraints; and nonsmooth optimization problems (NSP) of up to 1,000 decision variables and 1,000 constraints (in addition to bounds on the variables). The Monte Carlo Simulation engine is able to solve simulation problems with an unlimited number of uncertain variables and functions and simulation optimization problems with up to 1,000 variables and 1,000 constraints.

Solver SDK Platform includes an enhanced version of the linear, mixed integer and nonlinear programming "engine" used in the Microsoft Excel Solver, which was developed by Frontline Systems for Microsoft. This Solver has been proven in use since 1991, in tens of millions of copies of Microsoft Excel. Microsoft selected Frontline Systems' technology over several alternatives because it offered the best combination of robust solution methods, performance, and ease of use.

Loading and Solving Excel Models

Solver SDK includes the ability to load and solve optimization and simulation models created with any of our Excel products, including the Excel Solver. Excel is required to *create* your model, but it's *not required* on the target system; everything you need is contained with SolverSDK.dll.

Exploiting and Controlling Multiple Cores

Solver SDK uses Windows facilities to create multiple "threads" of execution that can be run on different processor cores. If you have other compute-intensive applications running at the same time as the Solver SDK, you can control the number of cores used, separately for optimization and simulation, using just one line of code.

Supports Multiple Users Concurrently

Solver SDK is fully reentrant and suitable for multi-threaded applications, such as those on Intranet, Web server, or Web service applications that are designed to serve multiple users concurrently.

Built-In Solver SDK Platform Engines

The Solver SDK Platform includes five bundled Solver Engines to find solutions for the full spectrum of optimization problems, up to certain size limits:

- The **GRG Solver** handles smooth nonlinear (NLP) problems of up to 1,000 decision variables and 1,000 constraints, plus bounds on the variables, and is augmented with Multistart or "clustering" methods for global optimization. For some smooth nonlinear problems, these methods will converge in probability to the globally optimal solution. For other problems, they often yield very good solutions in an acceptable amount of time.
- The **Evolutionary Solver** is a hybrid of genetic and evolutionary algorithms and classical optimization methods. It handles non-smooth (NSP) problems of up to 1,000 decision variables

and 1,000 constraints, plus bounds on the variables. It is greatly enhanced with parallel algorithms in the Solver SDK.

- The **LP/Quadratic Solver** handles both linear programming (LP) and quadratic programming (QP) problems with up to 8,000 variables (2,000 integer variables) and constraints.
- The **SOCP Barrier Solver** uses an interior point method to solve linear (LP), quadratic (QP), quadratically constrained (QCP), and second order cone programming (SOCP) problems – the natural generalization of linear and quadratic programming – with up to 2,000 variables and 8,000 constraints. The SOCP Barrier Solver can be used to solve linear programming models with uncertainty via robust optimization with the L2 norm.
- The **Risk Solver** within the SDK brings full-featured Monte Carlo simulation capability for problems with an unlimited amount of uncertain variables or functions and simulation optimization capability for problems with up to 1,000 decision variables and 1,000 constraints. Each time a simulation is ran, thousands of trials are performed and a full range of simulation results and statistics are available with simulation speeds of up to 100 times faster than normal Excel-based Monte Carlo simulation. The Solver SDK Platform supports 45 analytic and custom distributions (continuous and discrete), 30 statistics and risk measures, rank order correlation, distribution fitting, multiple random number generators and sampling methods with variance reduction, and powerful capabilities for multiple parameterized simulations.
- Solver SDK Platform licenses include access to **RASON® – Restful Analytic Solver® Object Notation**. RASON is a modeling language embedded in **JSON** and a REST API that allows users to easily create, test and deploy **decision services** powered by **analytic models** in **web** and **mobile** applications – using business rules and DMN decision tables, optimization, simulation, forecasting and machine learning. For more information see www.RASON.com.

Large-Scale Solver Engines

Solver SDK Platform supports multiple, field-installable Solver Engines, in addition to the "bundled" Solver engines. (This feature is not supported in Solver SDK Pro.) Such Solver engines are licensed as separate products that provide additional power and capacity to solve problems much larger and/or more difficult than the problems handled by the bundled Solver engines. A license for a Solver Engine enables you to use that Solver in Excel, Matlab, Java, C/C++, C#, Visual Basic, VB.Net and other languages. 64-bit versions of these Solver Engines are available.

Field installable Solver engines are seamlessly integrated into the Solver SDK Platform. Trial licenses for these Solver engines are available, allowing you to evaluate how well they perform on a challenging Solver model that you've developed. To obtain a trial license, please contact our Sales department via email at sales@solver.com or by phone at 888-831-0333.

The large-scale Solver Engines are more valuable than ever in Solver SDK Platform, which can transform linear programming models with uncertainty into larger, conventional linear programming or second order cone programming models, using stochastic programming and robust optimization methods.

Each Solver engine supports the same easy-to-use API (application programming interface) as the Solver SDK Platform. With one line of code, you can add the plug-in Solver to the collection of available engines and then use it to solve a problem. You define your model, solve problems, write Evaluators (callback functions), and access the solution in the same way for all Solver engines. This makes it easy to test several Solver engines to find out which one performs best on your problems.

Solver SDK Pro Overview of Features

The four built-in engines of the Solver SDK Pro are able to solve linear programming (LP) and mixed integer programming (MIP) problems of up to 2,000 variables (1,000 integers) and 8,000 constraints;

smooth nonlinear programming (NLP) problems of up to 500 variables and 250 constraints; and nonsmooth optimization problems (NSP) of up to 500 decision variables and 250 constraints (in addition to bounds on the variables). The Monte Carlo Simulation engine is able to solve simulation problems with 2,000 uncertain variables and 8,000 uncertain functions and simulation optimization problems with up to 500 variables and 250 constraints.

Exploiting and Controlling Multiple Cores

Solver SDK Pro comprehensively exploits multi-core processors through parallel algorithms in Monte Carlo simulation, simulation optimization, global optimization, non-smooth optimization, and automatic differentiation.

The SDK Pro uses Windows facilities to create multiple "threads" of execution that can be run on different processor cores. If you have other compute-intensive applications running at the same time as the Solver SDK, you can control the number of cores used, separately for optimization and simulation, using just one line of code.

Supports Multiple Users Concurrently

Solver SDK Pro is fully reentrant and suitable for multi-threaded applications, such as those on Intranet, Web server, or Web service applications that are designed to serve multiple users concurrently.

Built-In Solver SDK Pro Engines

The Solver SDK Pro includes four bundled Solver Engines to find solutions for the full spectrum of optimization problems, up to certain size limits:

- The **Standard GRG Solver** handles smooth nonlinear (NLP) problems of up to 500 decision variables and 250 constraints, plus bounds on the variables, and is augmented with Multistart or "clustering" methods for global optimization. For some smooth nonlinear problems, these methods will converge in probability to the globally optimal solution. For other problems, they often yield very good solutions in an acceptable amount of time.
- The **Evolutionary Solver** is a hybrid of genetic and evolutionary algorithms and classical optimization methods. It handles non-smooth (NSP) problems of up to 500 decision variables and 250 constraints, plus bounds on the variables. It is greatly enhanced with parallel algorithms in Solver Platform Pro.
- The **LP Simplex Solver** handles linear programming (LP) problems with up to 2,000 variables (1,000 integer variables) and 8,000 constraints.
- The **Risk Solver** within the SDK Pro brings full-featured Monte Carlo simulation capability for problems with up to 2,000 uncertain variables and 8,000 uncertain functions and simulation optimization capability for problems with up to 500 decision variables and 250 constraints. Each time a simulation is ran, thousands of trials are performed and a full range of simulation results and statistics are available with simulation speeds of up to 100 times faster than normal Excel-based Monte Carlo simulation. The Solver SDK supports 45 analytic and custom distributions (continuous and discrete), 30 statistics and risk measures, rank order correlation, distribution fitting, multiple random number generators and sampling methods with variance reduction, and powerful capabilities for multiple parameterized simulations.

How to Use This Guide

This User Guide includes a chapter on installing the Solver SDK files, and a chapter providing an overview of the Solver SDK object oriented API calls and how they may be used to define and solve linear programming (LP), quadratic programming (QP and QCP – supported in the SDK Platform only), nonlinear programming (NLP), nonsmooth programming (NSP), mixed-integer programming

(MIP), Monte Carlo simulation and simulation optimization problems. It also includes chapters giving specific instructions for writing, compiling and linking, and testing application programs in four languages: C/C++, C#, and VB.NET.

We highly recommend that you read the installation instructions and the overview of the Solver SDK calls first. Then you may turn directly to the chapter that covers the language of your choice. Each chapter gives step-by-step instructions for building and testing simple example programs written in that language. You can use these example programs as a starting point for your own applications.

The example programs installed in the path, C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples, define and solve multiple types of optimization problems – LP (linear programming) problems, QP (quadratic programming – supported in the SDK Platform only) problems, NLP (nonlinear programming) problems, NSP (nonsmooth programming), Monte Carlo simulation and simulation optimization problems in seven different programming languages (C/C++, .NET (C# and VB), Matlab, Java, and Visual Basic 6.0) along with several Web examples.

What's New in Analytic Solver V2025 Q1

We've just released **Analytic Solver V2025 Q1**, with new versions of Analytic Solver Desktop and Cloud (for Excel users), Solver SDK (for developers), and our cloud platform RASON. This release features both **usability** improvements and a powerful new way to solve a **wide range** of **nonlinear** optimization models using the **Gurobi Solver Engine** -- traditionally known for solving linear (and quadratic) mixed-integer models. This new approach yields **globally optimal** solutions **every time** -- not possible with other nonlinear Solvers -- and often **much faster** solutions.

Usability: New Flexible Dialogs

Analytic Solver has always offered an easy user interface with an Office-style Ribbon, Task Pane, and dialogs that you can move around and resize. But the **content** of many dialogs used fixed-size fonts and graphics. To help with both **accessibility**, and better use of **larger, high-resolution screens** when you're "consuming" this content, in Analytic Solver V2025 Q1 we've revised the Task Pane and scores of other dialogs with "**zoomable**" content. You can now press the CTRL key, then use the mouse wheel to enlarge or shrink the text and graphics in almost any dialog or pane of Analytic Solver. At the bottom of this page is an example, where we've enlarged the text in the top and bottom parts of the Task Pane, but left the middle part (Model Diagnosis) unchanged – so you can see the difference.

Performance: New Nonlinear Solver Power

Analytic Solver V2025 Q1 features a *very special* level of support for the just-released (November 19) **Gurobi Solver 12.0**. This popular Solver again improves performance on linear programming (about +4%) and mixed-integer programming (about 13%) models, and even better performance (about +28%) on non-convex mixed-integer quadratic models. But most interesting for our Excel users is Gurobi's improved support for (often non-convex) **mixed-integer nonlinear** optimization problems. Such problems arise in specialty chemicals, hydroelectric power, and some aeronautics and space applications.

Unlike other nonlinear Solvers (such as our LSGRG, LSSQP and KNITRO Solvers), Gurobi takes a different approach, that can yield **globally optimally** solutions **IF** you can formulate your model with (only) certain algebraic expressions -- defined via special Python API calls -- that use just over a [dozen mathematical operators](#) (plus, minus, times, divide, square, square root, exponential and logarithm, etc.). If you're really good at Python coding and really good with algebraic manipulation, you can code this yourself. But in Analytic Solver V2025 Q1, **you don't have to do any extra work** to define your nonlinear model for the Gurobi Solver.

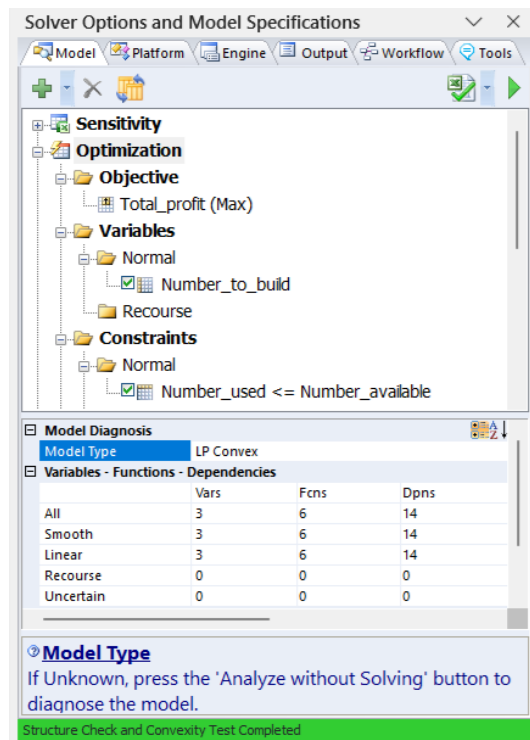
Standard Excel Formulas, New Solving Speed

You can write your model in Excel, using any of Excel's mathematical operators and any of Excel's 470+ built-in functions. Analytic Solver's **PSI Interpreter** will **analyze** your formulas (including formulas that depend on other formulas, to any level), **diagnose** your model as linear, quadratic, smooth nonlinear or non-smooth, and -- In V2025 Q1 -- tell you if your model is "nonlinear and **solvable via the new Gurobi Solver**". Analytic Solver can generate the internal "expression trees" that the Gurobi Solver requires.

Analytic Solver will even expand various Excel functions, based on their mathematical definitions, into chains of "Gurobi-supported primitive operators". For example, you can just go ahead and use the STDEV Excel function -- Analytic Solver will recognize this as "Gurobi-eligible" and create (internally) an expression tree using square, plus and square root operators, based on the definition of "standard deviation". The bottom line: With Analytic Solver, you can use the Gurobi 12.0 Solver to solve a much **wider range** of nonlinear models, much **more quickly and easily**, than you could any other way.

For "eligible" nonlinear optimization models, the Gurobi 12.0 Solver is often **faster** than other nonlinear Solvers, especially when **integer variables** are also used in the model, and when you require a **globally optimal** solution. This isn't always true however -- occasionally Gurobi is (much) slower -- so it's good that you have alternative nonlinear Solvers available to you, even in the basic Analytic Solver Optimization product. We do hope and expect to see even better performance in future versions of the Gurobi Solver.

Analytic Solver V2025 Q1 is a **free upgrade** for anyone with a paid Analytic Solver Optimization (or higher) license. If you've been a customer for years, you're probably accustomed to such dramatic enhancements -- but we're always working hard to bring you **more value, sooner** than anyone else in our "world" of advanced analytics.



What's New in Analytic Solver V2025 Q2

We've just released **Analytic Solver V2025 Q2**, with new versions of Analytic Solver Desktop and Cloud (for Excel users), Solver SDK (for developers), and our cloud platform RASON. This release features both **speed** and **usability** improvements, a **smarter** AI Agent to assist with model building, and support for the new/improved **Metalog 2** family of probability distributions.

Faster When Using, Not Just Solving

Customers (and prospective customers) regularly bring us large models that **stretch limits** in different ways. We've long supported models with millions of decision variables and constraints. But for example, customers have brought us models where one constraint uses a formula that depends on several other cells -- where each of these depends on several other cells -- to **10,000 or more** levels -- and we've found ways to make our analysis of those "deep formula chains" faster. When we find a "math + computer science" solution for these "stretch" cases, all customers end up benefiting.

More recently, customers have brought us models with so many constraints, formulas or Excel defined names that it took a long time to display or update the **Task Pane**. Other customers have brought us models with large numbers of **quadratic constraints** (each requiring a "Hessian evaluation"), or with constraint **right hand sides** deeply dependent on decision variables. In our V2025 Q2 release, Analytic Solver is **much faster** in all these cases.

And for those solving with the **Gurobi** Solver Engine, our V2025 Q2 release includes the latest 12.0.2 release from our friends at Gurobi Optimization, with their latest fixes and speed improvements.

Enhanced Excel Support, Smarter Human Support

Analytic Solver V2025 Q2 includes improved support for Excel models using the new **LET function** (the LAMBDA function was already supported). We've also improved support for recalculation of optimization and simulation models that use certain advanced Excel functions, notably our Psi functions and our support for DMN-style **decision tables** and "**box functions**".

And our **AI Agent** in V2025 Q2 is even better at **helping you** enhance and even find and fix problems in your model: It now takes advantage of both the latest ChatGPT large language model with our RAG support, and "**model diagnostics**" from our PSI Interpreter to better understand the structure of your model, and how it might be fixed or improved.

New Support for Metalog2 Distributions

Way back in 2017, Frontline Systems was among the very first software vendors to implement support for the new (introduced in 2016) **Metalog** family of probability distributions -- see also the [Metalog YouTube Channel](#). This family of distributions has some **remarkable properties**: virtually unlimited shape flexibility, flexible bounds, simple, closed-form quantile function (inverse CDF) equations, a simple, closed-form PDF (probability density function), and **Bayesian updating in closed form** in light of new data(!)

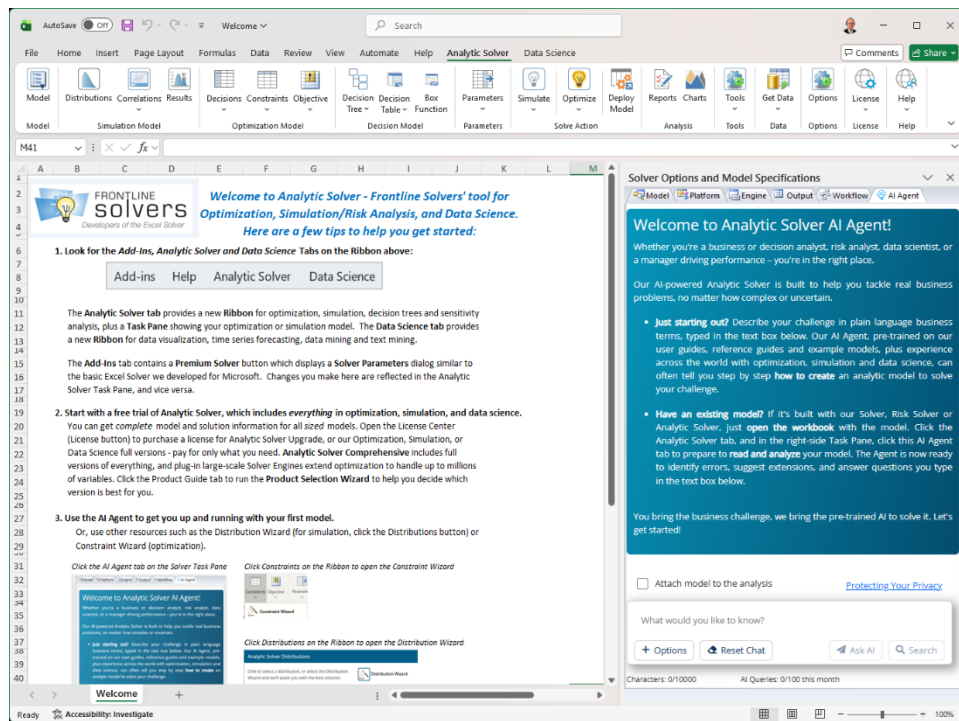
Over the last 8 years, the Metalog distributions have been **widely adopted**, in applications ranging from astrophysics and cosmology, crystallography, integrated circuit design and nanomaterials, to volcanic discharges in earth science, toxin levels in drinking water, oil-field production forecasting, electric vehicles and hydrogen production in renewable energy, cybersecurity risk mitigation, and conditional value at risk in quantitative finance.

Now in 2025, **Dr. Tom Keelin**, inventor of Metalog distributions, and two other authors are publishing a new paper *On the Properties of the Metalog Distributions* which refines and, in special cases, *redefines* the Metalog family of distributions (improving cases where the fitting process could yield infeasible results), earning the title "**Metalog 2**". And in Analytic Solver V2025 Q2, the refinements in this paper are **fully implemented** via our PsiMetalog2() function and our built-in automatic

distribution fitting. As in so many cases in the past, Analytic Solver users are **first to gain access** to this new/improved capability -- at no extra cost for current licensees!

What's New in Analytic Solver V2025 Q4

We've just released **Analytic Solver V2025 Q4**, with new versions of Analytic Solver Desktop and Cloud (for Excel users), Solver SDK (for developers), and our cloud platform RASON. In this release, we've focused on **ease of use** and better support for **model creation**. Most prominent is the new **AI Agent tab** in the Task Pane, which appears (in lieu of the Model tab) when you visit Analytic Solver on the Ribbon.



The **AI Agent's** ability to help you **build a model** in the first place is greatly improved in this release. You can *type in plain language* what you want to accomplish: For example “My company produces material in large sheets, that we need to cut into smaller sizes that our customers need. We’d like to do this while minimizing waste.” The AI Agent can recognize this as a “cutting stock optimization problem”, tell you how to formulate a model with the right variables and constraints, advise you what data you’ll need, and give you link(s) to example model(s) relevant to what you’re trying to accomplish.

You can also use the **AI Agent** to **diagnose a model** that you’ve previously created – looking for missing elements, possible errors, and suggested enhancements. Just open your model workbook, then go to the AI Agent tab in the Task Pane.

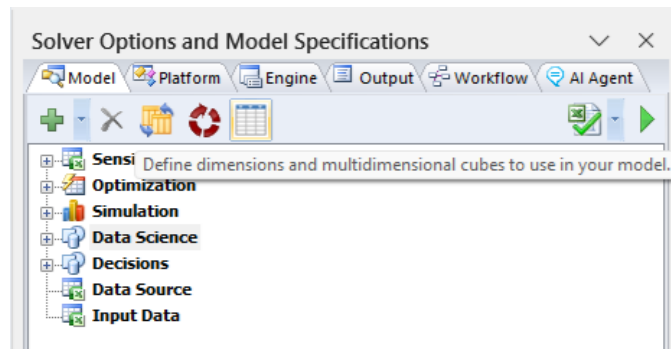
Semantic Search Without Using AI

Some customers are uncomfortable with the idea that an external agent like ChatGPT-5 might “read and understand” their model. If that’s your situation, you can still use the **Search** button (instead of the **Ask AI** button) at the bottom of the **AI Agent** tab. This performs a semantic search (*much better* than just searching for keyword matches) across all 3,500 pages

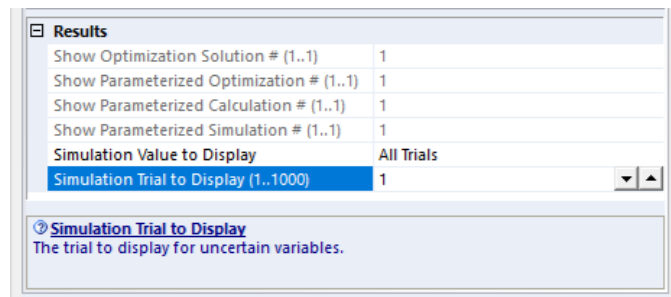
of Frontline’s User and Reference Guides, and our hundreds of example models. It’s also much faster than “Ask AI”.

A Shorter Ribbon, and Other Simplifications

Note that, with the AI Agent “up front” in the Task Pane, we no longer need the **AI Assist** and **Ask Question** icons on the Ribbon. We’ve also moved the **Dimensional Modeling** icon from the Ribbon to the Task Pane Model tab, where we’ve also placed the **Reset All** icon (in red):



And we’ve removed the **Tools** tab from the Task Pane: The controls allowing you to select (and display on the worksheet) specific Simulation Trials, or specific runs from multiple parameterized Simulations or Optimizations, are now – more conveniently – available at the bottom of the **Output** tab:



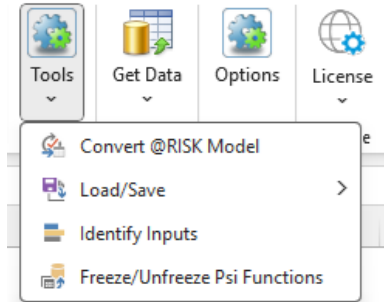
Easier to Find Distribution Fitting

In this release, we’ve *swapped the positions* of two popular features, based on customer feedback about where they *expected* to find them:

If you’re **defining probability distributions**, and you have sample data that you’d like to fit to a well-known type of distribution, you’ll find a **Fit Data** icon at the top of the main Distributions dialog, displayed when you click the **Distributions** icon on the Ribbon:



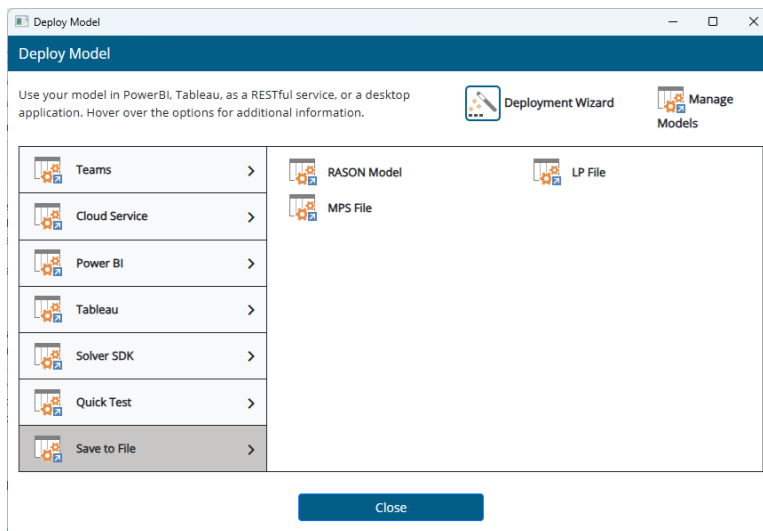
Fit Data (formerly available via the Tools icon, now here) replaces another popular feature, **Convert @RISK Model**. This option (which you’ll need only once, if at all) is now available via the **Tools** icon.



Easier to Find “LP or MPS Model Format”

Another popular feature formerly available under the **Tools** icon, often used to help “debug” linear or linear mixed-integer programming models, allow you to *write out* your model as an “**LP format file**”. This is a text file where every constraint is written in “algebraic linear coefficient form”. An older and even more basic file format, called a “MPS format file”, can also be written out.

From customer feedback, we concluded that this feature belonged with our *full array* of *model deployment* features, accessible in the dialog that appears for the **Deploy Model** icon:



If you haven’t previously spent time with the **Deploy Model** dialog, we *encourage* you to spend some time here: It’s the **easy way to share** your model results, or even your full solvable model, with **others in your company**, using Microsoft Teams, Power BI or Tableau, or our RASON Cloud Service.

With a *few mouse clicks*, you can turn your Excel model into a visual working model on your Power BI or Tableau dashboard – or *even* turn it into a “cloud service” readily *usable* from other *web* or *mobile* apps. If you’re not sure, just click the **Deployment Wizard** icon at the top of the dialog!

Installation

What you need

In order to install the Solver SDK software, which includes both the Solver SDK and SDK, you must be running Windows 10, Windows 7, Windows Server 2019, Windows Server 2016 or Windows Server 2012.

Getting Help

Frontline Systems technical support is ready and willing to offer assistance with installation of the Solver SDK. You may contact our Support department by phone (888-831-0333) or email (support@solver.com).

Installing the Software

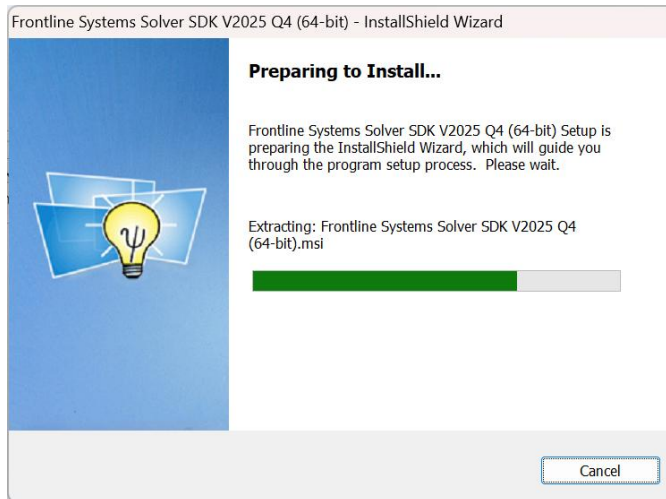
SDKSetup.exe installs both Solver SDK and XLMiner SDK. Your license determines your product selection. Your license may grant access to either Solver SDK or XLMiner SDK, or both. For more information on licensing, see *Logging in the First Time* below.

Note: Licenses for both Solver SDK Platform and Solver SDK Pro allow the use of XLMiner SDK at "basic" limits. See <https://www.solver.com/xlminer-sdk-platform> for more information.

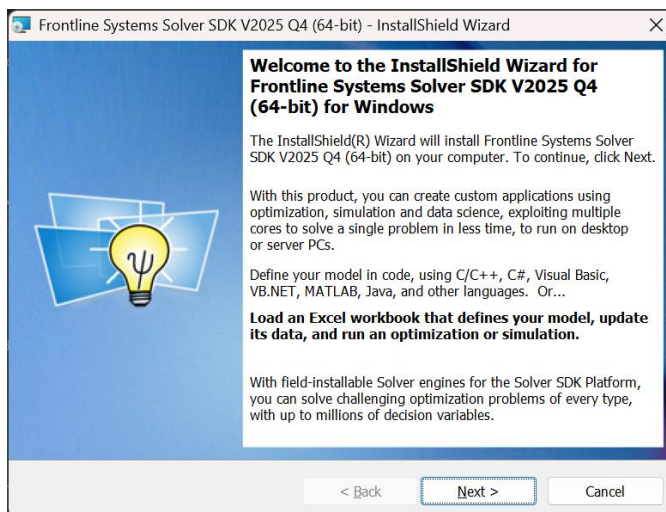
To install Solver SDK and XLMiner SDK to work with **64-bit** version of Windows, run **SDKSetup64.exe**.

Depending on your Windows security settings, you might be prompted with a message **Do you want to run this software?**, showing Frontline Systems, Inc. as the publisher. You may safely click **Run** in response to this message.

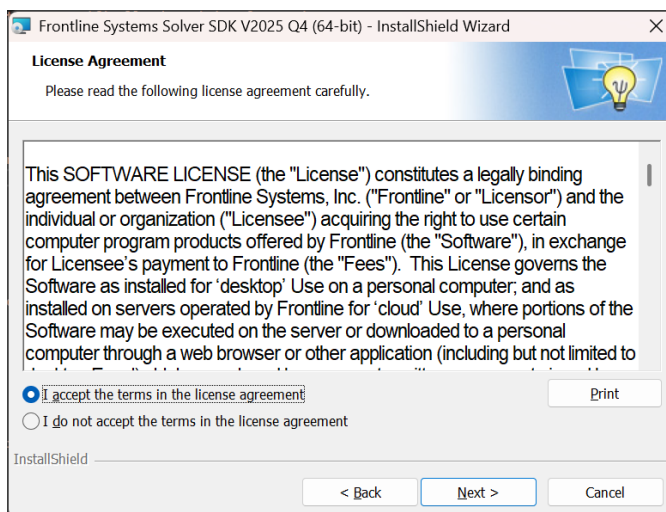
Next, you'll briefly see the standard Windows Installer dialog. Then a dialog box like the one shown below should appear:



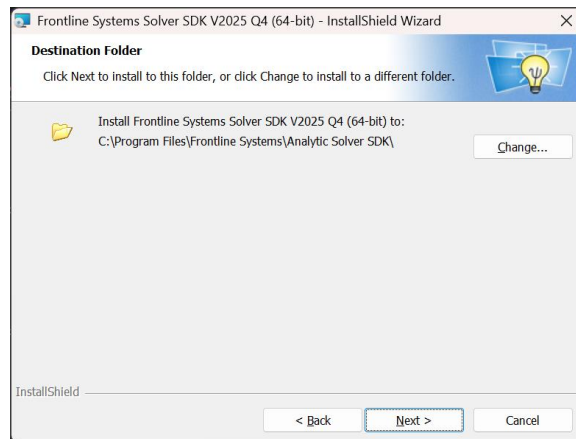
After the program initializes, the first dialog will clear and the second will appear. Click Next to proceed with the installation.



The Setup program will then ask if you accept Frontline's software license agreement. You must select "I accept" and click **Next** in order to proceed.

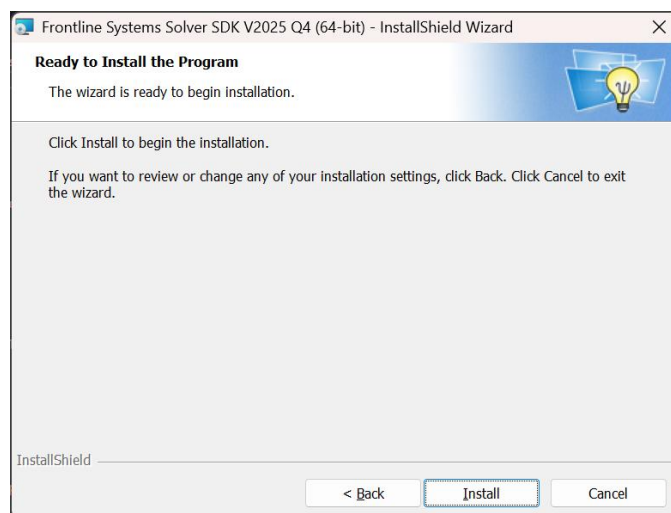


The Setup program then displays a dialog box like the one shown below, where you can select or confirm the folder to which files will be copied normally C:\Program Files\Frontline Systems\Analytic Solver SDK. Click **Next** to proceed.

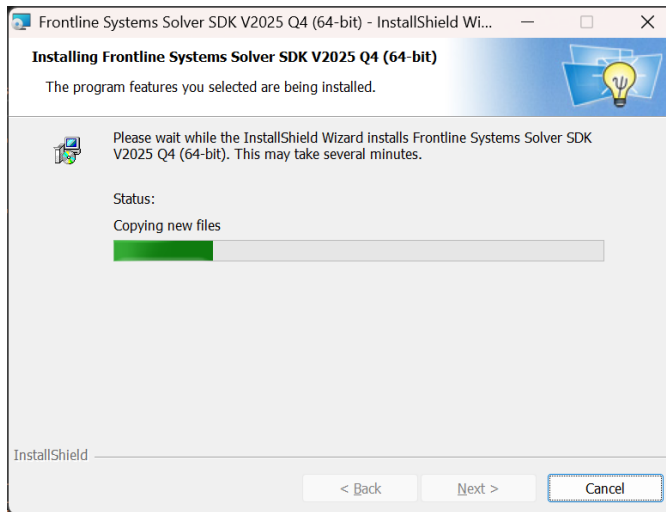


Click **Next** to proceed.

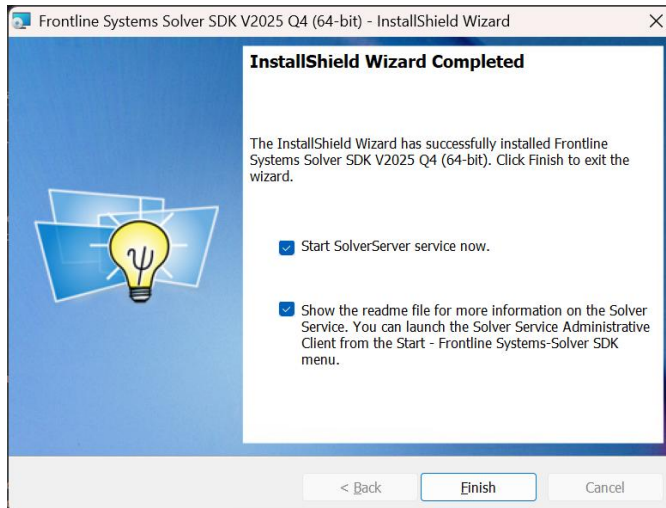
You will see a dialog confirming that preliminary steps are complete, and the installation is ready to begin. Click **Install** to proceed.



After you click **Install**, the Solver SDK files and examples will be installed. A progress dialog appears, as shown below. This may take several minutes to complete.



When the installation is complete, you'll see a dialog like the one below. Click **Finish** to exit the installation wizard.



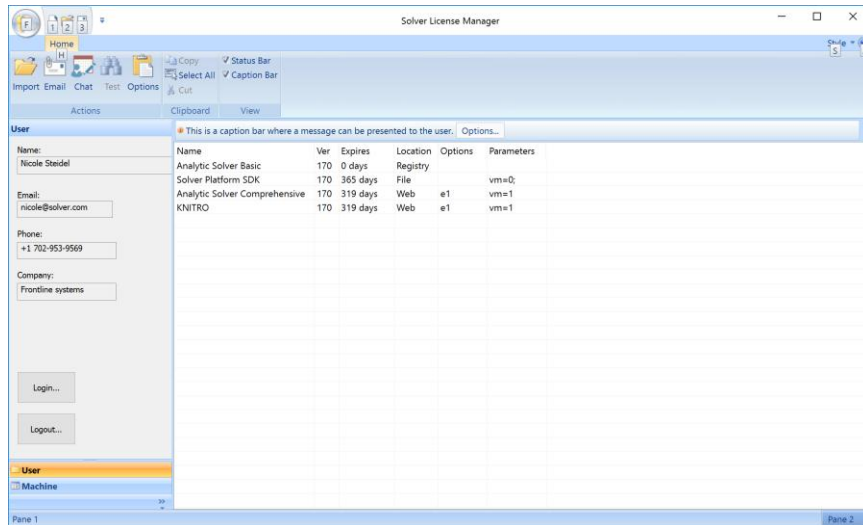
If you would like to start the Solver SDK Dedicated Server service, which allows a Client machine (PC, phone, tablet, etc.) to solve a model remotely with the Solver SDK Platform, then please check “Start the SolverServer Service now” checkbox. If you choose not to start the Solver SDK Dedicated Server service at this time, it can be started manually at a later time. For more information on our Solver SDK Dedicated Server service, please see the chapter, **Using Solver SDK Dedicated Server**.

Solver SDK and XLMiner SDK are now installed. Your license will determine what products are enabled. SDK example code, located at C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples, can now be run.

Logging in the First Time

In SDK V2025 Q4, your license is associated with **you**, and may be used on **more than one PC**. For example, you can run SDKSetup to install the software on your office PC, your company laptop, and your PC at home. But only **you** can use SDK, and only on **one** of these computers **at a time**.

After installation, you will first need to login to Frontline's License Manager before running SDK V2025 Q4 in order to obtain a license. Click **Start -- Frontline Systems -- License Manager** to open Frontline System's License Manager.



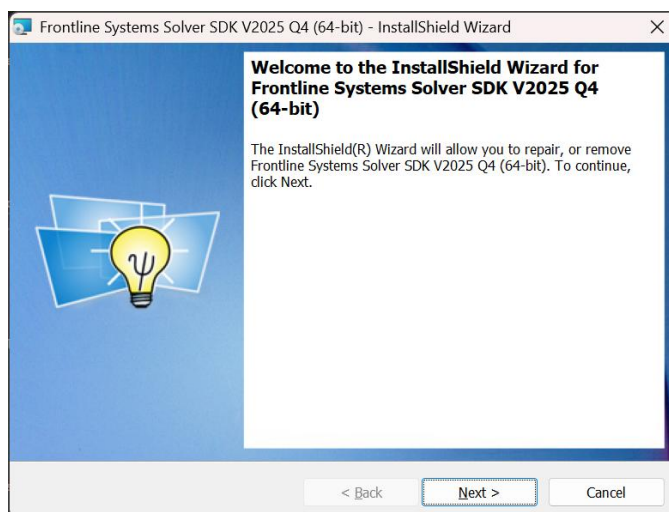
Click **Login..** then enter the **email address** and **password** that you used to register on Solver.com or AnalyticSolver.com. Once you've done this, your identity will be "remembered," so you won't have to login every time you use Solver SDK.

You can login and logout at any time, by clicking Logout on the License Manager dialog. If you share use of a single physical computer with other Solver SDK users, be careful to **login** with your own email and password, and **log out when you're done** – if you don't, other users could use up your allotted CPU time or storage.

Uninstalling the Software

To uninstall Solver SDK, just run the **SDKSetup** or **SDKSetup64** program as outlined above. You'll be asked to confirm that you want to remove the software.

You can also uninstall by choosing **Control Panel** from the **Start** menu, and double-clicking the **Programs and Features** or **Add/Remove Programs** applet. In the list box below "Currently installed programs," scroll down if necessary until you reach lines beginning with "Frontline," select Solver SDK Platform V2025 Q4 and click the Uninstall/Change or Add/Remove... button. Click OK in the confirming dialog box to uninstall the software.



Licensing the Solver SDK

Important Considerations

Please bear in mind that in order to lawfully distribute copies of the Solver SDK within your organization or to external customers, or to lawfully use the Solver SDK in a server-based application that serves multiple users, you will need an appropriate license from Frontline Systems. Licenses to use and/or distribute the Solver SDK in conjunction with your application are available at substantial discounts that increase with volume. Please contact Frontline Systems for licensing and pricing information.

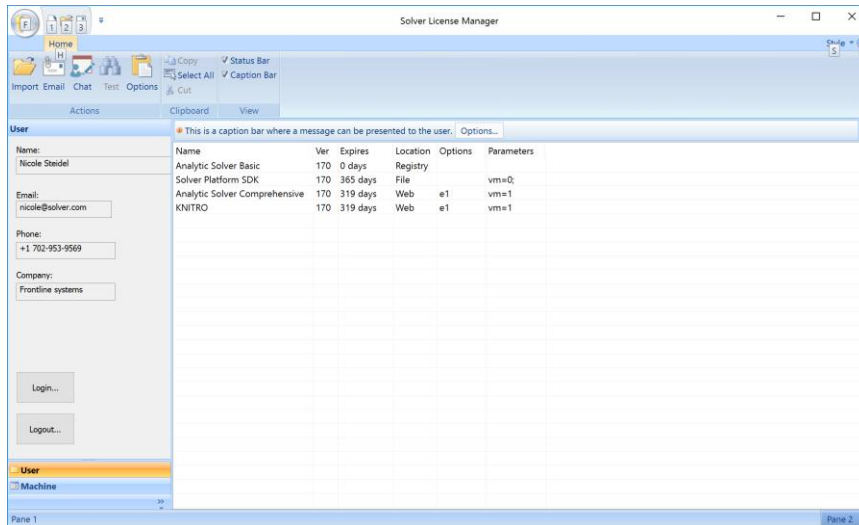
Frontline Systems offers three types of licenses for Solver SDK: Single-User, Multi-User and Runtime.

- A Single-User License is our standard "paid" license granted to a single named individual. This license allows you to install and use the software on more than one computer -- such as a desktop, laptop, home computer, or virtual computer -- but you may use the license only on one computer at a time.
- A Multiple-User License is our standard "paid" multi-user or "**concurrent use**" license, normally purchased by an organization. With this license, several different individuals may use the software, but only one of them at any given time.
- Runtime licenses give you the right to embed Solver SDK, and any desired plug-in engines, and duplicate or distribute copies, or make the application accessible to users on an Intranet or Web Server, or via a Web service.

Single User Licenses

A Single-User License is our standard "paid" license granted to a single named individual. As noted above, this license allows you to install and use the software on more than one computer -- such as a desktop, laptop, home computer, or virtual computer -- but you may use the license only on one computer at a time.

To obtain a permanent license for a Single-User license, go to **Start | Frontline Systems | License Manager**, click **Login** and then enter your Solver credentials.



For further flexibility, a license tied to the hardware of your machine may be created. For more information on this option, please contact Sales@Solver.com.

Multiple User Licenses

A Multiple-User license allows multiple users, working at multiple computers, to use a Frontline Systems software product, so long as the number of instances of the software running at any given time does not exceed the number of Multi-User licenses owned. Frontline must receive a complete list of the email addresses of the individuals who are entitled to share use of the license. Each of the users may install the software on more than one computer, just as for a Single-User License. Any of the named users can temporarily "acquire" the license and use the software, but only one at a time, per Multiple-User License. If another one of the named users seeks to acquire the license, he/she will receive a message indicating that the license is in use -- this will normally include the email address of the user who is "holding" the license. The second user must wait until the license is released and then try again. A license is allocated when Problem.Optimize or Problem.Simulate is called. A license is released when the Problem object (associated with the call to Optimize or Simulate) is destroyed. A user whose email address is listed on several Multiple-User Licenses can "acquire" a license as soon as one of them becomes free.

The users and computers may be **located anywhere**, and they may each have their own local copies of Frontline Systems software files, but the computers must be able to communicate with Frontline Solvers License Manager over a TCP/IP or IPX/SPX network.

For further flexibility, in most cases a Multiple-User License may be "checked out" from the license server and used for a period of time on a laptop or notebook computer that is disconnected from the network. (During this time, the license is not available for use on the network.) The license may be "checked in" to the license server at any time, making it again available for shared use. If you expect to use this option, please contact Frontline Systems Support at Support@solver.com.

As stated above for a Single User License, for further flexibility, a license tied to the hardware of your machine may be created. For more information on this option, please contact Sales@Solver.com.

Runtime Licensing Programs

Frontline's runtime licenses give you the right to embed the Solver SDK and— if required — one or more of our large-scale Solver Engines into your application, and duplicate or distribute copies, or make the application accessible to users on an Intranet or Web server, or via a Web service. Our

runtime License management options make it easy to monitor and control use of the software, using either Frontline-provided tools or your own tools.

A **development license** enables one or more application developers, to write, test, debug, and modify an application, using all of the facilities of the Solver SDK. In contrast, a **runtime license** enables one or more end users to run the finished application that uses the Solver to find solutions to optimization or simulation problems; it cannot be used to modify, recompile or debug the application itself. A runtime license may be temporarily used (and re-used) for client evaluations and sales demos, or it may be permanently deployed at client or customer sites.

A runtime license **costs less and "does less"** than a development license. Where a Solver SDK development license allows you to use any of five bundled Solver Engines for optimization and/or simulation, an SDK runtime license allows you to use one specific Solver Engine -- such as the LP/Quadratic Solver or the LSGRG Nonlinear Solver -- that is required to solve the specific problem defined by your finished application. Runtime license prices generally start at 50% of a development license, and are further discounted for quantity.

Runtime licenses may be purchased only after an appropriate **license agreement is signed** with Frontline Systems. We offer a standard form of license agreement that is sufficient for many common situations, and does not require a license origination fee. If our standard form of agreement is not appropriate, we'll be happy to negotiate a custom agreement with your organization that includes a license origination fee to cover the extra costs of negotiation.

The copy of the Solver SDK that you license for distribution, is customized for use by your application program(s). It recognizes a specific character license key string which you provide as a property of an object in your code. Without this license key string, the Solver SDK will not function -- all Solver SDK routines will return without doing anything. It is important that you keep your license key string confidential, and use it only in your application programs. Frontline Systems will treat you as responsible for the use of any copies of the Solver SDK that recognize your specific license key string.

See example code below for creating a new environment object and passing the license string. Example code is provided for COM, Visual Basic, VB.NET, C/C++/C#, Java, and Matlab.

COM, Visual Basic, VBA and VB.NET Example Code

```
Dim env as New Environment          'creates new environment
env.Create "r12345"                  'sets the runtime ID: r12345
.....
env = nothing                        'destroys environment
```

Procedural C Example Code

```
HEnv* env;
SolverEnvInit(& env, L"");           //creates new environment
SolverEnvRuntimeIDSet (env, L"r12345"); //sets the runtime ID: r12345
```

C++ Example Code

```
SolverPlatform.Environment env;      //creates new environment
env.Create(L"r12345");                //sets the runtime ID: r12345
```

C# Example Code

```
//creates new environment
SolverPlatform.Environment env = new SolverPlatform.environment();
env.create("r12345");               //sets the runtime ID: r12345
```

Java Example Code

```
//creates new Environment
SolverPlatform.Environment env = new SolverPlatform.Environment();
env.Create ("r12345"); //sets the runtime ID: r12345
```

Matlab Object Oriented Example Code

```
env = Environment(); //creates new environment
Create(env, "r12345"); //sets the runtime ID: r12345
.....
Destroy(env); //destroys environment
```

Matlab Procedural Example Code

```
env = SolverSDK("EnvInit"); //creates new Environment
SolverSDK("EnvRuntimeIDSet","r12345"); //sets runtime id: r12345
.....
env = SolverSDK('EnvFree'); //destroys Environment
```

Python Example Code

```
env = Environment(); //creates new environment
Create(env, "r12345"); //sets the runtime ID: r12345
.....
Destroy(env); //destroys environment
```

R Example Code

```
env = Environment(); //creates new environment
Create(env, "r12345"); //sets the runtime ID: r12345
.....
Destroy(env); //destroys environment
```

Frequently Asked Questions

Why do I keep getting the error, "The specified module could not be found"?

This error occurs when the application is unable to find the SolverSDK.dll. If you are using .NET to call the Solver SDK, then you simply need to save the SolverSDK.dll into the Bin directory of the .NET Project. If you are using Java to call the Solver SDK, then you must set the LD_LIBRARY_PATH. If you are using Matlab, then you must copy the SolverSDK.dll into the folder where your Matlab script is located.

Why can't the Solver SDK find my license file?

One of the most common problems that users encounter with the Solver SDK is one where a valid license for the SDK exists, but the SolverSDK.dll is unable to find the file and thus stops with a "No License Found" exception. The SolverSDK.dll searches for its license file in three locations. Saving your solver.lic license file into one of these three locations will prevent this type of error from occurring.

1. If the user is logged into Frontline's License Server, using SolverLicMan.exe, then Solver SDK will first look for the license on the server. SolverLicMan.exe is installed into the Solver SDK directory at C:\Program Files\Frontline Systems\Analytic Solver SDK\Bin.

2. The DLL first checks if the User is using the Environment Object in the code to specify the location of the license file.
3. If the DLL does not find the location of the license file in the Environment Object, then it will check the FRONTLIC environment variable.
4. If the license file is not found in Steps 1 or 2 above, then the DLL will look in the directory where the SolverSDK.dll is located.

If the license file is not found in any of the three locations above, the Solver SDK will stop with the exception, SolverSDK: No License Found.

Designing Your Application

Calling the Solver SDK

This chapter provides an overview of the way your application program should call the Solver SDK routines to define and solve an optimization or simulation problem. We'll use the syntax of the C++ programming language, but the order of calls, the names of routines, the arguments, and most other programming considerations are the same in other languages.

At run time, your application program uses dynamic linking to load the Solver SDK and call various routines within it. Details of the process of compiling, linking and running your application with the Solver SDK are provided in the chapters on the various languages.

Solver SDK includes the ability to solve models written in RASON, Frontline's modeling language embedded in JSON. For more information on RASON, see the website www.RASON.com. Browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\RASON to find several example models written using the RASON syntax. See the RASON User Guide, downloadable from WWW.RASON.COM under the HELP tab, to learn more about this powerful modeling language.

Loading and Solving RASON and Excel Models

The quickest and easiest way to use the Solver SDK is to load a model built with the Excel Solver or written in RASON. (*Load Model functionality supported only in the Solver SDK Platform*). This only requires three lines of code. The code below loads the active spreadsheet inside the Product Mix.xlsx Excel model. Note: Excel workbooks containing the Solver models must have extensions of .xlsx or .xlsm (saved with Microsoft Excel 2007 or later).

```
CProblem prob;  
prob.Load(L"Product Mix.xlsx", SolverPlatform_File_FormatXLStd);  
prob.Solver.Optimize();
```

Note: The File_Format argument is not required. Solver SDK can determine the file type automatically. For example, to load the productmix.json RASON model, installed into C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\RASON\Linear, simply use:

```
prob.Load(L"C:\\Program Files\\Program Files\\Analytic Solver  
SDK\\SolverSDK\\Examples\\RASON\\Linear\\ProductMix.json");
```

Three additional lines allow you to obtain the optimization status, final variable values, and final objective value.

```
Optimize_Status status = prob.Solver.OptimizeStatus;  
double[] varvalue = prob.VarDecision.FinalValue.Array;  
double[] objValue = prob.FcnObjective.FinalValue.Array;
```

It is also possible to solve a model on a spreadsheet that is not the active spreadsheet. The line of code below will load a model from Sheet1 within the Product Mix.xlsx workbook.

```
prob.Load("Product Mix.xlsx!Sheet1",
SolvePlatform_File_Format.XLStd);
```

The Platform Excel models may contain most Excel formulas or functions (even new Excel 2010 functions). However, user-defined VBA functions written in VBA are not supported. (Please contact our Technical Support (support@solver.com) specialists for help with user-defined functions.) Typically, if your model runs with the PSI Interpreter in Analytic Solver for Excel, then the model will also run in the Solver SDK. When a model is run in the SDK, problem setup is usually faster, and solution times are either the same speed or faster than in Excel.

Analyzing and Solving Excel Models in the SDK Platform

It is possible to diagnose a Solver Excel model that has been loaded into the Solver SDK Platform. To do so, simply load the model and then call *prob.Solver.Optimize(Solve_Type_Analyse)*. To get the diagnosis information, call *prob.ProblemType* as shown below.

```
int ret;
CProblem prob;
prob.Load("Product Mix.xlsx", SolverPlatform_File_FormatXLStd);
prob.Solver.Optimize(Solve_Type_Analyse);
ret = prob.ProblemType;
```

There are six different problem types that can be returned from *prob.ProblemType*.

Return Value	Problem Type	Description
1	Problem_Type_OptLP	Linear Problem
2	Problem_Type_OptQP	Quadratic Problem
4	Problem_Type_OptQCP	Quadratically Constrained Problem
8	Problem_Type_SOCP	Second Order Cone Programming
16	Problem_Type_OptNLP	Nonlinear Problem
32	Problem_Type_OptNSP	Nonsmooth Problem

Once you know the problem type, you can select the best engine for solving. Please see the table below for the engines best suited for each problem type in the Solver SDK Platform.

Solver SDK Platform Engine	Problem Type
Large Scale GRG Engine	OptNLP
Standard LPQP Engine	OptLP, OptQP
Standard Evolutionary Engine	OptNSP
Standard SOCP Engine	OptQCP

For example, if *prob.ProblemType* returns a 16 (Problem_Type_NLP), then the Large Scale GRG Engine should be selected for solving. Please see the section below titled, **Selecting an Engine in the SDK**, for an example on how to select an engine for solving.

There are four additional *Solve_Type*s for use with *prob.Solver.Optimize*: *Solve_Type_NoIntegers*, *Solve_Type_SimOpt*, *Solve_Type_Solve*, and *Solve_Type_SolveForRecourse*. *Solve_Type_NoIntegers* instructs the SDK to solve the "relaxation" of the problem. This means that the Solver SDK will ignore integer constraints (including alldifferent constraints) during the solving process. *Solve_Type_SimOpt* will instruct the Solver SDK Platform to solve a simulation optimization problem while *Solve_Type_Solve* tells the Solver SDK Platform to solve a "normal" optimization problem. (If *Solve_Type* is omitted, *Solve_Type_Solve* is used by default.) *Solve_Type_SolveForRecourse* instructs the SDK Platform to solve only for recourse variables. (When using this *Solve_Type*, the model loaded must include Recourse Variables.)

Input Objects

The PsiInput function in Excel provides a way to mark certain cells as containing (changeable) input data for an optimization or simulation model. 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 more information on the InputData object, please see the Solver SDK Platform Reference Guide.)

Parameter Objects

Analytic Solver in Excel includes three problem parameters, PsiOptParam, PsiSimParam and PsiSenParam. The values of PsiOptParam() are automatically varied when you perform multiple optimizations. PsiSimParam() functions are automatically varied when you perform multiple simulations. The values of PsiSenParam() function calls are automatically varied when you create a Sensitivity Analysis Report or Chart in Excel. If loading an Excel workbook, these parameters can all be accessed via the CParameter Class or, if using the Procedural API, through various Solver functions. For more information on this class, please see the Solver SDK Platform Reference Guide.

Solving Linear and Quadratic Optimization Problems

The overall structure of an application program calling the Solver SDK Pro to solve a linear programming problem or the Solver SDK Platform to solve a linear or quadratic programming problem is as follows:

```
/* Get data and set up an LP problem */
.....
//create an instance of a problem
CProblem prob;
//pass the number of variables in the model
prob.VarDecision.Size = nvars;
//pass the number of constraints in the model
prob.FcnConstraint.Size = ncons;
//pass the constraint right hand sides
prob.FcnConstraint.UpperBound.Array = rhsvalue;
//set lower bounds of all unconstrained variables to 0
prob.VarDecision.NonNegative();
//pass the DoubleMatrix that holds the objective coefficients
prob.Model.AllLinear[Function_Type_Objective] = fcnObj;
```



```

//pass the DoubleMatrix that holds the constraint coefficients
prob.Model.AllLinear[Function_Type_Constraint] = fcnConstr;
//set the objective to Maximize
prob.Solver.SolverType = Solver_Type_Maximize;
//solve the model
prob.Solver.Optimize();
//get the status, var values, and objective value
Optimize_Status status = prob.Solver.OptimizeStatus;
Double[] varvalue = prob.VarDecision.FinalValue.Array;
Double[] objValue = prob.FcnObjective.FinalValue.Array;
...

```

Your program should first create a new instance of the class `CProblem`. The class `CProblem()` represents a function that creates a new problem. It also initializes the type of Solver engine to be used and specifies the number of variables and functions. The class properties describe the type of analysis to be performed. For example, *prob.VarDecision.Size* sets the number of variables in the model. To set the model type, you call *prob.Solver.SolverType = Solver_Type_Maximize* for a maximization and *prob.Solver.SolverType = Solver_Type_Minimize* for a minimization. Optionally, you may call *prob.Model.FcnQuadratic* to specify a quadratic objective, and/or *prob.Model.IntegerType* to specify integer variables. To solve the problem, call *prob.Solver.Optimize*. To retrieve the solution, call *prob.Solver.OptimizeStatus*. This cycle may be repeated to solve a series of optimization problems.

Solving Nonlinear Optimization Problems

In a linear or quadratic problem, you can describe the objective and constraints with an array or matrix of constant coefficients; the Solver SDK can determine values for the objective and constraints by computing the sums of products of the variables with the supplied coefficients. In a nonlinear problem, however, the problem functions cannot be described this way; the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change.

As a result, you must write a "callback" function (called *NLPEvaluator_Evaluate()* below) that computes values for the problem functions (objective and constraints) for any given values of the variables. The Solver SDK will call this function repeatedly during the solution process. The overall structure of a program calling the Solver SDK to solve a nonlinear programming problem is:

```

Engine_Action NLPEvaluator_Evaluate(...)
{
/* Receive values for the variables, compute values */
/* for the constraints and the objective function */
}

int main()
{
/* Get data and set up an NLP problem */
...
CProblem prob
//pass the number of variables in the model
prob.VarDecision.Size = nvars;

```

```

//pass the number of constraints in the model
prob.FcnConstraint.Size = ncons;
//pass the constraint right hand sides
prob.FcnConstraint.UpperBound.Array = rhsvalue;
//pass the variable upper bounds
prob.VarDecision.UpperBound.Array = ub;
//pass the variable lower bounds
prob.VarDecision.LowerBound.Array = lb;
//call the evaluator to compute the obj and constraints
prob.Evaluators[Eval_Type_Function].OnEvaluate +=
new EvaluateEventHandler[NLPEvaluator_Evaluate];
//set the problem type to Minimize
prob.Solver.SolverType = Solver_Type_Minimize;
//solve the model
prob.Solver.optimize();
//get the status, var values, and objective value
Optimize_Status status = prob.Solver.OptimizeStatus;
Double[] varvalue = prob.VarDecision.FinalValue.Array;
Double[] objValue = prob.FcnObjective.FinalValue.Array;
...
}

```

Your program should first create a problem, *Problem prob = new Problem* then call additional routines in the *problem* class. Optionally, you may call *prob.Model.AllGradDepend()* to give the Solver more information about linear and nonlinear functions in your problem, *prob.Model.DependTest()* to have the Solver determine the *prob.Model.AllGradDepend()* information through a numerical test, or *prob.Model.IntegerType()* to specify integer variables – defining an integer nonlinear problem. Then call *prob.Solver.optimize()* to solve the problem. To retrieve the optimization status, the final decision variable values and the final objective function value, call *prob.Solver.OptimizeStatus()*, *prob.VarDecision.FinalValue*, and *prob.FcnObjective.FinalValue*, respectively. As with solving a linear optimization model, this cycle may be repeated to solve a series of Solver problems.

For more information on evaluators and solving nonlinear problems, see the next chapter *Solver SDK Platform Evaluators*.

Solving Nonsmooth Optimization Problems

You define and solve a problem with nonsmooth or discontinuous functions in much the same way as you would for a problem defined by smooth nonlinear functions. In a nonsmooth problem, however, the first partial derivatives of the objective and constraints may not be continuous functions of the variables, and they may even be undefined for some values of the variables. Because of this property, the gradient-based methods used by the GRG Solver are not appropriate, and instead the Evolutionary Solver "engine" must be used to solve the problem.

As for smooth nonlinear problems, you write a "callback" function, *NSPEvaluator_Evaluate* in the example below, that computes values for the problem functions for any given values of the variables. But you cannot use the optional *Eval_Type_Gradient* callback function (described below) to speed up the solution of a nonsmooth problem.

```

Engine_Action NSPEvaluator_Evaluate(...)
{
/* Receive values for the variables, compute values */
/* for the constraints and the objective function */
}

int main()
{
...

/* Get data and set up an NSP problem */
//pass the number of variables in the model
prob.VarDecision.Size = nvars;
//pass the number of constraints in the model
prob.FcnConstraint.Size = ncons;
//pass the constraint right hand sides
prob.FcnConstraint.UpperBound.Array = rhsvalue;
//pass the variable upper bounds
prob.VarDecision.UpperBound.Array = ub;
//pass the variable lower bounds
prob.VarDecision.LowerBound.Array = lb;
//call the evaluator to compute the obj and constraints
prob.Evaluators[Eval_Type_Function].OnEvaluate +=
new EvaluateEventHandler(NSPEvaluator_Evaluate);
prob.Engine = prob.Engines[CEngine::EVOName];
//set the problem type to Minimize
prob.Solver.SolverType = Solver_Type_Minimize;
//solve the model
prob.Solver.Optimize();
//get the status, var values, and objective value
Optimize_Status status = prob.Solver.OptimizeStatus;
Double[] varvalue = prob.VarDecision.FinalValue.Array;
Double[] objValue = prob.FcnObjective.FinalValue.Array;
...
}

```

Your program should first create a problem, *Problem prob = new Problem*, and then call *prob.Engine = prob.Engines[Engine::EvoName]*; to call the Evolutionary Engine. You could also set the problem type, *prob.ProblemType = Problem_Type_OptNSP*, to nonsmooth which will automatically call the Evolutionary Engine.

You may optionally call `prob.VarDecision.Integertype()` to specify integer variables – defining an integer nonsmooth problem. Then call `prob.Solver.Optimize()` to solve the problem. To retrieve the solution status, the final values of the decision variables and the final objective function value, call `prob.Solver.OptimizeStatus()`, `prob.VarDecision.FinalValue`, and `prob.FcnObjective.FinalValue`, respectively. As with both linear and nonlinear optimization problems, this cycle may be repeated to solve a series of Solver problems.

For more information on evaluators and solving nonsmooth problems, see the next chapter *Solver SDK Platform Evaluators*.

Genetic and Evolutionary Algorithms

A non-smooth optimization problem generally cannot be solved to optimality, using any known general-purpose algorithm. But the Evolutionary Solver can often find a "good" solution to such a problem in a reasonable amount of time, using methods based on genetic or evolutionary algorithms. (In a "genetic algorithm," the problem is encoded in a series of bit strings that are manipulated by the algorithm; in an "evolutionary algorithm," the decision variables and problem functions are used directly. Most commercial Solver products are based on evolutionary algorithms.)

An evolutionary algorithm for optimization is different from "classical" optimization methods in several ways. First, it relies in part on random sampling. This makes it a *nondeterministic* method, which may yield different solutions on different runs.

Second, where most classical optimization methods maintain a single best solution found so far, an evolutionary algorithm maintains a *population* of candidate solutions. Only one (or a few, with equivalent objectives) of these is "best," but the other members of the population are "sample points" in other regions of the search space, where a better solution may later be found. The use of a population of solutions helps the evolutionary algorithm avoid becoming "trapped" at a local optimum, when an even better optimum may be found outside the vicinity of the current solution.

Third – inspired by the role of mutation of an organism's DNA in natural evolution – an evolutionary algorithm periodically makes random changes or *mutations* in one or more members of the current population, yielding a new candidate solution (which may be better or worse than existing population members). There are many possible ways to perform a "mutation," and the Evolutionary Solver actually employs three different mutation strategies. The result of a mutation may be an infeasible solution, and the Evolutionary Solver attempts to "repair" such a solution to make it feasible; this is sometimes, but not always, successful.

Fourth – inspired by the role of sexual reproduction in the evolution of living things – an evolutionary algorithm attempts to combine elements of existing solutions in order to create a new solution, with some of the features of each "parent." The elements (e.g. decision variable values) of existing solutions are combined in a *crossover* operation, inspired by the crossover of DNA strands that occurs in reproduction of biological organisms. As with mutation, there are many possible ways to perform a "crossover" operation – some much better than others – and the Evolutionary Solver uses multiple variations of two different crossover strategies.

Fifth – inspired by the role of natural selection in evolution – an evolutionary algorithm performs a *selection* process in which the "most fit" members of the population survive, and the "least fit" members are eliminated. In a constrained optimization problem, the notion of "fitness" depends partly on whether a solution is feasible (i.e. whether it satisfies all of the constraints), and partly on its objective function value. The selection process is the step that guides the evolutionary algorithm towards ever-better solutions.

A drawback of an evolutionary algorithm is that a solution is "better" only in comparison to other, presently known solutions; such an algorithm *actually has no concept of an "optimal solution,"* or any way to test whether a solution is optimal. (For this reason, evolutionary algorithms are best employed on problems where it is difficult or impossible to test for optimality.) This also means that an evolutionary algorithm has no definite rule for *when to stop*, aside from the length of time, or the number of iterations or candidate solutions that you wish to allow it to explore. Aside from such limits, the Evolutionary Solver uses two heuristics to determine whether it should stop – one based on

the "convergence" of solutions currently in the population, and the other based on the rate of progress recently made by the algorithm.

Optimization Problems in Algebraic Notation

As outlined above, problems for the Solver SDK are generally defined by a series of calls made by your application program. In particular, for linear and if using the Solver SDK Platform, for quadratic problems as well, you must be careful to supply the correct values for objective and constraint coefficients and constraint and variable bounds to define the problem you want to solve. Problems you define exist only in main memory for the duration of your calls to the Solver SDK – they do not "persist" across runs of your application.

To make it easier to work with linear and quadratic problems, the Solver SDK can read and write text files containing a problem description in a form very similar to standard algebraic notation. An example of such a text file for a simple linear integer problem is shown below:

```
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 + 11.0 x2 <= 50.0
Bounds
    0.0 <= x1 <= +infinity
    0.0 <= x2 <= +infinity
Integers
    x1
    x2
End
```

After you setup and define this problem, you can produce a text file with the contents shown above by calling *prob.Save*. (This provides a convenient way to verify that the problem you defined programmatically is the problem you intended, by examining the resulting text file.) If you have such a text file containing your problem, you can define your model by calling *prob.Load()*, without setting up all of the array arguments normally required.

You can use *prob.Save()* and *prob.Load()* to "persist" the definition of a problem on disk between runs of your application program, without having to write code to store this data in some file format of your own design.

Both *prob.Save()* and *prob.Load()* support the following files types: RASON, Excel, JSON_INPUT, JSON_OUTPUT, LP, LPFML, MPS, and OSIL.

Saving Models in RASON format

It's possible to load an Excel model using *prob.load* and save the model in RASON format using *prob.save*. If saving a model using the RASON syntax, use *prob.Save()* with File Format = JSON_Output.

```
problem.Save (L"prb", SolverPlatform::File_Format_JSON_Output);
```

Setting the Problem Type

The Solver SDK categorizes optimization problems using the following six types: *Problem_Type_LP* (linear problem), *Problem_Type_OptQP* (quadratic optimization problem), *Problem_Type_OptQCP* (quadratically constrained optimization problem), *Problem_Type_OptNLP* (nonlinear optimization problem), and *Problem_Type_OptNSP* (non-smooth optimization problem). If the problem type is set, the Solver SDK will choose the best engine available to solve that problem type. Please see below for example code in C++ on how to set the problem type or click here for more examples in the [Procedural API](#), [C#](#) or [COM](#).

```
prob.ProblemType = Problem_Type_OptNSP;
```

If you are calling `AllLinear`, the Solver SDK Platform will automatically select the best linear solver to solve the model. If no additional Solver engines are present, such as the Gurobi or Large Scale LP/QP engine, then the LP/Quadratic engine will be selected. Likewise, if you are solving a nonlinear problem or nonsmooth problem and have passed the problem type `OptNLP` or `OptNSP`, then the best nonlinear or nonsmooth solver will be selected. Within the Solver SDK Platform, the best nonlinear engine is the Large Scale GRG engine; the best engine for solving nonsmooth problems is the Evolutionary Engine. Please see the table below for the engines best suited for each problem type.

Engine	Problem Type
Large Scale GRG Engine	OptNLP
Standard LPQP Engine	OptLP, OptQP
Standard Evolutionary Engine	OptNSP
Standard SOCP Engine	OptQCP

If you are calling `AllLinear`, the Solver SDK Pro will automatically select the best linear solver to solve the model. Likewise, if you are solving a nonlinear problem or nonsmooth problem and have passed the problem type `OptNLP` or `OptNSP`, then the best nonlinear or nonsmooth solver will be selected. Within the Solver SDK Pro, the best nonlinear engine is the Standard GRG engine; the best engine for solving nonsmooth problems is the Evolutionary Engine. Please see the table below for the engines best suited for each problem type.

Engine	Problem Type
Standard GRG Engine	OptNLP, OptQP, OptQCP
Simplex LP Engine	OptLP
Standard Evolutionary Engine	OptNSP

Selecting an Optimization Engine

Selecting an Engine in the SDK

You can also set the engines manually. Please see below for an example in C++ on how to set any of the bundled optimization engines, Standard Simplex LP, Standard LP/Quadratic, Standard GRG or Large Scale GRG, Standard Evolutionary, and Standard SOCP Barrier, respectively. Please see below for example code in C++ or click the following links: [C#](#), [Procedural API](#), or [COM](#).

```

prob.Engine = prob.Engines[CEngine::LPName];
prob.Engine = prob.Engines[CEngine::LPQPName];
prob.Engine = prob.Engines[CEngine::GRGName];
prob.Engine = prob.Engines[CEngine::EVOName];
prob.Engine = prob.Engines[CEngine::SOCPName];

```

Selecting Large Scale Solver Engines

If the size of your model exceeds the problem limits of the four bundled optimization engines, then you will need to add a Large Scale Engine to the engine collection. (You can download these additional engines from our Website at www.solver.com under Downloads/SDK Products. A password and activation code will be emailed to the address provided during registration.) For example, say you have a linear model with more than 9,000 variables. Since the LP/Quadratic engine, bundled within the SDK, has a variable limit of 8,000 variables, you will need to add an additional engine that is designed to solve large linear problems. Let us assume that we will be using the Gurobi Solver Engine which can solve linear models of virtually unlimited size. Please see the examples below which illustrate how to add this engine to the engine collection and then select the engine to solve the problem.

The first line of code below creates the variable, *engGurobi* of type *CEngine*, and initializes the engine to the Gurobi engine located at the path, C:\Program Files\Frontline Systems\Engines\gurobieng.dll. The second line of code adds engGurobi to the engine collection and then selects the engine to solve the problem. For examples in additional languages, click the following links: [C#](#), [Procedural API](#), or [COM](#).

```

CEngine engGurobi(L"GUROBI",L"C:\\Program Files\\Frontline
Systems\\gurobieng.dll");

prob.Engine = prob.Engines [L "engGurobi"];

```

Please see the table below for example code on selecting any of the remaining Large Scale field installable engines.

KNITRO	<pre> CEngine engKnitro(L"KNITRO",L"C:\\Program Files\\Frontline Systems\\Engines\\knitroeng.dll"); prob.Engine = prob.Engines [L "engKnitro"]; </pre>
LSGRG	<pre> CEngine engLSGRG(L"LSGRG",L"C:\\Program Files\\Frontline Systems\\Engines\\lsrgeng.dll"); prob.Engine = prob.Engines [L "engLSGRG"]; </pre>
LSLP	<pre> CEngine engLSLP(L"LSLP",L"C:\\Program Files\\Frontline Systems\\Engines\\lslpeng.dll"); prob.Engine = prob.Engines [L "engLSLP"]; </pre>
LSSQP	<pre> CEngine engLSSQP(L"LSSQP",L"C:\\Program Files\\Frontline Systems\\Engines\\lssqpeng.dll"); prob.Engine = prob.Engines [L "engLSSQP"]; </pre>
MOSEK	<pre> CEngine engMosek(L"MOSEK",L"C:\\Program Files\\Frontline Systems\\Engines\\mosekeng.dll"); prob.Engine = prob.Engines [L "engMosek"]; </pre>

OptQuest	<pre>CEngine engOptQuest(L"OPTQUEST",L"C:\\Program Files\\Frontline Systems\\Engines\\optquesteng.dll"); prob.Engine = prob.Engines [L "engOptQuest"];</pre>
XPRESS	<pre>CEngine engXpress(L"XPRESS",L"C:\\Program Files\\Frontline Systems\\Engines\\xpresseng.dll"); prob.Engine = prob.Engines [L "engXpress"];</pre>

An alternative way to add an engine is to use the *Discover* function. This function will find any engines located at a given path and add them to the engine collection. If a path is not specified, the SDK will first check the path where the Frontline Solver Engine(s) was installed. If a Frontline Solver Engine(s) is not found, the SDK will search for the engine(s) in the same location where the SolverSDK.dll is installed.

Note: The default installation path for the Frontline Solver Engines is: C:\Program Files\Frontline Systems\Engines. In V12.5 and earlier, the default installation path for the Frontline Solvers Engines was C:\Program Files\Frontline Systems\Analytic Solver SDK\Bin.

To select the engine, simply pass the name. In the examples below, we are selecting the Gurobi Solver installed in the path: C:\Program Files\Frontline Solvers\Engines. Frontline Systems offers seven additional Large Scale engines: KNITRO Solver, Large Scale GRG Solver, Large Scale LP Solver, Large Scale SQP Solver, Mosek Solver, OptQuest Solver, and Xpress Solver. Please see below for an example in C++ or click the following links: [C#](#), [Procedural API](#), or [COM](#).

```
ret = prob.Engines.Discover(L"C:\\program Files\\Frontline Systems\\Engines");
```

```
prob.Engine = prob.engines(L"Gurobi Solver");
```

Replace the last line of code above (*prob.Engine = prob.engines("LGurobi Solver");*) with any of the lines of code below to select any of the remaining seven engines.

KNITRO	<pre>prob.Engine = prob.engines(L"KNITRO Solver");</pre>
LSGRG	<pre>prob.Engine = prob.engines(L"Large Scale GRG Solver");</pre>
LSLP	<pre>prob.Engine = prob.engines(L"Large Scale LP Solver");</pre>
LSSQP	<pre>prob.Engine = prob.engines(L"Large Scale SQP Solver");</pre>
MOSEK	<pre>prob.Engine = prob.engines(L"MOSEK Solver");</pre>
OptQuest	<pre>prob.Engine = prob.engines(L"OptQuest Solver");</pre>
Xpress	<pre>prob.Engine = prob.engines(L"Xpress Solver");</pre>

Determining Linearity Automatically in Optimization Models

As explained above, the primary difference between solving a nonlinear problem and solving a linear problem is that you must provide a "callback" routine to evaluate the nonlinear problem functions, at trial points (values for the variables) determined by the Solver SDK. However, you do not have to initialize the arrays of coefficients passed to the callback function with values.

It is possible to use a callback routine for any Solver problem – even if it is an entirely linear problem. If you do choose to setup a linear problem using an evaluator, there will be a slight speed penalty as the engine will have to call the evaluator a number of times to compute the objective and constraint values.

If you are solving a specific problem or class of problems, you will probably know in advance whether your problem is linear or nonlinear. If you know whether each variable occurs linearly or nonlinearly in the objective and each constraint function, you can supply this information to the Solver through *prob.Model.AllGradDepend()*. (Such information can be used by advanced nonlinear solution algorithms to save time and/or improve accuracy of the solution.)

If you are solving a general series of problems, however, you might have some nonlinear and some linear problems, all being computed by callback routines. *prob.Model.DependTest()* allows you to ask the Solver to determine, through a numerical test, whether the problem is linear or nonlinear. In addition, *prob.Model.DependTest()* computes the information you would otherwise have to supply through a call to *prob.Model.AllGradDepend()*, and if the problem is entirely linear, *Model.DependTest()* computes the LP coefficients and places them in an array. You can then switch from the nonlinear Solver to the linear Simplex method by calling *prob.Engine = prob.Engines[CEngine::LPQPName]*, and then calling *prob.Solver.Optimize()* to solve.

It is also possible to diagnose a model using *prob.Solver.Optimize(Solve_Type_Analyse)*. See the section *Loading and Solve Excel Optimization Models* above for more information.

Diagnosing Infeasible Optimization Problems

When a call to *problem.Solver.Optimize()* finds no feasible solution to your optimization problem, the Solver SDK returns a status value indicating this result when you call *problem.Solver.OptimizeStatus()*. This means that there is no combination of values for the variables that will satisfy all of the constraints (and bounds on the variables) at the same time. If your model correctly describes the real-world problem, it may be that no solution is possible unless you can find a way to relax some of the constraints. But more often, this result means that you made a mistake in specifying some constraint upper or lower bound i.e. supplying an upper bound of 0 and a lower bound of 100.

If you have many constraints, it can be difficult to determine which of them contains a mistake, or conflicts with some other (combination of) constraints. To aid you, the Solver SDK includes a facility to find a subset of your 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. Such a subset of constraints is called an Irreducibly Infeasible Set (IIS) of constraints.

(Note: If you are solving a mixed integer problem and your model is found infeasible, your first step in diagnosing the problem should be to try to solve the "relaxation" of the MIP problem, ignoring the integer restrictions on the variables. You can do this by calling *problem.Solver.Optimize(Solve_Type_NoIntegers)*. (If you do not pass an argument after *problem.Solver.Optimize*, the Solver will solve the complete problem.) If the relaxation of the original problem is still infeasible, you can use the API calls described in this section to isolate the infeasibility.

To find an Irreducibly Infeasible Set of constraints, call the routine *problem.Solver.IISFind()*. This routine returns the indices of the constraints and the indices of the variable bounds that are included in the IIS. If you have both lower and upper bounds on the same variable(s), the SDK will tell you which bound is contributing to the infeasibility.

In general, when a model is infeasible, there can be more than one subset of constraints (possibly many subsets) that qualify as an IIS. Some of these will contain fewer constraints than others, making them easier to analyze. However, finding the "minimal-size" IIS can be computationally very expensive; hence, the IIS finder is designed to find an IIS that contains as few constraints (rows) as possible in a reasonable amount of time. Since variable bounds are easier to analyze than full constraints, the IIS finder favors fewer rows over fewer bounds.

Solution Properties of Quadratic Optimization Problems

If calling the Solver SDK Platform to solve a quadratic problem, please read on. If calling the Solver SDK Pro to solve a quadratic problem, please refer to the section above, *Solving Nonlinear Optimization Problems*, for steps on how to setup and solve a smooth, nonlinear model.

A quadratic programming (QP) problem is one in which the objective is a quadratic function, and the constraints are all linear functions of the variables. A general quadratic function may be written as the sum of a quadratic term $x^T Q x$ and a linear term $c x$:

$$F(x) = x^T Q x + c x$$

The matrix Q is the Hessian (matrix of second partial derivatives) of the objective function. Because the function is quadratic or second degree, all elements of this matrix are constant. You supply the elements of the Q matrix when you call `problem.Model.FcnQuadratic[Function_Type_Objective]`, and the elements of the c vector when you call `problem.Model.AllLinear[Function_Type_Constraint]`.

Depending on the properties of the Q matrix, a quadratic function may have one, many, or no optimal (minimum or maximum) values. If the Q matrix is *positive definite* (for a minimization problem; *negative definite* for a maximization problem), the function will have a "bowl" shape and a single optimal solution (a *strong minimum*). If the Q matrix is *positive semi-definite*, the function will have a "trough" and (infinitely) many optimal solutions, all with the same objective function value (a *weak minimum*). If the Q matrix is *indefinite*, the function will have a "saddle point" (which has many, but not all, of the properties of an optimal solution), however the true optimal solution(s) – one or many of them – will lie somewhere on the boundaries of the constraints.

Quadratic programming algorithms are specialized (for speed and accuracy) to solve problems where the Q matrix is positive definite (when minimizing) or negative definite (maximizing). The QP algorithm used in the Solver SDK Platform is somewhat more general: It can handle problems where the Q matrix is positive semi-definite (or negative semi-definite), in which case it will converge to a point in the "trough" with the correct minimum (maximum) objective function value. If applied to a problem where the Q matrix is *indefinite*, however, the Solver SDK Platform may converge either to a saddle point, or to (one of) the optimal solution(s) on the constraint boundary – depending on the initial values of the variables. In this case a call to the `Optimize_Status` will return a status code of `No_Remedies` to indicate that the solution is not necessarily optimal.

Most problems based on real-world data yield a Q matrix which is positive definite. For example, if you are solving a portfolio optimization problem where the Q matrix represents variances and covariances of pairs of securities calculated from a historical price series, it can be shown that the matrix will be positive definite if the number of observations in the price series is greater than the number of variables.

If, however, you are solving problems based on arbitrary user-provided data, you should take care to diagnose the model by calling the function, `problem.Solver.Optimize(Solver_Type_Analyze)` and then checking the value of `problem.Solver.OptimizeStatus()`. If the model is diagnosed as nonconvex, you may wish to use the nonlinear Solver "engine" instead of the quadratic Solver "engine" – but you must bear in mind that either solution algorithm will find only a local optimum which is not necessarily the global optimum.

The `problem.VarDecision.InitialValue()` function provides initial values for the variables. If the problem is positive (negative) semi-definite or indefinite, these initial values will influence the path taken by the solution algorithm and the final solution to which it converges. If you must solve problems of this type, you can use the `problem.VarDecision.InitialValue()` to exercise some control over the solutions returned by the Solver SDK Platform.

Solving Simulation Problems

The Solver SDK includes a Monte Carlo Simulation engine to solve problems that contain uncertainty. You define and solve a simulation problem in much the same way as you would for a problem

designed to solve a smooth nonlinear optimization problem, by calling an evaluator to compute the uncertain function values. In the example below, the evaluator, *SIMEvaluator*, computes values for the uncertain functions for any given values of the uncertain variables.

```
Engine_Action SIMEvaluator_Evaluate(...)
{
/* Receive values for the variables, compute values */
/* for the constraints and the objective function */
}

int main()
{
...

/* Get data and set up an NSP problem */
//pass the number of variables in the model
prob.VarUncertain.Size = nvars;
//pass the number of constraints in the model
prob.FcnUncertain.Size = ncons;
//pass the distributions of the 2 uncertain variables
prob.Model.VarDistribution[0] = CDistribution(L"PsiUniform(0,2)");
prob.Model.VarDistribution[1] = CDistribution(L"PsiNormal(1,1)");
//call the evaluator to compute the uncertain functions
prob.Evaluators[Eval_Type_Sample].OnEvaluate +=
new EvaluateEventHandler[SIMEvaluator_Evaluate];
//solve the model
prob.Solver.Simulate();
//get the simulation status and the uncertain variables and
functions
//statistics and percentiles
_Simulate_Status status = prob.Solver.SimulateStatus;
CStatistics VarStat = prob.VarUncertain.Statistics;
CDoubleMatrix VarPct = prob.VarUncertain.Percentiles;
CStatistics FcnStat = prob.Functions[0].Statistics;
CDoubleMatrix FcnPct = prob.Functions[0].Percentiles;
...
}
```

Your program should first create a problem, *Problem prob = new Problem*, and then call *prob.Solver.Simulate()* to run a simulation. You may optionally call *prob.Model.DistCorrelations = CDoubleMatrix(Array_Order_ByRow, nvars, nvars, correl)* to pass a correlation matrix. It is also possible to embed an optimization problem inside of the simulation evaluator which will be solved on each simulation trial.

To retrieve the solution status, and the statistics for the uncertain variables and functions (such as mean, minimum, maximum etc.) call *prob.Solver.SimulateStatus()*, *prob.VarUncertain.Statistics*, and *prob.FcnUncertain[].Statistics*, respectively. As with optimization problems, this cycle may be repeated to solve a series of Simulation problems.

For more information on evaluators and solving simulation problems, see the next chapter *Solver SDK Platform Evaluators*.

Passing Sparse Matrices

DoubleMatrix Object

In most large optimization problems, the constraint coefficient matrix (the Jacobian) (and – in some quadratic problems – the Hessian) are *sparse* – meaning that most of the matrix elements are zero. For example, in many larger problems there are groups of constraints that are functions of a small subset of the variables, but that do not depend on any of the other variables – leading to many zero coefficients in the constraint matrix.

When these matrices are sparse, it is more efficient to store – and to process – only the nonzero matrix elements. The SDK includes a Double Matrix object that allows you to easily create memory-efficient, large, sparse matrices by supplying the row and column indices of the nonzero elements. This means that no storage at all is needed for the zero elements, which can include 90% to 95% of all elements in large, sparse problems. So, if you create a matrix with one million rows and one million columns, it will take very little memory initially. You can then assign nonzero elements to the matrix by writing assignment statements, treating the DoubleMatrix object much like an ordinary two-dimensional array.

DoubleMatrix Methods

When passing the objective and constraint matrices in sparse form, you must pass two double matrices, one containing the coefficients for the objective (*problem.Model.AllLinear[Function_Type_Objective]*) and a second for the coefficients of the constraints (*problem.Model.AllLinear[Function_Type_Constraint]*).

The **Create** method creates a structure for a new matrix of the specified size, but does not initialize any of its elements. The **InitDense** method creates a new matrix whose storage is optimized for the situation where most elements are non-zero – i.e. the matrix is dense. The **InitSparse** method creates a new matrix whose storage is optimized for the situation where most elements are zero – i.e. the matrix is sparse. In both cases, the matrix is initialized with values from the Elements array, taking these elements in the ArrayOrder order (initially in "column major" order). Method **Clear** removes all non-zeros from the matrix, but preserves its size; method **Destroy** removes all non-zeros and also sets the number of rows and columns to zero.

```
DoubleMatrix.Create Rows, Columns
```

```
DoubleMatrix.InitDense Rows, Columns, Elements
```

```
DoubleMatrix.InitSparse Rows, Columns, Elements
```

```
DoubleMatrix.Clear
```

```
DoubleMatrix.Destroy
```

As an example, consider the following sparse constraint matrix:

1.2	0.0	3.4
0.0	5.6	0.0
7.8	0.0	0.0
0.0	9.0	0.0

Here, the number of variables (*nvars*) = 3, the number of constraints (*ncons*) = 4 and there are 5 nonzero elements. To pass this matrix in sparse form, you would first create a *DoubleMatrix* object and then pass each element directly.

```
DoubleMatrix fcnConstr(ncons,nvars)

fcnConstr.Value[0][0] = 1.2;
fcnConstr.Value[0][2] = 3.4;
fcnConstr.Value[1][1] = 5.6;
fcnConstr.Value[2][0] = 7.8;
fcnConstr.Value[3][1] = 9.0;
```

The first line creates the *DoubleMatrix*, *fcnConstr*, of size *ncons* x *nvars*. *Nvars* is the number of variables (3) and *ncons* (4) is the number of constraints. The remaining lines of code pass the nonzero elements directly to the *fcnConstr* *DoubleMatrix*. As you can see, no zero elements are passed. Each time an element is passed to the *DoubleMatrix*, several internal arrays are updated in the SDK.

To speed up the matrix population, you can use the optional functions, *NZBgn()* and *NZEnd()*. These functions instruct the SDK not to update the *DoubleMatrix* until all elements have been collected. When populating a large matrix, this will result in a significant speed improvement. It is important to note that you should not use this matrix between the calling of *NZBgn()* and *NZEnd()* as the matrix will not be populated until *NZEnd()* is executed. If you need to 'undo' the effect of *NZBgn*, call **NZCancel** (instead of *NZEnd* – in this case, all of the assignments since *NZBgn* was called will have no effect).

```
DoubleMatrix fcnConstr(ncons,nvars)

fcnConstr.NZBgn();

fcnConstr.Value[0][0] = 1.2;
fcnConstr.Value[0][2] = 3.4;
fcnConstr.Value[1][1] = 5.6;
fcnConstr.Value[2][0] = 7.8;
fcnConstr.Value[3][1] = 9.0;

fcnConstr.NZEnd();
```

The **MakePSD** method can be used on a non-positive semidefinite (non-PSD) matrix to transform the matrix into a positive semidefinite (PSD) matrix that is "close to" the original matrix. This is most often used for correlation matrices in simulation, but may also be used in certain optimization problems. The *Weights* argument is optional; if it is specified, the elements of the transformed matrix with the largest weights in the corresponding positions of the *Weights* matrix will be as close as possible to the original matrix elements, and the elements with the smallest weights will experience the greatest change in the transformed matrix.

```
DoubleMatrix.MakePSD [Weights]
```

Using the Solver SDK in Multi-Threaded Applications

The Solver SDK is multi-threaded and fully reentrant. The SDK can solve multiple problems concurrently: You may setup and solve several *problems* by creating multiple instances of the problem

class. You can freely mix calls to any of the Solver SDK routines simply by calling the appropriate instance of `CProblem`.

The Solver SDK may be used to solve decomposition problems (which involve both a "master problem" and a "slave problem"), or any recursive problem where the computation of the objective or constraints for a main problem requires the solution of another optimization subproblem. The multi-threaded versions may also be used in Web server or Intranet applications, where the application calling the Solver SDK consists of several threads executing concurrently, each thread dealing with a specific user session.

One sequence of calls that is illegal in every version of the Solver DLL is a call to *problem.Solver.Optimize()* for a given problem, then a second call to *problem.Solver.Optimize()* for that same problem, made before the first call returns (e.g. made from within your *evaluator()* or other callback routine).

Memory Usage

The Solver SDK allocates memory to store problem data when a problem is created. This information is maintained until the problem is destroyed. Memory is allocated in two ways, on demand (before *prob.Solver.Optimize()*) and automatically (after *prob.Solver.Optimize()*).

Before the call to *prob.Solver.Optimize()*, memory is allocated on demand. For example, the memory for a sparse matrix is allocated when the user sets the first nonzero element. Memory is allocated automatically after the call to *prob.Solver.Optimize()* so that the user can retrieve solution and sensitivity data, for example, dual values, final values, etc. The allocated memory is freed when the user destroys the problem. (Note: Our Legacy API will still store the pointers and later copy the data from the allocated problem into the user allocated data.)

Solver SDK Evaluators

Introduction

The Solver SDK employs the use of evaluators or callback functions for two main reasons: To check the progress of the Solver during the optimization or simulation process (Progress Information Evaluators) or to provide the crucial role of computing constraint function values or passing gradient or Hessian values (Solver SDK Platform only) when solving an optimization problem (most commonly nonlinear or nonsmooth problems) or computing uncertain function values when solving a simulation problem (Computing Evaluators).

Once you have setup your optimization or simulation problem, you can *optionally* call a Progress Information Evaluator to check the simulation trial or optimization iteration number, report the current objective, check for a user abort code, etc. If you are solving a nonlinear or nonsmooth optimization model, a Computing Evaluator is *required* to calculate the objective or constraint function values or if solving a simulation problem to compute the uncertain function values. It is also possible, although it is not required, to pass exact first and second order derivatives to a quadratic or smooth, nonlinear problem. Regardless of what the evaluator does, it is always created using the same methodology. This chapter explains the types of optimization and simulation evaluators offered by the Solver SDK and gives an example of each.

Note: When using an evaluator in a multi-threaded application on a multiple core machine, you must ensure that your evaluators are threadsafe.

If you encounter any problems in creating or writing any one of these evaluators, please contact our Technical Support desk by emailing Support@solver.com or calling 888-831-0333.

Using an Evaluator to Check the Progress of an Optimization

An evaluator can be used to obtain control from the Solver SDK during, before, or after an optimization to check the progress of the solution process, to display a message, etc. For an optimization, there are five such evaluators available of type: Eval_Type_Iteration, Eval_Type_NewSolution, Eval_Type_OptBegin, Eval_Type_Opt_End and Eval_Type_Optimization. Please see the table below for an explanation of each. These types of progress information evaluators are **always** optional.

Optimization Progress Information Evaluators

Eval_Type_Iteration	Called on each optimization iteration. Can be used to track the progress

Eval_Type_NewSolution	(check the objective function, number of iterations, etc) of the optimization. Called each time a new solution is found to a mixed-integer problem (MIP), or nonlinear/nonsmooth optimization model.
Eval_Type_OptBegin	Called at the beginning of an optimization run (of two or more solves).
Eval_Type_OptEnd	Called after the end of an optimization run (of two or more solves).
Eval_Type_Optimization	Called at the end of an optimization.

The evaluator of type *Eval_Type_Iteration* is the most commonly used of these "Progress Information" optimization evaluators. As such, we will use this evaluator in our example below. Note: Each of the evaluators above can be created with the same steps as shown below. Simply substitute *Eval_Type_Iteration* when creating the evaluator with *Eval_Type_X* where *X* is *NewSolution*, *OptBegin*, *OptEnd*, or *Optimization*.

First we create the evaluator in the main body of our code. In this example, the name of our evaluator of type *Eval_Type_Iteration* is *EvalIteration*. As mentioned above, this type of evaluator is called at every iteration of an optimization and can be used to obtain control during the solution process in order to display a message, check for a user abort action, etc.

```
int main()
{
    /* Get data and set up an optimization problem */
    ...
    CProblem prob
    /* Pass the number of variables, constraints, pass constraint upper
    and lower bounds, variable upper and lower bounds, etc. */
    CIterEvaluator EvalIteration;
    prob.Evaluators[Eval_Type_Iteration].Evaluator = & EvalIteration;
```

The example code below gives an example of an evaluator of type *Eval_Type_Iteration* which writes the iteration number and the current objective function value to the C++ Console on each iteration. This will allow the User to keep watch as the Solver progresses to a solution.

```
class CIterEvaluator : public IEvaluator
{
public:
    virtual _Engine_Action Evaluate(CEvaluator& evaluator)
    {
        cout << "Iteration " <<
            evaluator.Problem.Engine.Stat.Iterations << endl;
        cout << "Objective = " <<
            evaluator.Problem.FcnObjective.Value[0]<< endl;
        return Engine_Action_Continue;
    }
}
```


Note: If, for any reason, you want the Solver SDK to stop optimizing the model, then you should return *Engine_Action_Stop* inside of a progress evaluator, not inside an evaluator of type *Function*, *Jacobian* or *Hessian*.

This completes the section explaining how to construct an optional evaluator to gain control during the optimization process. Please click the following links for more example code in [C#](#), the [Procedural API](#) or in [COM](#).

Using a Function Evaluator to Compute the Objective and Constraints

An evaluator can be used to compute the objective function and constraints of an optimization problem. If you are solving a linear problem, you can either use an evaluator to compute the objective function and constraints, or you can call *AllLinear* to supply the objective and constraint coefficient matrices directly to the SDK. (For more information, please see the *Creating a Linear Problem* sections in each of the next four chapters.) If you are solving a nonlinear or nonsmooth model, you **must** provide an evaluator to compute the constraints and objective function.

In a linear model or a quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices using *FcnQuadratic* for a quadratic objective and *AllLinear* for a linear objective and/or constraints. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. In a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. As a result, you must use an evaluator to pass the objective and constraint functions, specifically an evaluator of type *Eval_Type_Function*.

The Solver uses an evaluator of type *Eval_Type_Function* (i) to compute values for the problem functions at specific trial points as it seeks an optimum, and (ii) to compute estimates of the partial derivatives of the objective (the gradient) and the constraints (the Jacobian). (It is possible to pass these derivatives directly to the SDK using the *Eval_Type_Gradient* evaluator. For more information, see the subheading, *Calling an Evaluator to Supply the Jacobian* below.)

The partial derivatives are estimated by a "rise over run" calculation, in which the value of each variable in turn is perturbed, and the change in the problem function values is observed. For a problem with N variables, the Solver SDK will call *Eval_Type_Function()* N times on each occasion when it needs new estimates of the partial derivatives (2*N times if the "central differencing" option is used). This often accounts for 50% or more of the calls to the evaluator during the solution process.

We must first create the evaluator in the main body of our program which calls the Solver SDK. Again, the Solver will call the evaluator of type *Eval_Type_Function* to compute the constraints and objective function values and also to generate partial derivatives. The name given to the evaluator of type *Eval_Type_Function* is *evalNLP*.

```
int main()
{
    /* Get data and set up an optimization problem */
    ...
    CProblem prob
    /* Pass the number of variables, constraints, pass constraint upper
    and lower bounds, variable upper and lower bounds, etc. */
    CNLPEvaluator evalNLP;
```

```
prob.Evaluators[Eval_Type_Function].Evaluator = & evalNLP;
```

The evaluator below is an example of an evaluator that could be called to compute an objective and two constraints for a simple nonlinear optimization problem.

```
Objective: x^2 + y^2
Constraints:  x + y
             x * y
```

The first line of code creates a *Problem* and a convenient pointer. The next two lines of code setup pointers to the decision variables (*pVar*) and the constraints (*pFcn*). These pointers make the code more readable but are purely optional.

```
CProblem& prob = evaluator.Problem;
const double* pVar = prob.VarDecision.Value;
double* pFcn = prob.FcnConstraint.Value;
```

These next two lines of code compute the constraints, $x + y$ and $x * y$.

```
pFcn[0] = pVar[0] + pVar[1];
pFcn[1] = pVar[0] * pVar[1];
```

The line of code below computes the objective function $x^2 + y^2$.

```
prob.FcnObjective.Value[0] = pVar[0] * pVar[0] + pVar[1] * pVar[1];
```

Finally, this last line of code tells the Solver SDK to continue solving. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the functions (objective or constraints). The Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine will disregard the current point and continue solving.

```
return Engine_Action_Continue;
```

Complete Evaluator

Please see below for a complete snapshot of the code required for our evaluator of type *Eval_Type_Function*.

```
class CNLPEvaluator : public IEvaluator
public:
    virtual _Engine_Action Evaluate (CEvaluator& evaluator)
    {
        CProblem& prob = evaluator.Problem;
        const double* pVar = prob.VarDecision.Value;
        double* pFcn = prob.FcnConstraint.Value;
        pFcn[0] = pVar[0] + pVar[1];
        pFcn[1] = pVar[0] * pVar[1];
        prob.FcnObjective.Value[0] = pVar[0] * pVar[0] + pVar[1] *
            pVar[1];
        return Engine_Action_Continue;
    }
}
```

This is all that is required to solve a problem with an evaluator (of type *Eval_Type_Function*) to compute the constraints and objective function for an optimization problem. For example code in additional languages, please click the appropriate link: [C#](#), [Procedural API](#), [COM](#). Please read on for more information on supplying partial derivatives using an *optional* evaluator of type *Eval_Type_Gradient*.

Using a Gradient Evaluator to Provide First Order Partial Derivatives

To speed up the solution process for nonlinear problems, and to give the Solver SDK Platform exact partial derivatives (as opposed to estimates of the partial derivatives computed by the Solver SDK), you can supply a second evaluator of type *Eval_Type_Gradient* which returns values for all elements of the objective gradient and constraint Jacobian matrix in one call. (The Solver SDK Pro does not support an evaluator of this type.) This callback function is *optional*, but if it is present, the Solver SDK Platform will call it instead of making repeated calls to the *Eval_Type_Function()* evaluator to evaluate partial derivatives. The *Eval_Type_Function* evaluator must still be present as it will still be called to compute the objective function and constraint values each time a new point is tried. However, the number of calls to this evaluator will be drastically reduced since it will no longer be used to compute the partial derivatives.

We must first create the evaluator in the main body of our program which calls the Solver SDK Platform. The name given to the evaluator of type *Eval_Type_Gradient* is *evalJac*.

```
int main()
{
    /* Get data and set up an optimization problem */
    ...
    CProblem prob
    /* Pass the number of variables, constraints, pass constraint upper
    and lower bounds, variable upper and lower bounds, etc. */
    CNLPEvaluator evalNLP;
    prob.Evaluators[Eval_Type_Function].Evaluator = & evalNLP;
    CJacEvaluator evalJac;
    prob.Evaluators[Eval_Type_Gradient].Evaluator = & evalJac;
```

Below is the Jacobian evaluator of type *Eval_Type_Gradient* for the example evaluator above, *evalNLP*, with the objective = $x^2 + y^2$ and the constraints: $x + y$ and $x * y$. When passing the Jacobian, you must first create two double matrices to hold the partial derivatives of the objective and constraints. The first double matrix, *objGrad*, will hold the partial derivatives of the objective function and the second double matrix, *matGrad*, will hold the partial derivatives of the constraints.

First, we create a double matrix to hold the partial derivatives for the objective function, $x^2 + y^2$.

```
CDoubleMatrix objGrad =
    evaluator.Problem.Model.AllGradValue[Function_Type_Objective]
    [Variable_Type_Decision];
```

The line of code below passes the partial derivative for x^2 (which is $2x$),

```
objGrad.Value[0, 0] = 2.0 * evaluator.Problem.VarDecision.Value[0];
```

and the line of code below passes the partial derivative for y^2 (which is $2y$).

```
objGrad.Value[0, 1] = 2.0 * evaluator.Problem.VarDecision.Value[1];
```

Next we create a double matrix to hold the partial derivatives for the constraints, $x + y$ and $x * y$.

```
CDoubleMatrix matGrad =  
    evaluator.Problem.Model.AllGradValue[Function_Type_Constraint]  
    [Variable_Type_Decision];
```

The next two lines of code pass the partial derivatives for the first constraint, $x + y$. The partial derivative of x is 1 and the partial derivative of y is also 1.

```
matGrad.Value[0, 0] = 1.0;  
matGrad.Value[0, 1] = 1.0;
```

While the next two lines of code pass the partial derivatives for the second constraint, $x * y$. The first partial derivative is y and the second is x .

```
matGrad.Value[1, 0] = evaluator.Problem.VarDecision.Value[1];  
matGrad.Value[1, 1] = evaluator.Problem.VarDecision.Value[0];
```

Finally, we instruct the Solver SDK Platform to continue solving.

```
return Engine_Action.Continue;
```

Complete Evaluator

Please see below for a complete snapshot of the code required for our evaluator of type *Eval_Type_Gradient*.

```
class JacEvaluator : public IEvaluator  
{  
    public:  
        virtual _Engine_Action Evaluate (CEvaluator& evaluator)  
        {  
            CDoubleMatrix objGrad =  
                evaluator.Problem.Model.AllGradValue  
                [Function_Type_Objective][Variable_Type_Decision];  
            objGrad.Value[0, 0] = 2.0 *  
                evaluator.Problem.VarDecision.Value[0];  
            objGrad.Value[0, 1] = 2.0 *  
                evaluator.Problem.VarDecision.Value[1];  
            CDoubleMatrix matGrad =  
                evaluator.Problem.Model.AllGradValue  
                [Function_Type_Constraint][Variable_Type_Decision];  
            matGrad.Value[0, 0] = 1.0;  
            matGrad.Value[0, 1] = 1.0;  
            matGrad.Value[1, 0] = evaluator.Problem.VarDecision.Value[1];  
            matGrad.Value[1, 1] = evaluator.Problem.VarDecision.Value[0];  
  
            evaluator.Problem.Model.AllGradValue[Function_Type_Constraint]  
                [Variable_Type_Decision] = matGrad;  
            return Engine_Action.Continue;  
        }  
}
```

Writing a function that computes the 2nd order derivative matrix (Jacobian) can be difficult – it is easy to make errors in computing the various partial derivatives. To aid in debugging, Solver SDK has the ability to call the *Eval_Type_Gradient()* evaluator and compute its own estimates of the partial derivatives by calling the evaluator *Eval_Type_Function*. Solver SDK will compare the results and call a third evaluator, *Eval_Type_GradError*, which can be used to display an error message for mismatching partial derivatives. In this case, you are *required* to create *three* types of evaluators, *Eval_Type_Function*, *Eval_Type_Gradient*, and *Eval_Type_GradError* in the main body of your code. If only *Eval_Type_Function* and *Eval_Type_Gradient* are present, Solver SDK will *not* compute its own estimates of the partial derivatives.

The Evolutionary Solver "engine," which is used if any of your problem functions are nonsmooth or discontinuous (as indicated by *prob.Model.DependTest()*), does not make use of the *Eval_Type_Gradient()* function, even if you call the evaluator in your program.

This completes the section explaining how to construct an optional evaluator to pass the first order derivatives of the objective and constraints when solving a nonlinear optimization problem. Please click the following links for more example code in [C#](#), the [Procedural API](#) or in [COM](#).

Using a Hessian to Provide Second Order Partial Derivatives

If using the Solver SDK Platform, an evaluator of type *Eval_Type_Hessian* can be called to pass exact second order partial derivatives for a quadratic, quadratically constrained or nonlinear problem. In a quadratic problem, an evaluator of type *Eval_Type_Hessian* can be called to compute the Q-matrix for the objective. In a quadratically constrained problem this type of evaluator can be called to compute the Q-matrix for a quadratic constraint. (Most times this is because the Q-matrix is not constant. However, it is also possible to pass a constant Q-matrix in this way.) In a nonlinear problem, an evaluator of type *Eval_Type_Hessian* can be called to pass exact second order partial derivatives much like the evaluator of type *Eval_Type_Gradient* is used to pass exact first order partial derivatives.

Each function in the problem (objective and constraints) will have it's own Hessian matrix, although the matrix could contain all zeros (for example, if all terms in the function are linear). The size of a Hessian matrix will always be number of variables x number of variables.

Solving a Quadratic Problem with a Hessian Evalautor

If supplied to a quadratic or quadratically constrained problem, an evaluator of type *Eval_Type_Hessian* will be called to compute a Q-Matrix that is in the objective or in a constraint. If the Q-Matrix is constant, then it is possible to use the *FcnQuadratic* method to pass the Q-matrix rather than with an evaluator of type *Eval_Type_Hessian*. However, if the Q-Matrix is not constant, then you *must* supply the Q-Matrix via the evaluator of type *Eval_Type_Hessian*.

In the Solver SDK Platform, the LP/ Quadratic and SOCP Solver engines will ask for the Hessian matrix when solving either a quadratic or quadratically constrained problem. If you are solving a quadratic problem, you must set the problem type to *Problem_Type_OptQP* before calling *prob.Solver.Optimize()*. If you are solving a quadratically constrained problem, then you must set the problem type to *Problem_Type_OptQCP* and then specify how each variable occurs (linearly or quadratically) in both the objective and constraints by using the *AllGradDepend* method.

The example code below solves a quadratic investment portfolio model. The mathematical representation of this model is below.

Minimize: Portfolio Variance ($x^T Q x + c x$) where the Stock Variance/Covariance Matrix is contained in the table below and there are 0 linear coefficients.

	Stock 1 (x1)	Stock 2 (x2)	Stock 3 (x3)	Stock 4 (x4)	Stock 5 (x5)
Stock 1 (x1)	.000204	.000424	.000170	.000448	-.000014

Stock 2 (x2)	.000424	.012329	.001785	0.001633	-0.000539
Stock 3 (x3)	0.000170	0.001785	0.000365	0.000425	-0.000075
Stock 4(x4)	0.000448	0.001633	0.000425	0.005141	0.000237
Stock 5 (x5)	-0.000014	-0.000539	-0.000075	0.000237	0.000509

Subject to: $X1 + X2 + X3 + X4 + X5 = 1$
 $.086X1 + .0714X2 + .095X3 + .107X4 + .069X5 \geq .085$
 $0 \leq X1, X2, X3, X4, X5 \leq 1$

This model contains 5 variables and 2 constraints. The variables are the percentages to be allocated to each investment or asset class and are $x1, x2, x3, x4$, and $x5$. These variables may not be less than 0 or greater than 1. The first constraint requires that the total of all the variables must equal 1 ($x1 + x2 + x3 + x4 + x5 = 1$). The second constraint ensures that the portfolio return must be greater than or equal to 8.5%. The objective function is minimizing the variance of the portfolio. In linear algebra notation this function can be written as $x^T Q x + c x$ where x is an array of n decision variables, Q is an $n \times n$ matrix of coefficients, and c is an n vector of linear coefficients. (In this example, all linear coefficients are 0.)

We must first create the evaluator in the main body of our program which calls the Solver SDK Platform. Note that when using an evaluator of type *Eval_Type_Hessian* to compute the Q-Matrix you do not need to also pass an evaluator of type *Eval_Type_Function*. The name given to the evaluator of type *Eval_Type_Hessian* is *evalHess*.

To begin, the example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables (*nvars*) and constraints (*ncons*) in the model.

```
int nvars = 5, ncons = 2
```

Next we create and initialize several arrays. The *objcoeff* array is of size *nvars* (number of variables or 5) and will hold the linear coefficients of our objective function. In this example, all five linear coefficients are 0. This means that there are no linear terms in our objective function.

```
double objcoeff[] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
```

The *ub_constraints* array is of size *ncons* (number of constraints or 2) and holds the upper bounds of the two constraint right hand sides. Note: *CConstants::PINF* is a C++ constant for "positive infinity".

```
double ub_constraints[] = { 1, CConstants::PINF };
```

The *lb_constraints* array is of size *ncons* and holds the lower bounds of the two constraint right hand sides. Notice that the lower and upper bounds of the first constraint are both 1 to denote an equality constraint ($x1 + x2 + x3 + x4 + x5 = 1$).

```
double lb_constraints[] = {1.0, 0.085};
```

The *concoeff* array is of size number of variables x number of constraints ($5 * 2$) and holds the constraint coefficients. This array of coefficients will be passed "row wise". Therefore, the first five coefficients belong to the first constraint (total of all variables must equal 1) and the last five coefficients belong to the second constraint (portfolio return).

```
double concoeff[] = { 1.0, 1.0, 1.0, 1.0, 1.0, 0.086, 0.0714, 0.095,
                    0.107, 0.069 };
```

The following two arrays are both of size *nvars* and hold the decision variables lower bounds (*lb*) and upper bounds (*ub*).

```
double lb[] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
```

```
double ub[] = { 1.0, 1.0, 1.0, 1.0, 1.0 };
```

The next line of code creates an instance of the CProblem class (*prob*), passes the Solver Type (*Solver_Type_Minimize*) and also the number of decision variables (*nvars*) and constraints (*ncons*) to

the Solver SDK. (For more information on creating an instance of the CProblem class, see the subsequent four chapters on calling the SDK in the programming language of your choice.)

```
CProblem prob(Solver_Type_Minimize, nvars, ncons);
```

The next two lines of code pass the constraint upper and lower bounds, respectively, to the SDK.

```
problem.FcnConstraint.UpperBound    = rhs;
problem.FcnConstraint.LowerBound    = lhs;
```

While the next two lines of code pass the decision variable upper and lower bounds, respectively, to the SDK.

```
problem.VarDecision.UpperBound = ub;
prob.VarDecision.LowerBound = lb;
```

The two lines of code below call *prob.Model.AllLinear[Function_Type_Objective]* to pass the linear terms of the objective coefficient matrix, *objcoeff*, and *prob.Model.AllLinear[Function_Type_Constraint]* to pass the constraint coefficient matrix, *concoeff*.

```
problem.Model.AllLinear[Function_Type_Objective][Variable_Type_Decision]=CDoubleMatrix(Array_Order_ByCol, 1, nvars, objcoeff);
problem.Model.AllLinear[Function_Type_Constraint][Variable_Type_Decision]= CDoubleMatrix(Array_Order_ByRow, ncons, nvars, concoeff);
```

The line of code below sets the problem type to *OptQP* (Quadratic Problem). If you omit the problem type, the Solver SDK will not call the evaluator. Setting this problem type will automatically choose the LP/Quadratic engine in the Solver SDK Platform to solve the problem.

```
problem.ProblemType = Problem_Type_OptQP;
```

Here we create the evaluator *evalHess* of type *Eval_Type_Hessian*. This evaluator will be called to compute the Q matrix in the objective function of this quadratic problem.

```
CHessEvaluator evalHess;
prob.Evaluators[Eval_Type_Hessian].Evaluator = & evalHess;
```

Finally, we ask the Solver to optimize.

```
Prob.Solver.Optimize();
```

From here, the SDK will call the evaluator below to compute the objective function.

First, we setup a convenient pointer to the Problem, *prob*.

```
CProblem& prob = evaluator.Problem;
```

Then, setup the *qmatval* array and pass the Q-matrix.

```
double qmatval[] =
{ 0.000204,  0.000424,  0.000170,  0.000448, -0.000014,
  0.000424,  0.012329,  0.001785,  0.001633, -0.000539,
  0.000170,  0.001785,  0.000365,  0.000425, -0.000075,
  0.000448,  0.001633,  0.000425,  0.005141,  0.000237,
 -0.000014, -0.000539, -0.000075,  0.000237,  0.000509 };
```

Next, we check the function type of the current function. If the function type is *Function_Type_Objective* (Objective function), then we call the *FcnHessValue* method.

```
if (prob.Solver.FunctionType == Function_Type_Objective)
```

The left side of the equation below calls the *FcnHessValue* method while passing the function type (*Function_Type_Objective*), the function index (always 0 for the objective function) and the Variable type (*Variable_Type_Decision*). The right side of this equation creates a DoubleMatrix of size 5 rows and 5 columns (number of variables x number of variables), indicates that the Q-matrix coefficients will be passed by column (*Array_Order_ByCol*) and then passes the *qmatval* array which holds the coefficients.

```

        prob.Model.FcnHessValue[Function_Type_Objective][0][Variable_Type_Decision] =
            CDoubleMatrix(Array_Order_ByCol, 5, 5, qmatval);

```

Finally, we ask the SDK to continue solving.

```

return Engine_Action_Continue;

```

Complete Evaluator

Please see below for a complete snapshot of the code required for our evaluator of type *Eval_Type_Hessian*.

```

class HessEvaluator : public IEvaluator
{
public:
    virtual _Engine_Action Evaluate (CEvaluator& evaluator)
    {
        CProblem& prob = evaluator.Problem;

        double qmatval[] =
            { 0.000204, 0.000424, 0.000170, 0.000448, -0.000014,
              0.000424, 0.012329, 0.001785, 0.001633, -0.000539,
              0.000170, 0.001785, 0.000365, 0.000425, -0.000075,
              0.000448, 0.001633, 0.000425, 0.005141, 0.000237,
              -0.000014, -0.000539, -0.000075, 0.000237, 0.000509 };

        if (prob.Solver.FunctionType == Function_Type_Objective)
            prob.Model.FcnHessValue[Function_Type_Objective][0]
                [Variable_Type_Decision] =
                CDoubleMatrix(Array_Order_ByCol, 5, 5, qmatval);

        return Engine_Action_Continue;
    }
}

```

For example code in calling an evaluator of type *Eval_Type_Hessian* for a quadratic problem in additional languages, please click the appropriate link: [C#](#), [Procedural API](#), [COM](#).

Solving a Quadratically Constrained Problem with a Hessian Evaluator

When solving a quadratically constrained problem, you must also call the *AllGradDepend* method to pass an array which indicates whether the variables occur linearly or quadratically, in addition to setting the problem type of *OptQCP*, as indicated above. Let's modify the example above and this

time maximize the *expected return* of the portfolio while adding a constraint that keeps the portfolio variance less than 1%. This modified model will have a linear objective, one linear constraint and one quadratic constraint (the variance constraint). For simplicity, we will use the same Q-matrix.

Maximize: Portfolio Expected Return (Expected returns for each stock are contained in the table below) $.07X1 + .08X2 + .095X3 + .065X4 + .15X5$

Stock 1 (x1)	Stock 2 (x2)	Stock 3 (x3)	Stock 4 (x4)	Stock 5 (x5)
8.6%	7.14%	9.5%	10.7%	6.9%

Subject to: $X1 + X2 + X3 + X4 + X5 = 1$

$$(x^T Q x + c x) \geq .01$$

where the Stock Variance/Covariance Matrix is contained in the table below

	Stock 1 (x1)	Stock 2 (x2)	Stock 3 (x3)	Stock 4 (x4)	Stock 5 (x5)
Stock 1 (x1)	.000204	.000424	.000170	.000448	-.000014
Stock 2 (x2)	.000424	.012329	.001785	0.001633	-0.000539
Stock 3 (x3)	0.000170	0.001785	0.000365	0.000425	-0.000075
Stock 4(x4)	0.000448	0.001633	0.000425	0.005141	0.000237
Stock 5 (x5)	-0.000014	-0.000539	-0.000075	0.000237	0.000509

$$0 \leq X1, X2, X3, X4, X5 \leq 1$$

This model contains 5 variables and 2 constraints. The variables are the percentages to be allocated to each investment or asset class and are x1, x2, x3, x4, and x5. These variables may not be less than 0 or greater than 1. The first constraint requires that the total of all the variables must equal 1 ($x1 + x2 + x3 + x4 + x5 = 1$). The second constraint ensures that the portfolio *variance* must be less than or equal to 1%. In linear algebra notation this function can be written as $x^T Q x + c x$ where x is an array of n decision variables, Q is an $n \times n$ matrix of coefficients, and c is an n vector of linear coefficients. (In this example, all linear coefficients are 0.) The objective function is maximizing the return of the portfolio.

To begin, the example code will again initialize two variables, *nvars* and *ncons*, to pass the number of variables (*nvars*) and constraints (*ncons*) in the model.

```
int nvars = 5, ncons = 2;
```

Again, we will setup the same arrays, but in this example, our *objcoeff* array coefficients will be nonzero to reflect the stock returns ($.086x1 + .0714x2 + .095x3 + .107x4 + .069x5$).

```
double objcoeff[] = { 0.086, 0.0714, 0.095, 0.107, 0.069 };
```

The arrays *ub_constraints* and *lb_constraints* change to the following to reflect the 2nd constraints left hand side of " ≤ 0.01 ".

```
double ub_constraints[] = {1.0, 0.01 };
```

```
double lb_constraints[] = {1.0, CConstants::MINF };
```

The *lb* and *ub* arrays remain the same.

```
double lb[] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
```

```
double ub[] = { 1.0, 1.0, 1.0, 1.0, 1.0 };
```

The *concoeff* array is still of size number of variables x number of constraints ($5 * 2$) and still holds the constraint coefficients. This array of coefficients will be passed "row wise". Therefore, the first five coefficients belong to the first constraint ($x1 + x2 + x3 + x4 + x5 = 1$) and the last five coefficients

belong to the second constraint (Portfolio Return). Notice that the coefficients for our last constraint are now all zero as this constraint has been changed from a linear constraint in the example above to a quadratic constraint in this example.

```
double concoeff[] = { 1.0, 1.0, 1.0, 1.0, 1.0, 0, 0, 0, 0, 0 };
```

This next line of code creates an array, *concoefftype*, of type *_Depend_Type* (type defined within the Solver SDK) which contains information on how each variable occurs in the two constraints, linearly or quadratically. This array will be passed "by row" so the first five array elements belong to the first constraint and the last five belong to the second constraint.

```
_Depend_Type concoefftype[] =
{Depend_Type_Linear,Depend_Type_Linear,    Depend_Type_Linear,
Depend_Type_Linear,    Depend_Type_Linear,Depend_Type_Quadratic,
    Depend_Type_Quadratic, Depend_Type_Quadratic,
    Depend_Type_Quadratic, Depend_Type_Quadratic};
```

The next line of code creates an instance of the CProblem class (*prob*), passes the Solver Type (*Solver_Type_Minimize*) and also the number of decision variables (*nvars*) and constraints (*ncons*) to the Solver SDK. (For more information on creating an instance of the CProblem class, see the subsequent chapters on calling the SDK.)

```
CProblem prob(Solver_Type_Minimize, nvars, ncons);
```

The next four lines of code remain unchanged passing the constraint upper and lower bounds, and the upper and lower bounds for the decision variables, respectively, to the SDK.

```
problem.FcnConstraint.UpperBound    = rhs;
problem.FcnConstraint.LowerBound    = lhs;
problem.VarDecision.UpperBound = ub;
prob.VarDecision.LowerBound = lb;
```

The two lines of code below call *prob.Model.AllLinear[Function_Type_Objective]* to pass the objective coefficient matrix, *objcoeff*, and *prob.Model.AllLinear[Function_Type_Constraint]* to pass the linear terms of the constraint coefficient matrix, *concoeff*.

```
problem.Model.AllLinear[Function_Type_Objective][Variable_Type_Decision]=CDoubleMatrix(Array_Order_ByCol, 1, nvars, objcoeff);
problem.Model.AllLinear[Function_Type_Constraint][Variable_Type_Decision]= CDoubleMatrix(Array_Order_ByRow, ncons, nvars, concoeff);
```

The next line of code passes the *concoefftype* using the *AllGradDepend* method. The left side of the equation calls the *AllGradDepend* method while passing the Function Type (*Function_Type_Constraint*) and the Variable Type (*Variable_Type_Decision*). The right side of the equation creates a Double Matrix object of size *ncons* (2) by *nvars* (5), then passes the array *concoefftype* "row-wise". This line of code is required when computing the Q-matrix inside of an evaluator.

```
problem.Model.AllGradDepend[Function_Type_Constraint][Variable_Type_Decision]= CDependMatrix(Array_Order_ByRow, ncons, nvars, concoefftype);
```

The line of code below sets the problem type to *OptQCP* (Quadratically Constrained Problem). If you omit the problem type, the Solver SDK will not call your evaluator of type *Eval_Type_Hessian*. Setting this problem type will automatically choose the SOCP engine in the Solver SDK Platform to solve the problem.

```
problem.ProblemType = Problem_Type_OptQCP;
```

Here we create the evaluator *evalHess* of type *Eval_Type_Hessian*. This evaluator will be called to compute the Q matrix for the second constraint.

```
CHessEvaluator evalHess;
prob.Evaluators[Eval_Type_Hessian].Evaluator = & evalHess;
Finally, we ask the Solver to optimize.
Prob.Solver.Optimize();
```

From here, the SDK will call the evaluator of type *Eval_Type_Hessian* found below.

First, we setup a convenient pointer to the Problem, *prob*.

```
CProblem& prob = evaluator.Problem;
```

Then, setup the *qmatval* array for the portfolio return constraint.

```
double qmatval[] =
{ 0.000204, 0.000424, 0.000170, 0.000448, -0.000014,
  0.000424, 0.012329, 0.001785, 0.001633, -0.000539,
  0.000170, 0.001785, 0.000365, 0.000425, -0.000075,
  0.000448, 0.001633, 0.000425, 0.005141, 0.000237,
  -0.000014, -0.000539, -0.000075, 0.000237, 0.000509 };
```

Next, we check the function type of the current function. If the function type is *Function_Type_Constraint*, then we call the *FcnHessValue* method to pass the Hessian matrices for each constraint.

```
if (prob.Solver.FunctionType == Function_Type_Constraint)
```

The next line of code passes the Q-matrix of the first constraint to the *FcnHessValue* method. The left side of this equation calls the *FcnHessValue* method while passing the function type (*Function_Type_Constraint*), the function index (0) and the variable type (*Variable_Type_Decision*). The right side of this equation creates a *DoubleMatrix* of size 5 rows and 5 columns (number of variables x number of variables) with all 0 values. This is because the second order partial derivatives (Hessian matrix) are all zeros for the first constraint ($x_1 + x_2 + x_3 + x_4 + x_5 = 1$).

```
prob.Model.FcnHessValue[Function_Type_Constraint][0][Variable_Type_Decision] = CDoubleMatrix(5, 5, (double) 0);
```

The next line of code passes the Q-matrix of the second constraint to the *FcnHessValue* method. The left side of this equation calls the *FcnHessValue* method while passing the function type (*Function_Type_Constraint*), the function index (1) and the variable type (*Variable_Type_Decision*). The right side of this equation creates a *DoubleMatrix* of size 5 rows and 5 columns (number of variables x number of variables), indicates that the Q-matrix coefficients will be passed by column (*Array_Order_ByCol*) and then passes the *qmatval* array which holds the coefficients.

```
prob.Model.FcnHessValue[Function_Type_Constraint][1][Variable_Type_Decision] =
CDoubleMatrix(Array_Order_ByCol, 5, 5, qmatval);
```

Finally, we ask the SDK to continue solving.

```
return Engine_Action_Continue;
```

Complete Evaluator

Please see below for a complete snapshot of the code required for our evaluator of type *Eval_Type_Hessian*.

```
class HessEvaluator : public IEvaluator
{
```

```

public:
    virtual _Engine_Action Evaluate (CEvaluator& evaluator)
    {
        CProblem& prob = evaluator.Problem;

        double qmatval[] =
            { 0.000204,  0.000424,  0.000170,  0.000448, -0.000014,
              0.000424,  0.012329,  0.001785,  0.001633, -0.000539,
              0.000170,  0.001785,  0.000365,  0.000425, -0.000075,
              0.000448,  0.001633,  0.000425,  0.005141,  0.000237,
              -0.000014, -0.000539, -0.000075,  0.000237,  0.000509 };

        if (prob.Solver.FunctionType == Function_Type_Constraint)
        {
            prob.Model.FcnHessValue[Function_Type_Constraint][0]
                [Variable_Type_Decision] = CDoubleMatrix(5, 5, (double) 0);
            prob.Model.FcnHessValue[Function_Type_Constraint][1]
                [Variable_Type_Decision] = CDoubleMatrix(Array_Order_ByCol, 5,
                    5, qmatval);
        }
        return Engine_Action_Continue;
    }
}

```

For example code in calling an evaluator of type *Eval_Type_Hessian* for a quadratically constrained problem in additional languages, please click the appropriate link: [C#](#), [Procedural API](#), [COM](#).

Solving a Nonlinear Problem with a Hessian Evalautor

If either the KNITRO or LSSQP "plug in" Solver engines are being called to solve a nonlinear problem and an evaluator of *Eval_Type_Hessian* is supplied, then this evaluator will be called in lieu of the evaluator of type *Eval_Type_Function*, to pass the **second** order partial derivatives of the objective function and constraints. In this instance, you must provide the evaluator of type *Eval_Type_Function* and if solving a generic nonlinear model, it is advisable to provide the evaluator of type *Eval_Type_Gradient* as well, but this is optional. The GRG engine included in the Solver SDK Platform is not designed to call this evaluator.

In our example code below, we compute the second order derivatives for the evaluator *evalNLP* of type *Eval_Type_Function* in the section above, *Using a Function Evaluator to Compute the Objective and Constraints*. We must first create the evaluator in the main body of our program which calls the Solver SDK Platform. The name given to the evaluator of type *Eval_Type_Hessian* is *evalHess*.

```

int main()
{
    /* Get data and set up an optimization problem */
    ...

    CProblem prob

    /* Pass the number of variables, constraints, pass constraint upper
    and lower bounds, variable upper and lower bounds, etc. */

```

```
//evaluator of type Eval_Type_Function must be present
CNLPEvaluator evalNLP;
prob.Evaluators[Eval_Type_Function].Evaluator = & evalNLP;
//evaluator of type Eval_Type_Gradient may be present
CJacEvaluator evalJac;
prob.Evaluators[Eval_Type_Gradient].Evaluator = & evalJac;
CJacEvaluator evalHess;
prob.Evaluators[Eval_Type_Hessian].Evaluator = & evalHess;
```

Below is the Hessian evaluator for the example model with the objective = $x^2 + y^2$ and the constraints: $x + y$ and $x * y$. When passing the Hessian, you must first create a double matrix for each function. This example contains two constraints and one objective function, therefore, we have three total functions.

The first double matrix, *objHess*, will hold the 2nd derivatives of the objective function and the second double matrix, *matHess1*, will hold the 2nd derivatives of the second constraint. (The second derivatives for the first constraint ($x + y$) are all 0 so there is no need to pass a double matrix for this constraint.)

First we check the function type of the current function.

```
if (evaluator.Problem.Solver.FunctionCurrent.FunctionType ==
Function_Type_Objective)
```

If the Function Type is *Function_Type_Objective*, then, we create a double matrix, *objHess*, to hold the second order partial derivatives for the objective function, $x^2 + y^2$.

```
{
    CDoubleMatrix objHess =
        evaluator.Problem.Model.FcnHessValue[Function_Type_Objective]
        [0][Variable_Type_Decision];
    objHess.Value[0,0]= 2;
    objHess.Value[0,1] = 0;
    objHess.Value[1,0] = 0;
    objHess.Value[1,1] = 2;
}
```

Again we check the function type, but this time for *Function_Type_Constraint*, then create one (1) double matrix, *matHess1*, to hold the second order partial derivatives for the second constraint, $x * y$. Since the second partial derivatives for the first constraint ($x + y$) are all zero we can pass a double matrix of size 2 x 2 with all elements equal to 0 with one line of code.

The line of code,

evaluator.Problem.Model.FcnHessValue[Function_Type_Constraint][0][Variable_Type_Decision] = CDoubleMatrix(4, 4, (double) 0), creates a double matrix of size 2 by 2 (number of variables x number of variables) and passes all 0 elements to the *FcnHessValue* method.

```
if (evaluator.Problem.Solver.FunctionCurrent.FunctionType ==
Function_Type_Constraint)
```

```
{
    evaluator.Problem.Model.FcnHessValue[Function_Type_Constraint][0][Variable_Type_Decision] = CDoubleMatrix(2, 2, (double) 0);
}
```

```

CDoubleMatrix matHess1 =
evaluator.Problem.Model.FcnHessValue[Function_Type_Constraint]
[1][Variable_Type_Decision];
    matHess1.Value[0,0] = 0;
    matHess1.Value[0,1] = 1;
    matHess1.Value[1,0] = 1;
    matHess1.Value[1,1] = 0;
}

```

Finally, we instruct the Solver SDK Platform to continue solving.

```
return Engine_Action.Continue;
```

Complete Evaluator

Please see below for a complete snapshot of the code required for our evaluator of type *Eval_Type_Hessian*.

```

class HessEvaluator : public IEvaluator
{
public:
    virtual _Engine_Action Evaluate (CEvaluator& evaluator)
    {
        if (evaluator.Problem.Solver.FunctionCurrent.FunctionType ==
            Function_Type_Objective)
        {
            CDoubleMatrix objHess = evaluator.Problem.Model.FcnHessValue
                [Function_Type_Objective][0][Variable_Type_Decision];
            objHess.Value[0,0]= 2;
            objHess.Value[0,1] = 0;
            objHess.Value[1,0] = 0;
            objHess.Value[1,1] = 2;
        }
        if (evaluator.Problem.Solver.FunctionCurrent.FunctionType ==
            Function_Type_Constraint)
        {
            problem.Model.FcnHessValue[Function_Type_Constraint][0]
                [Variable_Type_Decision] =
                CDoubleMatrix(2, 2, (double)0);
            CDoubleMatrix matHess = evaluator.Problem.Model.FcnHessValue
                [Function_Type_Constraint][1][Variable_Type_Decision];
            CDoubleMatrix matHess =
            matHess.Value[0,0] = 0;
            matHess.Value[0,1] = 1;
            matHess.Value[1,0] = 1;
        }
    }
}

```

```

        matHess.Value[1,1] = 0;
    }
    return Engine_Action.Continue;
}

```

This completes the section explaining how to construct an optional Hessian evaluator to compute the Q-matrix for a quadratic objective, a quadratic constraint, or to pass the second order derivatives of the objective and constraints when solving a nonlinear optimization problem. Please click the following links for more example code in [C#](#), the [Procedural API](#) or in [COM](#).

Using an Evaluator to Check the Progress of a Simulation Run

An evaluator can be used to obtain control from the Solver SDK before or after a simulation to check the progress of the solution process, to display a message, etc. For a simulation, there are three such evaluators available of type: *Eval_Type_SimBegin*, *Eval_Type_SimComplete*, and *Eval_Type_Simulation*. Please see the table below for an explanation of each. Simulation Progress Information Evaluators are **always** optional.

Simulation Progress Information Evaluators

<i>Eval_Type_SimBegin</i>	Called at the beginning of a simulation run (of two or more simulations).
<i>Eval_Type_SimComplete</i>	Called at the end of a simulation run (of two or more simulations).
<i>Eval_Type_Simulation</i>	Called after each simulation.

We will use the evaluator of type *Eval_Type_Simulation* in our example below. Note: Each of the evaluators above can be created with the same steps as shown below. Simply substitute *Eval_Type_Simulation* when creating the evaluator with *Eval_Type_X* where *X* is *SimBegin* or *SimComplete*.

First we create the evaluator in the main body of our code. In this example, the name of our evaluator of type *Eval_Type_Simulation* is *EvalSimulation*. (The name of the class is *CSimulation*.) As mentioned above, this type of evaluator is called after every simulation (when multiple simulations are being run) and can be used to obtain control during the solution process in order to display a message, check for a user abort action, etc.

```

int main()
{
    /* Get data and set up a simulation problem */
    ...

    CProblem prob

    /* Pass the number of uncertain variables and pass their
    distributions, pass the number of uncertain constraints, setup
    correlation matrices, etc. */

    CSimulation EvalSimulation Evaluator;
    prob.Evaluators[Eval_Type_Simulation].Evaluator = & EvalSimulation;
}

```

The example code below gives an example of an evaluator of type *Eval_Type_Simulation* which writes the Simulation index to the C++ Console after each simulation is finished. This will allow the User to keep watch as the Solver finishes each simulation.

```
class CSimulation : public IEvaluator
{
    public:
    virtual _Engine_Action Evaluate(CEvaluator& evaluator)
    {
        cout << "Simulation " <<
            evaluator.Problem.Solver.SimulationIndex << endl;
        return Engie_Action_Continue;
    }
}
```

This completes the section explaining how to construct an optional evaluator to gain control during the simulation process. Please click the following links for more example code in [C#](#), the [Procedural API](#) or in [COM](#).

Using an Evaluator to Compute Uncertain Functions in a Simulation Problem

When running a simulation, the evaluator of type *Eval_Type_Sample* is called on each trial to calculate the uncertain functions, much like the evaluator of type *Eval_Type_Function* is called to compute the objective and constraints in a nonlinear optimization problem. If your simulation model contains uncertain functions, then this evaluator is *required*.

To begin we create an evaluator of type *Eval_Type_Sample* in the main body of code. The name of the evaluator is *evalSim*.

```
int main()
{
    /* Get data and set up a simulation problem */
    ...
    CProblem prob
    /* Pass the number of uncertain variables and pass their
       distributions, pass the number of uncertain constraints, setup
       correlation matrices, etc. */
    CSimEvaluator evalSim Evaluator;
    prob.Evaluators[Eval_Type_Sample].Evaluator = & evalSim;
```

The example evaluator of type *Eval_Type_Sample* below is called to compute the uncertain functions, $x + y$ and $x * y$. Again, we start out by creating a problem and three convenient pointers, *prob*, *pVar*, and *pFcn*.

```
CProblem& prob = evaluator.Problem;
const double* pVar = problem.VarUncertain.Value;
double* pFcn = problem.FcnUncertain.Value;
```


The next two lines of code compute the uncertain functions, $x + y$ and $x * y$.

```
pFcn[0] = pVar[0] + pVar[1];    // X+Y
pFcn[1] = pVar[0] * pVar[1];    // X*Y
```

Finally, we tell the Solver SDK Platform to continue simulating. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the uncertain variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the uncertain functions. The SDK will mark the current trial as an error trial and continue with the remaining trials.

```
return Engine_Action_Continue;
```

Complete Evaluator

Please see below for a complete snapshot of the code required for our evaluator of type *Eval_Type_Sample*.

```
class CSimEvaluator : public IEvaluator
{
    public:
    virtual _Engine_Action Evaluator (CEvaluator& evaluator)
    {
        CProblem& prob = evaluator.Problem;
        const double* pVar = problem.VarUncertain.Value;
        double* pFcn = problem.FcnUncertain.Value;
        pFcn[0] = pVar[0] + pVar[1];    // X+Y
        pFcn[1] = pVar[0] * pVar[1];    // X*Y
        return Engine_Action_Continue;
    }
}
```

This completes the section explaining how to construct an evaluator to compute the uncertain functions for a simulation problem. Please click the following links for more example code in [C#](#), the [Procedural API](#) or in [COM](#).

Conclusion

As mentioned in the introduction of this chapter, regardless of what the evaluator does, it is always created using the same methodology. First, the evaluator is created along with specifying the type of evaluator (*Eval_Type_Sample*, *Eval_Type_Function*, etc.) in the main body of the code. Then the evaluator is written to perform the intended function whether it is a Progress Information Evaluator or a Computing Evaluator.

Multiple evaluators can be included in the same problem. For example, a generic nonlinear optimization problem could have up to 7 evaluators, an evaluator of type *Eval_Type_Iteration* to gain control during the solution process, *Eval_Type_Function*, to compute the objective function and constraints, *Eval_Type_Gradient*, to pass the first order derivatives, *Eval_Type_Hessian*, to pass the second order derivatives, *Eval_Type_NewSolution* to gain control each time a new solution is found,

Eval_Type_OptBegin to gain control before the optimization is run, and Eval_Type_Optimization to gain control after the optimization has finished.

Calling the Solver SDK from C++

C++ Compilers

To use the Solver SDK from C++ you must have an appropriate compiler and linker capable of building Windows applications, such as Microsoft Visual C++, Microsoft Visual Studio C++, Borland C++, or Symantec C++. This Guide will give explicit instructions for use with Microsoft Visual Studio C++ , but use with other compilers should be similar.

To simplify the input/output and focus on the Solver SDK routines and arguments, the example applications in this chapter use *console* calls for output and will be built as a "Win32 console application" in Microsoft Visual Studio C++.

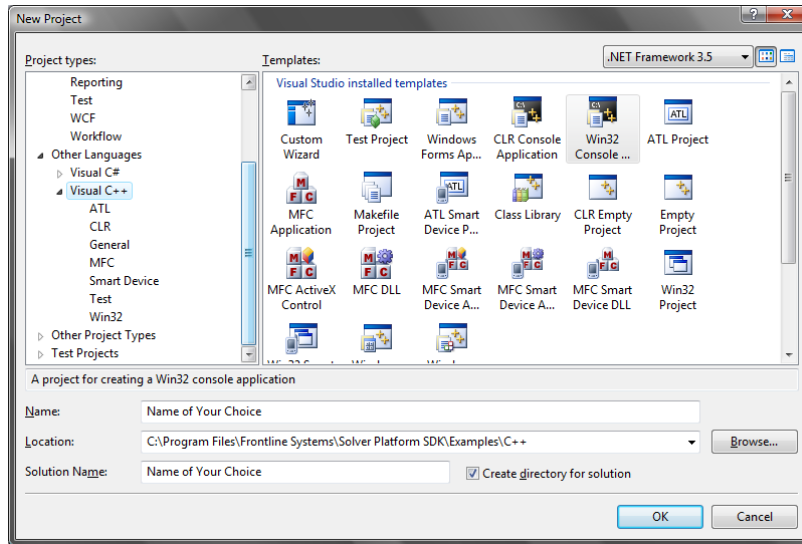
Basic Steps: Solving Three Simple Models

This chapter will outline the steps required to compile, link and run a simple linear, nonlinear and simulation example using the Solver SDK Microsoft Visual Studio C++. The steps required for other compilers will be similar.

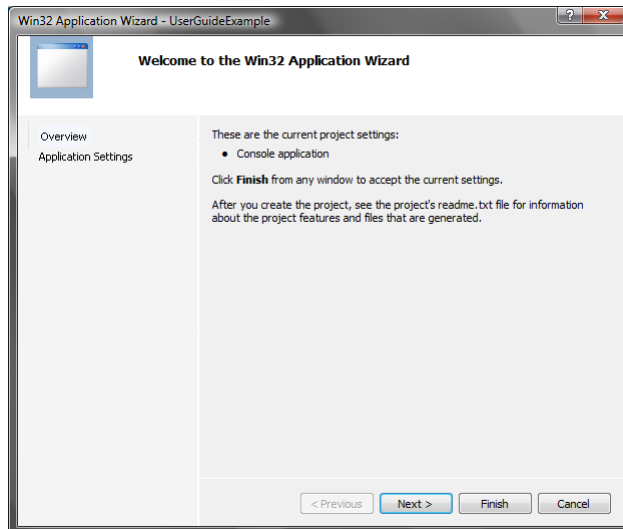
Creating a New C++ Project

First, follow the steps in the "Installation" chapter, which will install all required SDK files, including Example files, into the **Frontline Systems** directory (located at C:\Program Files) on your hard disk.

1. **Create a new project.** Start Microsoft Visual Studio, Select File | New Project, select C++ on the left of the dialog and select Win32 Console Application under Templates (on the right of the dialog). Type a name of your choice in the Project name edit box, and **C:\Program Files\Frontline Systems\Analytic Solver SDK\Examples\C++** (or a path of your choosing) in the Location edit box. Then click OK.



Then Click Finish.



2. **Add the library file, SolverSDK.lib.** To create an application that calls the Solver SDK in C++, you must include the import library file, SolverSDK.lib in your project or your linker's response file. (Alternatively, you can write code to use "run-time dynamic linking" by calling Windows API routines such as *LoadLibrary()*.)

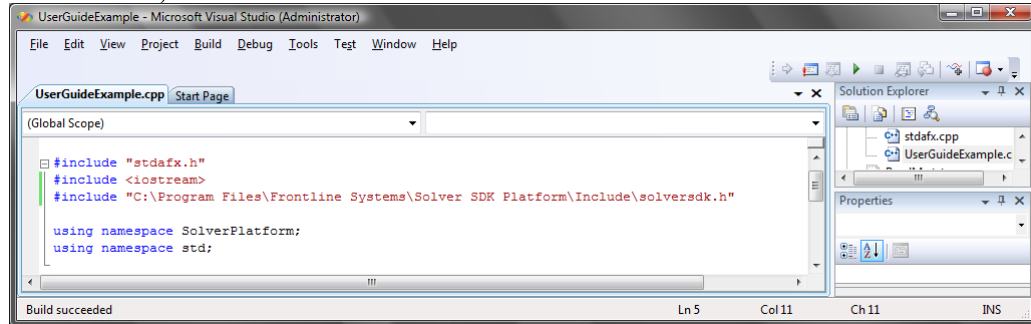
Right click Source Files in the Solution Explorer and select Add | Add Existing Item in the dialog, navigate to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Lib, and add the file, SolverSDK.lib.

3. **Add an include statement and an (optional) directive.** In order to access the Solver SDK functions, you must add an *include* statement for the file *solversdk.h*. In addition, you can also add an optional *using namespace* directive for *SolverPlatform*. If you choose not to include the *using namespace* directive, you must add "SolverPlatform::" to the beginning of each SDK function call. For example:

```
SolverPlatform::CProblem problem;
```

```
problem.Model.AllLinear[SolverPlatform::Function_Type_Objective]
= SolverPlatform::CDoubleMatrix(Array_Order_ByRow, 1, nvars,
objcoeff);
```

This example uses the *using SolverPlatform* directive which is shown in the screenshot below. (The precompiled header files such as *stdafx.h* and *iostream* provide basic operating and output functions.)



From here you can solve either a linear, nonlinear or simulation model. For more details on each, please read on.

Creating a Linear Model

1. **First Steps.** Follow the steps in the section, *Creating a New C++ Project*. In this example, we will name the function that will hold the linear example, *linear()*.
2. **Create a linear model.** For this example, we will solve a linear programming model with two variables (*x1* and *x2*) and three constraints. Both final variable values must be 0 or larger. The example model is below.

```
Maximize 2 x1 + 3 x2
Subject To:
9 x1 + 6 x2 <= 54
6 x1 + 7 x2 <= 42
5 x1 + 10 x2 <= 50
x1, x2 non-negative
```

To begin, the example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model.

```
int nvars = 2, ncons = 3;
```

Next, we create an array to hold the upper bounds of the constraint right hand sides.

```
double ub_constraints[] = { 54, 42, 50};
```

This next line of code (*CProblem prob*) creates an instance of the *CProblem* class. There are two different constructors which create a *Problem*. The first constructor (*CProblem prob*) will only create the *Problem*. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or a simulation problem. If this constructor is used, a user must either call *prob.VarDecision.Size* to automatically add a variable block or create a variable block using *CVariable vars(Variable_Type_Decision, L"vars", 1)* where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add constraints automatically using *prob.FcnConstraint.Size* or create a function with *CFunction*

`con1(Function_Type_Constraint, L"con1", 1)` (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling `prob.Functions.Add(con1)` to add the function to the Problem.

The second constructor, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type_Minimize*, *Solver_Type_Maximize*, or *Solver_Type_Simulate*) the number of variables (*nvars*) and the number of constraints (*ncons*). An example of this constructor would be: `CProblem problem(Solver_Type_Minimize, nvars, ncons)`.

```
CProblem prob;
```

The next line of code creates one variable block of size *nvars* (two).

```
prob.VarDecision.Size = nvars;
```

while the next line of code creates one function block of size *ncons* (two).

```
prob.FncConstraint.Size = ncons;
```

As mentioned above, we could have added the variables using the two lines of code below. The first line of code creates a variable block of type *Variable_Type_Decision*, names the block "vars" and then sets the size of the block to 2. The second line of code adds the *vars* variable block to the Problem.

```
CVariable vars (Variable_Type_Decision, L"vars",2);
```

```
prob.Variables.Add(vars);
```

In addition, we could have added the constraints using the two lines of code below. The first line of code creates a function block of type *Variable_Type_Constraint*, names the block "cons" and then sets the size of the block to 3. The second line of code adds the *cons* function block to the Problem.

```
CFunction cons (Function_Type_Constraint, L"cons",3);
```

```
prob.Functions.Add(cons);
```

To tell the SDK to maximize the model, we use the line of code below. Alternatively, *Solver_Type_Minimize* will minimize the problem.

```
prob.Solver.SolverType = Solver_Type_Maximize;
```

We pass the *ub_constraints* array which holds the upper bounds of the constraint right hand sides to the SDK.

```
prob.FcnConstraint.UpperBound = ub_constraints;
```

To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
prob.VarDecision.NonNegative();
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass upper and lower variable bounds to the SDK.

The next two lines of code create two different *DoubleMatrix* objects to hold the objective and constraint coefficients: *objcoeff* - for the objective coefficients, and *concoeff* - for the constraint coefficients. Since there is only one objective, we create the *objcoeff* array of size *nvars* (number of variables) x 1 for the objective coefficients and the second array, *concoeff*, of size *nvars* x *ncons* (number of variables) for the constraint coefficients. Notice that we are using *NZBgn()* and *NZEnd()* when passing the constraint matrix (*concoeff*) as explained in the subsection *Passing Sparse Matrices* in the chapter *Designing Your Application*. As a result, the Solver SDK will not update the *concoeff* *DoubleMatrix* object until *NZEnd()* is executed.

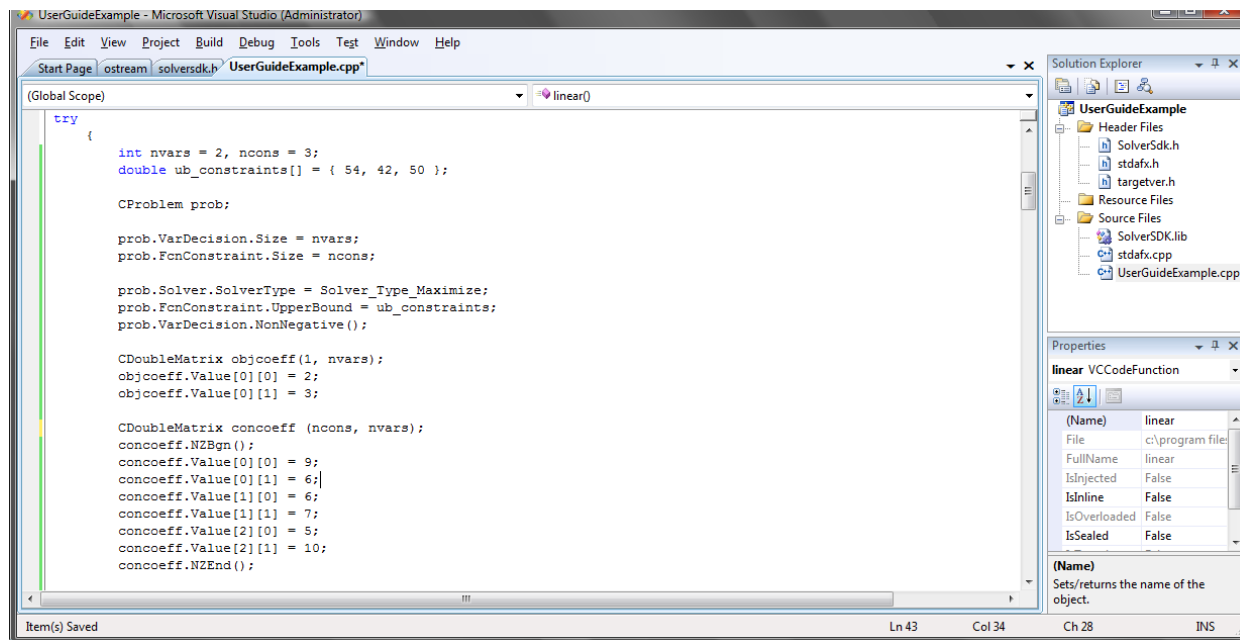
```
CDoubleMatrix objcoeff(1, nvars);
```

```

objcoeff.Value[0][0] = 2;
objcoeff.Value[0][1] = 3;

CDoubleMatrix concoeff (ncons, nvars);
concoeff.NZBgn();
concoeff.Value[0][0] = 9;
concoeff.Value[0][1] = 6;
concoeff.Value[1][0] = 6;
concoeff.Value[1][1] = 7;
concoeff.Value[2][0] = 5;
concoeff.Value[2][1] = 10;
concoeff.NZEnd();

```



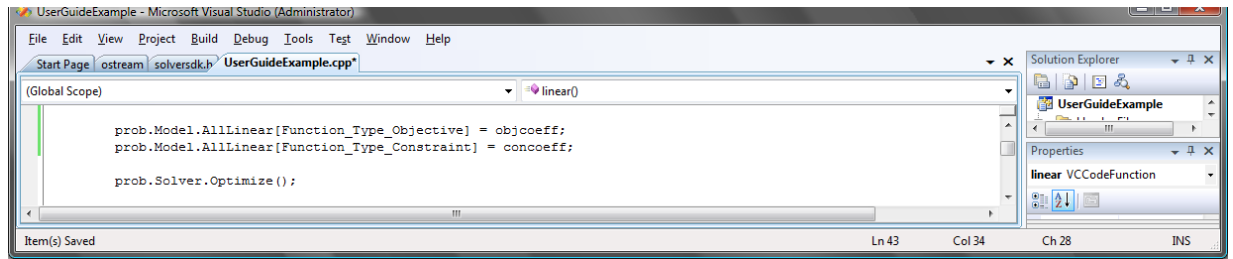
3. **Call AllLinear functions and solve the linear model.** The example code below calls *prob.Model.AllLinear[Function_Type_Objective]* to pass the objective coefficient matrix, *objcoeff*, and *prob.Model.AllLinear[Function_Type_Constraint]* to pass the constraint coefficient matrix, *concoeff*. Calling *prob.Model.AllLinear* sets the problem type to "linear" (Problem_Type_OptLP). Consequently, the SDK will automatically choose the best linear engine available to solve the problem. If no additional "plug in" engines have been added, then the LP/Quadratic engine will be selected in the Solver SDK Platform and the LP Simplex engine will be selected in the Solver Platform Pro.

```

prob.Model.AllLinear[Function_Type_Objective] = objcoeff;
prob.Model.AllLinear[Function_Type_Constraint] = concoeff;

Solve the optimization problem by calling:
prob.Solver.Optimize();

```



4. **Obtain and display the solution to the linear model.** We obtain the solution status along with the final objective and decision variable values,

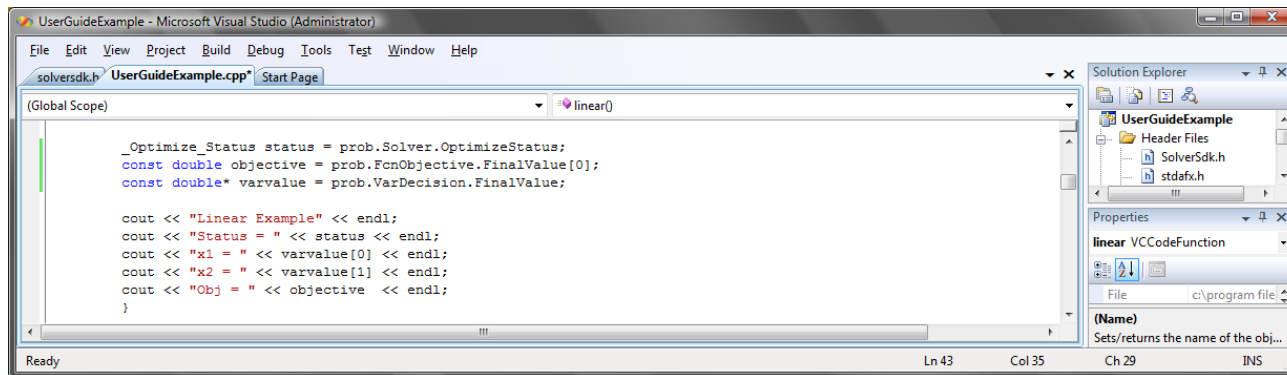
```

_Optimize_Status status = prob.Solver.OptimizeStatus;
Const double objective = prob.FcnObjective.FinalValue[0];
Const double* varvalue = prob.VarDecision.FinalValue;

and display them on the C++ console.

cout << "Status = " << status << endl;
cout << "x1 = " << varvalue[0] << endl;
cout << "x2 = " << varvalue[1] << endl;
cout << "Obj = " << prob.FcnObjective.FinalValue[0]

```

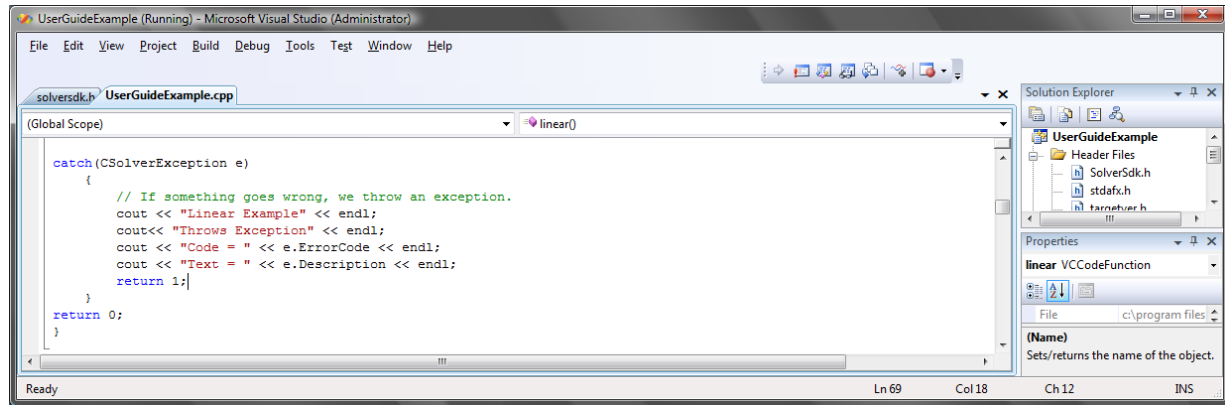


5. **Catch any errors.** The catch statement handles any exceptions.

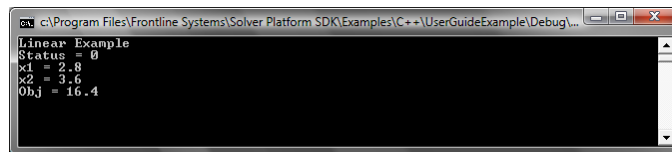
```

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

```

6. **Build and run the application.** To compile and link this project, select Build | Build *userguideexample*. Then use Debug | Start Debugging to run the program. An output window like the one below should appear.



Creating a Nonlinear Problem

1. **Basic Steps.** Follow the steps above in the section, *Creating a New C++ Project*. In this example, we will name the function that will hold the nonlinear example, *nonlinear()*.
2. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constraint ($x + y$) forces the total of the two variables to be equal to 1 and the second constraint ($x * y$) forces the product of the two variables to be zero or larger. Both final variable values must be non-negative (0 or larger).

```
Minimize  $x^2 + y^2$ 
Subject To:
 $x + y = 1$ 
 $x * y \geq 0$ 
 $x, y$  non-negative
```

To begin, we initialize the number of variables (*nvars*) and constraints (*ncons*).

```
int nvars = 2, ncons = 2;
```

Next, we create two arrays of size *ncons* (2 constraints) to hold the upper and lower bounds on the constraint right hand sides. Notice the upper and lower bound of the first constraint is 1 to denote an equality constraint. *CConstants::PINF* sets the upper bound of the second constraint to positive infinity.

```
double ub_constraints[2] = { 1.0, CConstants::PINF };
double lb_constraints[2] = { 1.0, 0 };
```

The next two lines of code create two arrays to hold the lower and upper variable bounds.

```
double lb[2] = { 0, 0 };
double ub[2] = { 1, 1 };
```

This next line of code (*CProblem prob (Solver_Type_Minimize, nvars, ncons)*) creates an instance of the *CProblem* class. There are two different constructors which create a *Problem*. The first constructor (*CProblem prob*) will only create the problem. This constructor will not create any

variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call *prob.VarDecision.Size* to automatically add a variable block or create a variable block using *CVariable vars(Variable_Type_Decision, L"vars", 1)* where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add constraints automatically using *prob.FcnConstraint.Size* or create a function with *CFunction con1(Function_Type_Constraint, L"con1", 1)* (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling *prob.Functions.Add(con1)* to add the function to the Problem.

The second constructor, which is used here, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type_Maximize*, *Solver_Type_Minimize*, or *Solver_Type_Simulate*) the number of variables (*nvars*) and the number of constraints (*ncons*).

```
CProblem prob(Solver_Type_Minimize, nvars, ncons);
```

Here we pass the constraint upper (*ub_constraints*) and lower (*lb_constraints*) bounds to the SDK,

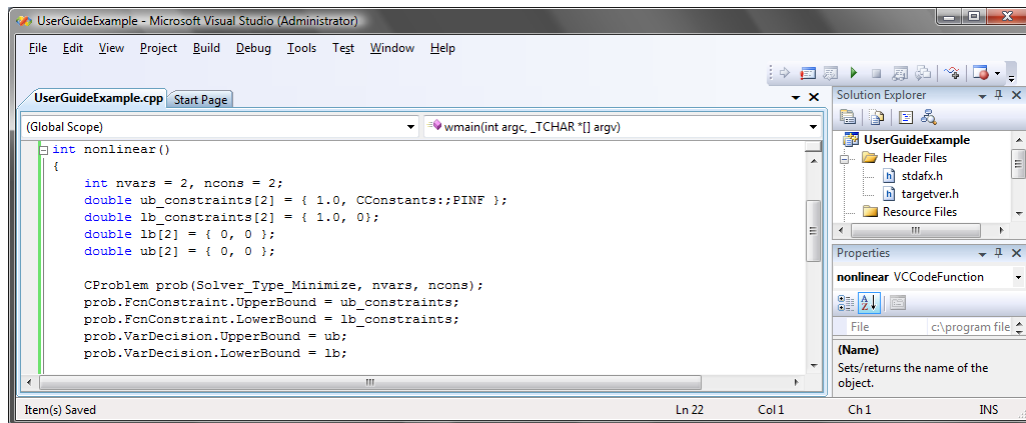
```
prob.FcnConstraint.UpperBound = ub_constraints;
```

```
prob.FcnConstraint.LowerBound = lb_constraints;
```

following up with the variable upper (*ub*) and lower (*lb*) bounds.

```
prob.VarDecision.UpperBound = ub;
```

```
prob.VarDecision.LowerBound = lb;
```

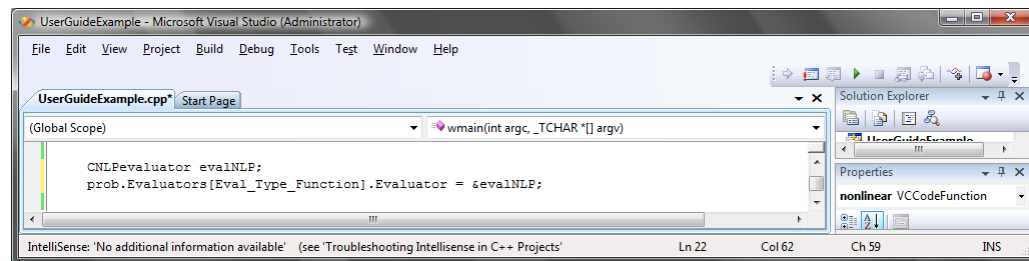


3. **Create the nonlinear evaluator.** In a linear model or quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied. (It is possible to pass these derivatives directly to the SDK using the *Eval_Type_Gradient* evaluator. For more information on supplying this type of evaluator, see the subheading *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter, *Solver SDK Evaluators*.)

We create and pass the evaluator of type *Eval_Type_Function* (as described above) to the SDK. The name of this evaluator is *evalNLP*.

```
CNLP evaluator evalNLP;
```

```
prob.Evaluators[Eval_Type_Function].Evaluator = &evalNLP;
```



4. **Write the nonlinear evaluator.** Inside of the evaluator *evalNLP*, we first create a convenient pointer, *p*, to the Problem,

```
CProblem& p = evaluator.Problem;
```

and another pointer, *pVar*, to the variables.

```
const double* pVar=p.VarDecision.Value;
```

Next, we obtain a pointer to the constraint functions and calculate the constraints.

```
double* pFcn=p.FcnConstraint.Value;
```

Here, we pass the first constraint, $x + y$,

```
pFcn[0] = pVar[0] + pVar[1];
```

and now the second constraint, $x * y$.

```
pFcn[1] = pVar[0] * pVar[1];
```

Then we compute the objective function, $x * x + y * y$.

```
p.FcnObjective.Value[p.ObjectiveIndex] = pVar[0] * pVar[0] +  
pVar[1] * pVar[1];
```

Finally, we tell the SDK to continue optimizing by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the functions (objective or constraints). The Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine (for solving nonsmooth problems) will disregard the current point and continue solving.

```
return Engine_Action_Continue;
```

An alternative way to write this evaluator without using the *p*, *pVar*, and *pFcn* pointers is below.

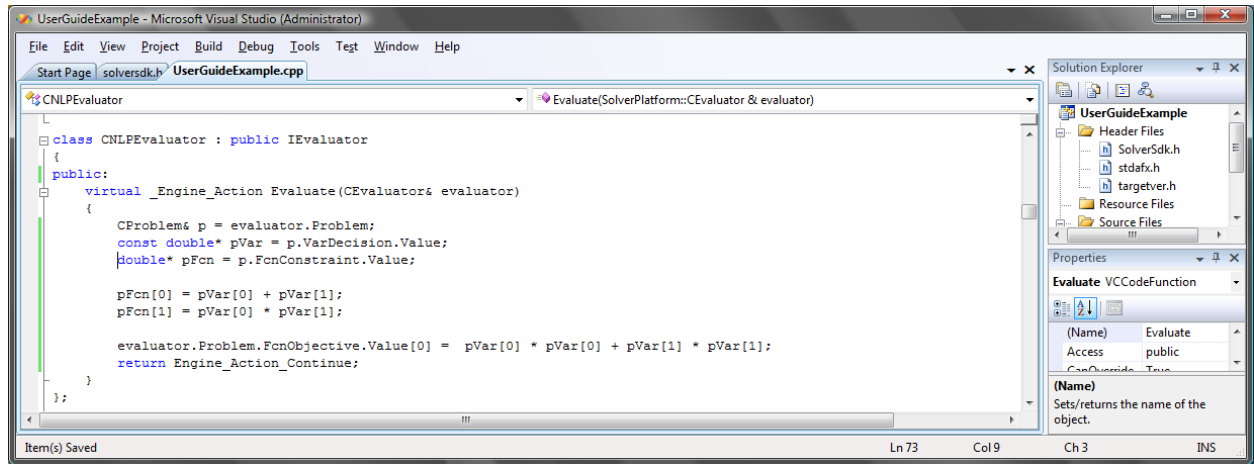
```
class CNLPEvaluator : public IEvaluator  
{  
public:  
    virtual _Engine_Action Evaluate(CEvaluator& evaluator)  
    {  
        evaluator.Problem.FcnConstraint.Value[0] =  
            evaluator.Problem.VarDecision.Value[0] +  
            evaluator.Problem.VarDecision.Value[1];  
        evaluator.Problem.FcnConstraint.Value[1] =  
            evaluator.Problem.VarDecision.Value[0] *  
            evaluator.Problem.VarDecision.Value[1];  
        evaluator.Problem.FcnObjective.Value[0] = pVar[0] *  
            pVar[0] + pVar[1] * pVar[1];  
        return Engine_Action_Continue;  
    }  
};
```

```

    }
};

```

Our evaluator is complete. We will now finish the code inside of the *nonlinear()* function.

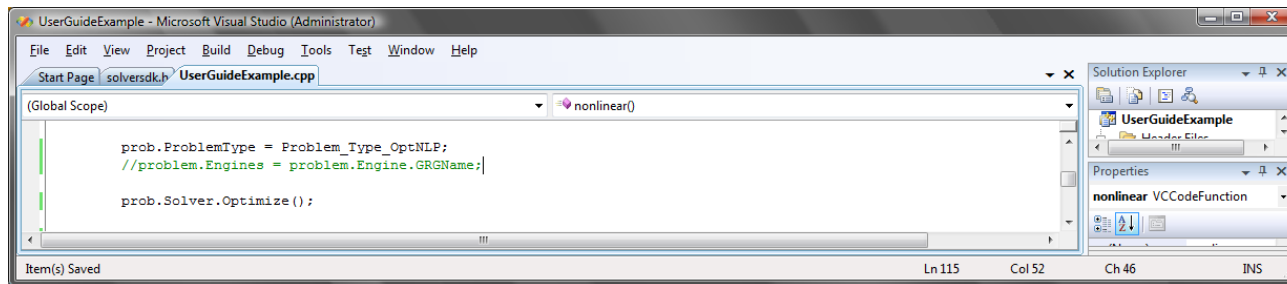


- Solve the nonlinear problem.** Back inside the *nonlinear()* function, we provide the problem type as nonlinear which directs the SDK to select the GRG engine to solve the model. Alternatively, we also could have set the GRG engine manually (in either the SDK Platform or SDK Pro) by calling: *prob.Engine = prob.Engines.[Engines.GRGName];*

```
prob.ProblemType = Problem_Type_OptNLP;
```

To solve the nonlinear optimization problem, we call:

```
prob.Solver.Optimize();
```



- Obtain the solution status and final answer.** We obtain the solution status, along with the final objective and variable values,

```
_Optimize_Status status = prob.Solver.OptimizeStatus;
```

```
const double* objvalue = prob.FcnObjective.FinalValue;
```

```
const double* varvalue = prob.VarDecision.FinalValue;
```

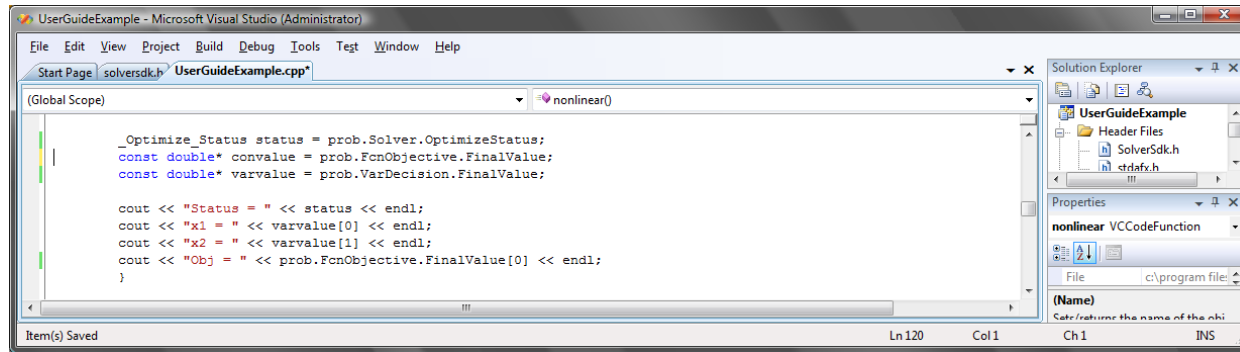
and display them on the C++ console.

```
cout << "Status = " >> status << endl;
```

```
cout << "x1 = " >> varvalue[0] << endl;
```

```
cout << "x2 = " >> varvalue[1] << endl;
```

```
cout << "Obj = " >> prob.FcnObjective.FinalValue[0]
```



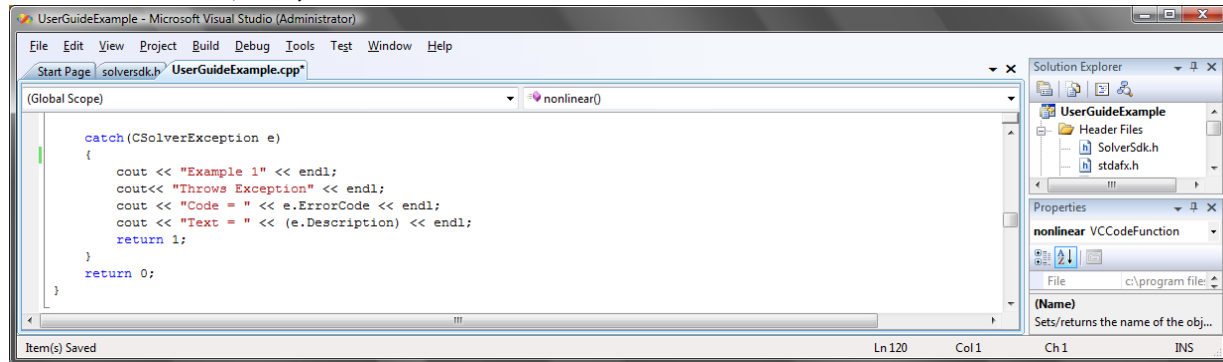
7. Catch any errors. The catch statement will handle any exceptions.

```

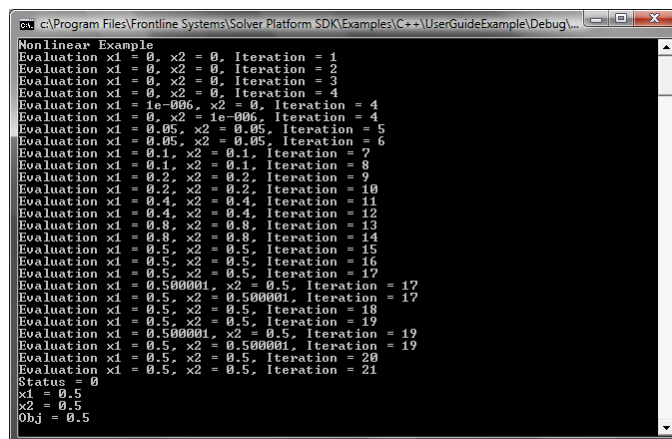
Catch (CSolverException e)
{
    cout << "Example 1" << endl;
    cout << "Throws Exception " << endl;
    cout << "Code = " << e.ErrorCode << endl;
    cout << "Text = " << e.Description << endl;

    return 1; }

```



7. Build and run the application. To compile and link this project, select Build | Build *userguideexample*. Then use Debug | Start Debugging to run the program. An output window like the one below should appear.



Solving a Simulation Problem

1. **Basic Steps.** Follow the steps in the chapter, *Creating a New C++ Project*. In this example, we will name the function that will hold the simulation example, *simulation()*.
2. **Create a simulation problem.** The simulation model that we will be solving in this example contains two uncertain variables, *x* and *y*. The uncertain variable *x* has a uniform distribution with a lower parameter of 0 and an upper parameter of 2. The uncertain variable *y* has a Normal distribution with a mean and standard deviation equal to 1. In addition, there are two uncertain functions. The first uncertain function is the total of the two uncertain variables. The second is the product of the two uncertain variables.

Given two uncertain variables *X* and *Y* with distributions:

```
x = PsiUniform(0,2)   and   y = PsiNormal(1,1)
```

```
Function[1]:  x + y
```

```
Function[2]:  x * y
```

To begin, we initialize the number of uncertain variables (*nvars*) and functions (*nfcns*).

```
int nvars = 2, nfcns = 2;
```

This example uses a correlation matrix which measures the degree to which the movements of the two uncertain variables are related. This is optional. The correlation matrix is:

1	.5
.5	1

The correlation coefficients in the *correl* array are being passed row-wise.

```
double correl[] = { 1.0, 0.5, 0.5, 1.0 };
```

This next line of code (*CProblem prob (Solver_Type_Simulate, nvars, nfcns)*) creates an instance of the *CProblem* class. There are two different constructors which create a *Problem*. The first constructor (*CProblem prob*) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving an optimization or simulation problem. If this constructor is used, a user must either call *prob.VarUncertain.Size* to automatically add a variable block or create a variable block using *CVariable vars (Variable_Type_Uncertain, L"vars", 1)* where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add uncertain functions automatically using *prob.FcnUncertain.Size* or create a function with *CFunction con1 (Function_Type_Uncertain, L"con1", 1)* (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling *prob.Functions.Add(con1)* to add the function to the Problem.

The second constructor, which is used here, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type_Minimize*, *Solver_Type_Maximize*, or *Solver_Type_Simulate*) the number of variables (*nvars*) and the number of functions (*nfcns*).

```
CProblem prob(Solver_Type_Simulate, nvars, nfcns);
```

Next, we pass the distributions of the two uncertain variables, Uniform and Normal.

```
prob.Model.VarDistribution[0] = Distribution(L"PsiUniform(0,2)");
```

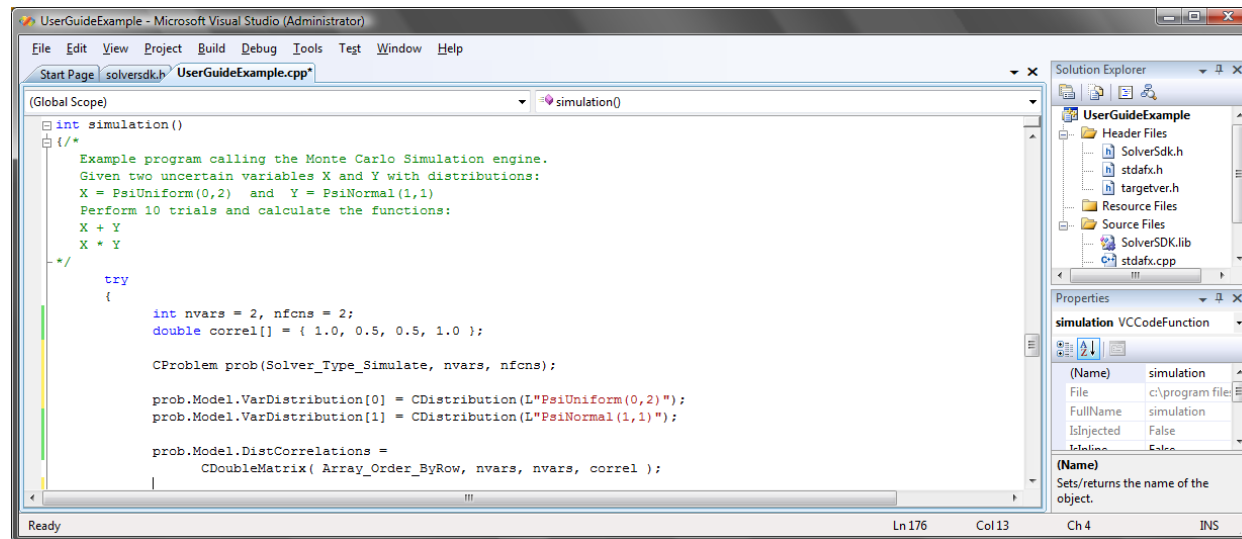
```
prob.Model.VarDistribution[1] = CDistribution(L"PsiNormal(1,1)");
```

Here we create a new double matrix of size *nvars* x *nvars* (2 x 2) to hold the correlation matrix. We are passing the *correl* array row-wise.

```

prob.Model.DistCorrelations =
CDoubleMatrix( Array_Order_ByRow, nvars, nvars, correl );

```



3. **Create the simulation evaluator.** When running a simulation, the evaluator *Eval_Type_Sample* is called so the user can calculate the uncertain functions, much like *Eval_Type_Function* is called to compute the objective function and constraints in a nonlinear optimization problem.

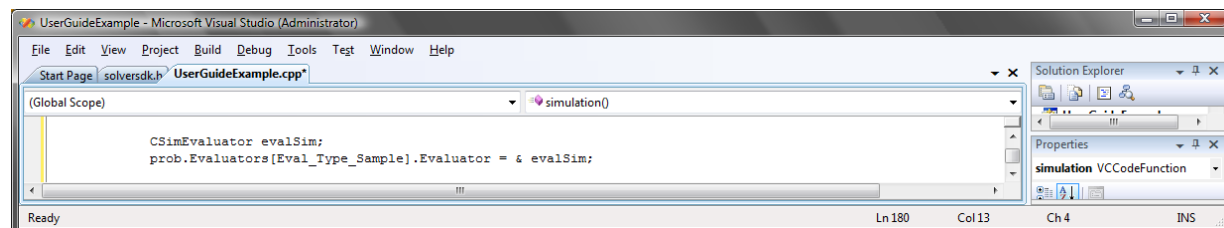
We create the *Eval_Type_Sample* evaluator (as described above) and pass the evaluator to the SDK. The name of the evaluator is *evalSim*.

```

CSimEvaluator evalSim;

problem.Evaluators[Eval_Type_Sample].Evaluator = &evalSim;

```



4. **Write the simulation evaluator.** Inside of the evaluator *evalSim*, we first create a convenient pointer, *p*, to the Problem,

```

CProblem& p = evaluator.Problem;

```

and a second pointer, *pVar*, to the uncertain variable values.

```

const double* pVar = p.VarUncertain.Value;

```

Next, we create a pointer, *pFcn*, to the uncertain functions.

```

double* pFcn=p.FcnUncertain.Value;

```

We compute the uncertain function values given the current variable values.

```

pFcn[0] = pVar[0] + pVar[1];

```

```

pFcn[1] = pVar[0] * pVar[1];

```

Finally, we tell the SDK to continue simulating by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop*

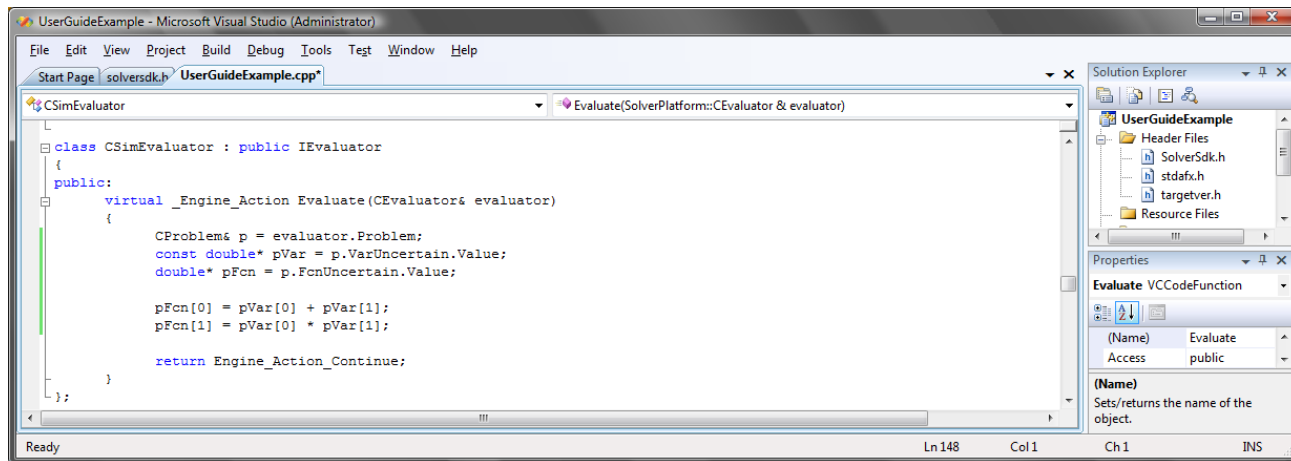
will indicate to the SDK that an error has occurred during the calculation of the uncertain functions. The SDK will mark the current trial as an error trial and continue with the remaining trials.

```
return Engine_Action_Continue;
```

An alternative way to write this evaluator without using the *p*, *pVar*, and *pFcn* pointers is below.

```
class CSimEvaluator : public IEvaluator
{
public:
    virtual _Engine_Action Evaluate(CEvaluator& evaluator)
    {
        evaluator.Problem.FcnUncertain.Value[0] =
            evaluator.Problem.VarUncertain.Value[0] +
            evaluator.Problem.VarUncertain.Value[1];
        evaluator.Problem.FcnUncertain.Value[1] =
            evaluator.Problem.VarUncertain.Value[0] *
            evaluator.Problem.VarUncertain.Value[1];

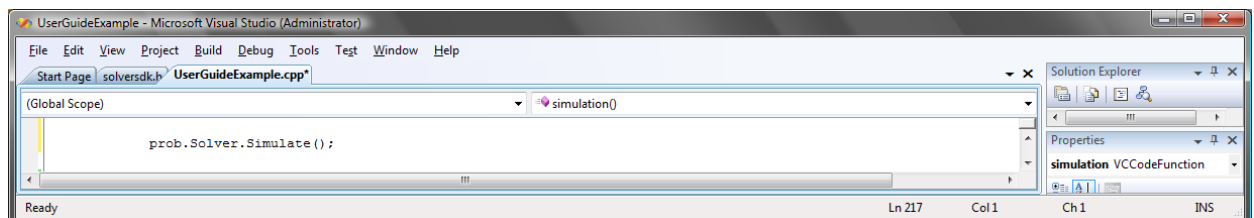
        return Engine_Action_Continue;
    }
};
```



Our evaluator is complete. We will now finish the code inside of the *simulation()* function.

5. **Solve the simulation model.** Back inside the *simulation()* function, we call the following line of code to run the simulation.

```
prob.Solver.Simulate();
```



6. **Obtain and display various simulation statistics.** We obtain the simulation status.

```
_Simulate_Status status = prob.Solver.SimulateStatus;
cout << "Simulation Example" << endl;
cout << "Status = " << status << endl;
```



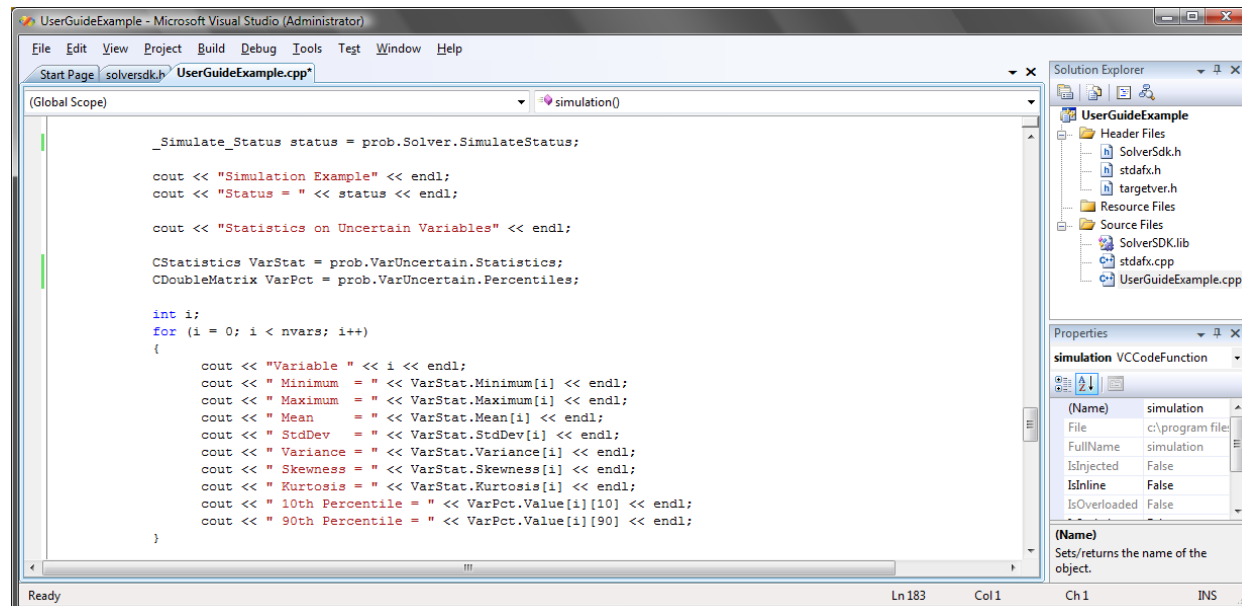
```
cout << "Statistics on Uncertain Variables" << endl;
```

The next two lines of code create convenient pointers, *VarStat* for the uncertain variable statistics and *VarPct* for the uncertain variable percentiles.

```
CStatistics VarStat = prob.VarUncertain.Statistics;
CDoubleMatrix VarPct = prob.VarUncertain.Percentiles;
```

Display statistics for the two uncertain variables.

```
int i;
for (i = 0; i < nvars; i++)
{
    cout << "Variable " << i << endl;
    cout << " Minimum  = " << VarStat.Minimum[i] << endl;
    cout << " Maximum  = " << VarStat.Maximum[i] << endl;
    cout << " Mean      = " << VarStat.Mean[i] << endl;
    cout << " StdDev   = " << VarStat.StdDev[i] << endl;
    cout << " Variance = " << VarStat.Variance[i] << endl;
    cout << " Skewness = " << VarStat.Skewness[i] << endl;
    cout << " Kurtosis = " << VarStat.Kurtosis[i] << endl;
    cout << " 10th Percentile = " << VarPct.Value[i][10] << endl;
    cout << " 90th Percentile = " << VarPct.Value[i][90] << endl;
}
```

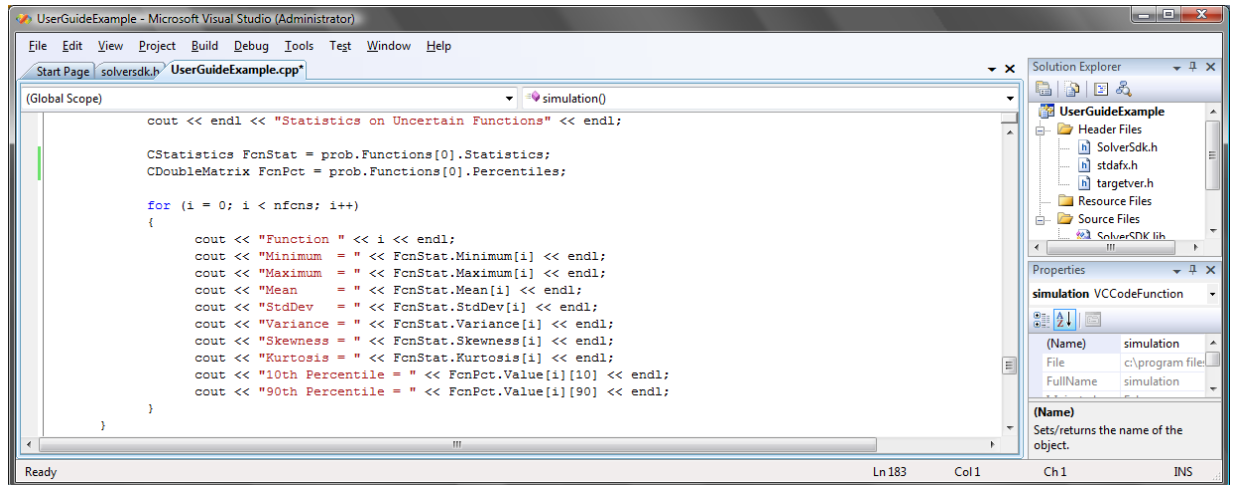


Here we create two more pointers, *FcnStat* and *FcnPct*. *FcnStat* is a pointer to the uncertain function statistics and *FcnPct* is a pointer to the uncertain function percentiles.

```
cout << endl << "Statistics on Uncertain Functions" << endl;
CStatistics FcnStat = prob.Functions[0].Statistics;
CDoubleMatrix FcnPct = prob.Functions[0].Percentiles;
```

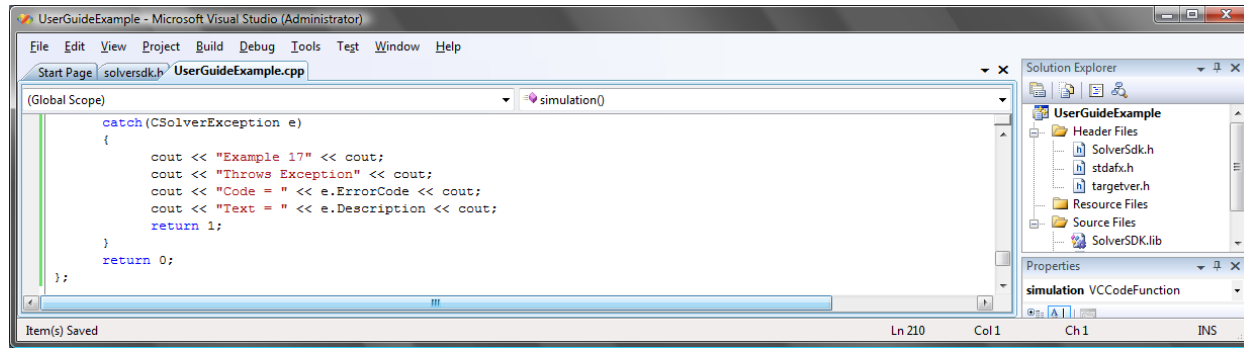
Display statistics for the uncertain functions.

```
for (i = 0; i < nfcns; i++)
{
    cout << "Function " << i << endl;
    cout << "Minimum  = " << FcnStat.Minimum[i] << endl;
    cout << "Maximum  = " << FcnStat.Maximum[i] << endl;
    cout << "Mean      = " << FcnStat.Mean[i] << endl;
    cout << "StdDev   = " << FcnStat.StdDev[i] << endl;
    cout << "Variance = " << FcnStat.Variance[i] << endl;
    cout << "Skewness = " << FcnStat.Skewness[i] << endl;
    cout << "Kurtosis = " << FcnStat.Kurtosis[i] << endl;
    cout << "10th Percentile = " << FcnPct.Value[i][10] << endl;
    cout << "90th Percentile = " << FcnPct.Value[i][90] << endl;
}
```



7. **Catch any errors.** The catch statement will handle any exceptions.

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



8. **Build and run the application.** To compile and link this project, select Build | Build *userguideexample*. Then use Debug | Start Debugging to run the program. An output window like the one below should appear.

```

c:\Program Files\Frontline Systems\Solver Platform SDK\Examples\C++\UserGuideExample\Debug...
Simulation Example
Status = 0
Statistics on Uncertain Variables
Variable 0
Minimum = 0.000234428
Maximum = 1.99322
Mean = 1.00004
StdDev = 0.57764
Variance = 0.333668
Skewness = -0.000129764
Kurtosis = -1.20361
10th Percentile = 0.200811
90th Percentile = 1.79961
Variable 1
Minimum = -2.45391
Maximum = 4.18373
Mean = 0.999783
StdDev = 1.00008
Variance = 1.00016
Skewness = -0.00759284
Kurtosis = -0.0262772
10th Percentile = -0.27724
90th Percentile = 2.28047
Statistics on Uncertain Functions
Function 0
Minimum = -2.31234
Maximum = 5.89406
Mean = 1.99982
StdDev = 1.38082
Variance = 1.90444
Skewness = -0.0224909
Kurtosis = -0.324248
10th Percentile = 0.198875
90th Percentile = 3.77204
Function 1
Minimum = -1.39843
Maximum = 7.72154
Mean = 1.28484
StdDev = 1.48674
Variance = 2.2104
Skewness = 1.20642
Kurtosis = 1.18194
10th Percentile = -0.0776312
90th Percentile = 3.42668

```

C/C++ Source Code

The source code file StandardExamples.cpp (installed into C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\C++\Standard Examples) contains over thirty examples of how to solve linear, integer, quadratic, nonlinear, nonsmooth, Monte Carlo simulation and simulation optimization programming problems. The list below gives a brief explanation of each example problem.

You can easily open and run each example by opening Visual Studio, clicking File | Open Project, browsing to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\C#\Standard Examples and opening the file, StandardExamples-VS2003.sln (for Visual Studio 2003), StandardExamples-VS2005.sln (for Visual Studio 2005), StandardExamples-VS2008.sln (for Visual Studio 2008) or StandardExamples-VS2010.sln (for Visual Studio 2010). To run the example, go to Debug | Start Debugging in Visual Studio.

1. Linear Programming Example (Dense Matrix)
 - Simple two-variable LP problem which passes a dense constraint matrix "row-wise".
2. Integer Linear Programming Example (Dense Matrix)

- Simple MIP problem, identical to the previous LP problem except that the decision variables are required to have integer values and the coefficients are passed column wise.
3. Markowitz Optimization with QP Solver
 - Quadratic programming problem, using the classic Markowitz method to find an efficient portfolio of five securities.
 4. Nonlinear programming with GRG engine
 - Simple two-variable nonlinear optimization problem, which defines an evaluator of type *Eval_Type_Function* to compute the problem functions.
 5. Non-Linear Programming with Evolutionary engine
 - Finds the global optimum of a classic two-variable problem, the Branin function, which has three local optima.
 6. Non-Linear Programming with the GRG engine using Gradients
 - Identical to example 5, except that an evaluator of type *Eval_Type_Gradient* is also defined, to give the SDK faster and more accurate partial derivatives of the problem functions.
 7. Markowitz Optimization with SOCP Solver
 - Identical to example 3, except that the *AllGradDepend* method is used to pass *objtype* and *matvaltype* arrays to the SOCP engine.
 8. Quadratically Constrained Programming with SOCP Solver
 - Similar to examples 3 and 7, except that the objective maximizes the return.
 9. Exception Handling Example
 - Illustrates how to "catch" an exception.
 10. Reading an MPS, OSiL or Excel File
 - Example shows how a problem can be loaded from a text file.
 11. Markowitz Optimization using Hessians
 - Identical to examples 3 and 6, except that an evaluator of type *Eval_Type_Hessian* is defined to give the SDK faster and more accurate second order derivatives of the problem functions.
 12. Second order Cone programming Example
 - Simple example of a problem with two cone constraints.
 13. Integer Programming with All-Different Constraint
 - Similar to example 1 and 2 with an added All-Different Constraint.
 14. Linear Programming Infeasibility Diagnosis Example
 - Attempts to solve a linear problem that is infeasible, then it uses the IIS finder to diagnose the infeasibility.
 15. Illustrating Multiple Solutions
 - Example illustrates how a user can access all incumbent solutions that the SDK finds during the optimization.
 16. Linear Programming with Sparse Matrix
 - Example illustrates how to pass a sparse matrix.
 17. Simulation Example
 - Example program calling the Monte Carlo simulation engine.

18. Simulation Example with Distribution Fitting
 - Illustrates how to fit a distribution to a dataset.
19. Creating a Problem Object
 - Example illustrates two different methods of creating the *CProblem* object.
20. Adding Variable and Function Blocks
 - This example illustrates the use of pointers to create a *CProblem* object.
21. Creating a vector of Problem Objects
 - Creates a vector of problem objects, each with 2 vars and 4 fens.
Note: This example only will work if multiple licences for the Solver SDK Platform exist.
22. Traveling Salesman Problem 1 [All-Different Constraint]
 - Example models the Traveling Salesman Problem using an All-Different Constraint.
23. Traveling Salesman Problem 2 [Multiple All-Different Constraints]
 - This example is identical to Example 22, but this example models four decision variables using four blocks of variables, each of size 1.
24. Simulation Example 2
 - This example illustrates the use of the Truncate function to define the uncertain distributions; it also illustrates the use of a simple binary search as an alternative to the PsiTarget function.
25. Simulation Example 3
 - This example is similar to example24; the difference lies in how we model the variables and the uncertain functions. In this example, we create a single block of 2 variables, and just 1 uncertain function block with 100 functions.
26. Simulation Rank Correlations Example
 - Example illustrates how to specify a correlation matrix.
27. Optimization within Simulation Example
 - In this example we find the optimal order quantity automatically, by embedding an optimization problem inside each trial of a simulation problem.
28. Simulation within Optimization Example
 - In this example, which is similar to example 27, we find the optimal order quantity by embedding a simulation problem inside of an optimization problem.
29. Simulation Optimization using SIP/SLURP
 - Example generates a sample of demand and price (the uncertain variables) in the main program, stores these sampled values in appropriate arrays, then uses these values to compute the constraints and the objective of the optimization problem in the Evaluator.
30. Sudoku with All-Different Constraints
 - Constructs 3 arrays of variable objects each representing 9 blocks of variables, where each block will contain 9 variables.
31. Simulation Statistical Functions Example
 - Illustrates the different statistical functions that are supported for uncertain variables.
32. PSD Correlation Matrix Example
 - This example demonstrates using a correlation matrix to correlate input uncertain variables.

There are five additional examples installed into the C++ Examples directory. You can run each of them, by opening Visual Studio and the appropriate solution file inside of each directory described below.

1. CollegeFundModel
 - This example illustrates how to load and solve a model from an Excel workbook that contains a Solver simulation model. The simulation predicts the amount of funds needed to cover college tuition.
2. CutStock
 - A cutting stock problem involves allocating space on large sheets of paper or timbers to be cut into smaller pieces to meet demand while minimizing waste.
3. Portfolio
 - An example that uses the Markowitz method to obtain the best combination of stocks to minimize the variance of the portfolio given a return.
4. ProductMix
 - This example demonstrates how to load and solve a model from an Excel workbook that contains a Solver optimization model.
5. TSP
 - This example uses the Evolutionary Engine to solve a classic Traveling Salesman Problem.

Calling the Solver SDK from .NET

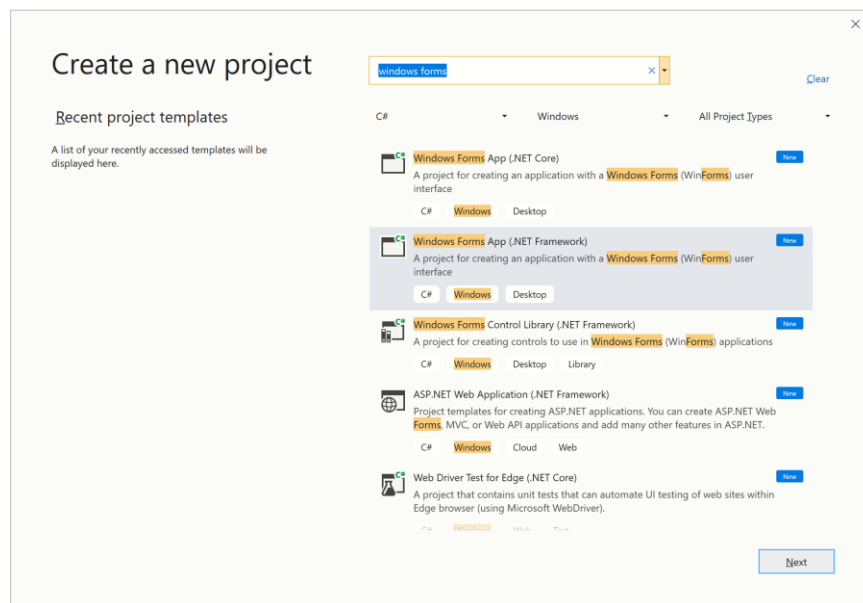
Introduction

This chapter details the steps required to compile, link and run a simple linear, nonlinear, and simulation example using the Solver SDK using 64-bit Microsoft Visual Studio .NET (C#, F# and VB.NET). *Important note: The F# wrapper can be called through VS 2019.*

Creating a New C# Project

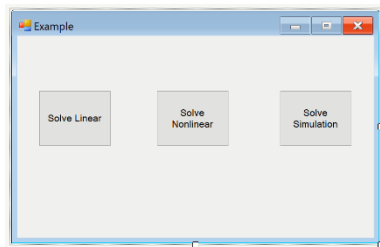
First, follow the steps in the "Installation" chapter, which will install all required SDK files, including Example files, into the Frontline Systems directory located at C:\Program Files.

1. **Create a new project.** Start Microsoft Visual Studio, select File | New Project, select C# for Language and Windows Forms App (.NET Framework) for Template. Then click Next.



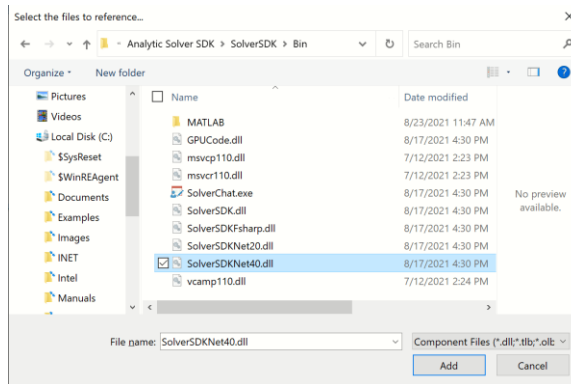
Type a name of your choice in the Project name field, and then click OK. This example uses the name CSharpNETProject

A new form will open. Construct a form of your choice using the Visual Studio Toolbox. In this example, the simple form shown below will be referenced.



2. **Set a reference and add an (optional) directive.** In order to access the methods and properties in the Solver SDK, you must set a reference. The SDK itself is a C++ unmanaged code DLL, but we can set a reference to a "wrapper" DLL. During installation of the SDK, two .NET DLLs are installed into the Bin directory located at C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK: SolverSDKNET20.dll for NET 2.0/3.0/3.5 and SolverSDKNET40.dll for .NET4.0. After you set the appropriate reference (depending on your version of .NET), your C# code will call the .NET DLL which in turn will call the SolverSDK.dll. At runtime, the SolverSDKNETXX.dll **must** be able to find the SolverSDK.dll. Simply copying the SolverSDK.dll into the same directory as the SolverSDKNETXX.dll will ensure that the SolverSDK.dll is found.

To add a reference, click **References** in the Solution Explorer and select **Add Reference**. Select **Browse**, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Bin, select the appropriate **SolverSDKNetXX.dll** and press **Add**.



Adding the directive, *using SolverPlatform*, is optional in C#. If you choose to forgo the directive, then you must add "SolverPlatform" to the beginning of each SDK function call. For example:

```
using (SolverPlatform.Problem prob = new
SolverPlatform.Problem())

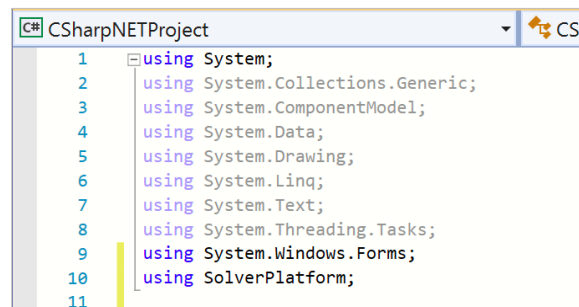
SolverPlatform.DoubleMatrix objcoeff = new
SolverPlatform.DoubleMatrix(1, nvars);

prob.Solver.SolverType = SolverPlatform.Solver_Type.Maximize;

prob.Model.AllLinear[SolverPlatform.Function_Type.Objective] =
objcoeff;

prob.Model.AllLinear[SolverPlatform.Function_Type.Constraint] =
concoeff;
```

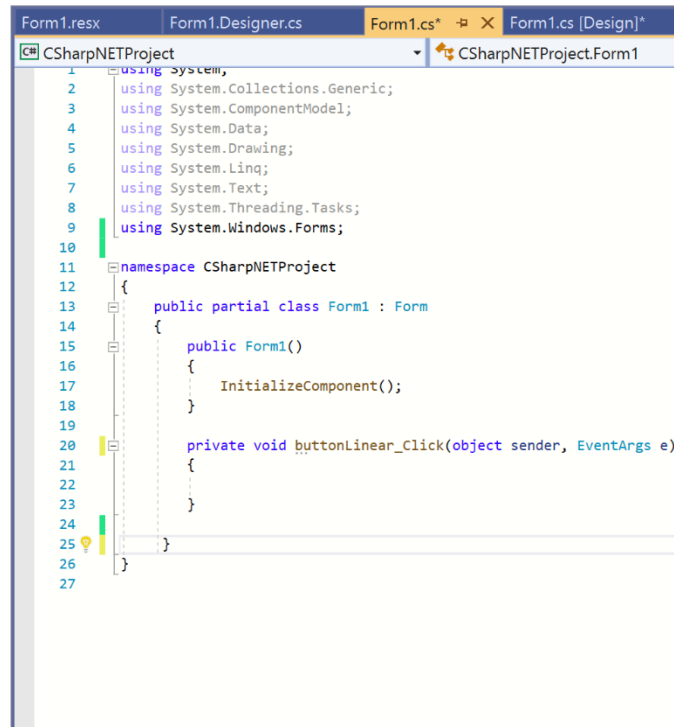
This example uses the *using SolverPlatform* directive which is shown in the screenshot below.



From here you can solve either a linear, nonlinear, or simulation model. For more details on each, please read on.

Creating a Linear Model

1. **First steps.** Follow the steps in the section, *Creating a New C# Project*, and then double click on the *SOLVE LINEAR* button. Doing so will open the *buttonLinear_Click* event which is where we will write our code to solve the linear optimization problem.



2. **Create the linear model.** In this section, we will solve a linear optimization problem with two variables (x_1 and x_2) and three constraints. Both final variable values must be zero or larger.

Maximize $2x_1 + 3x_2$

Subject To:

$9x_1 + 6x_2 \leq 54$

$6x_1 + 7x_2 \leq 42$

$5x_1 + 10x_2 \leq 50$

x_1, x_2 non-negative

To begin, the example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model.

```
int nvars = 2;
```

```
int ncons = 3;
```

The next few lines of code set up the objective and constraint coefficient matrices using two separate double matrices: *objcoeff* – for the objective coefficients and *concoeff* – for the constraint coefficients. Since there is only one objective, we pass the double matrix, *objcoeff*, of size *nvars* (number of variables) x 1 for the objective coefficients and the second double matrix, *concoeff*, of size *nvars* x *ncons* (number of constraints) for the constraint coefficients. Notice that we are using *NZBgn()* and *NZEnd()* when passing the constraint matrix (*concoeff*) as explained in the subsection *Passing Sparse Matrices* in the chapter *Designing Your Application*. As a result, the Solver SDK will not update the *concoeff* DoubleMatrix object until *NZEnd()* is executed.

```
DoubleMatrix objCoeff = new DoubleMatrix(1, nvars);
```

```
objCoeff[0, 0] = 2;
```

```
objCoeff[0, 1] = 3;
```

```
DoubleMatrix conCoeff = new DoubleMatrix(ncons, nvars);
```

```
conCoeff.NZBgn();
```

```
conCoeff [0, 0] = 9;
```

```

conCoeff [0, 1] = 6;
conCoeff [1, 0] = 6;
conCoeff [1, 1] = 7;
conCoeff [2, 0] = 5;
conCoeff [2, 1] = 10;
conCoeff.NZEnd();

```

Next, we create an array to hold the upper bounds of the constraint right hand sides.

```
double[] ub_constraints = { 54, 42, 50 };
```

This next line of code (*using (Problem prob = new Problem())*) creates an instance of the *CProblem* class. There are two different constructors which create a *Problem*. The first constructor (*Problem prob = new Problem()*) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call *prob.VarDecision.Size* to automatically add a variable block or create a variable block using *Variable vars(Variable_Type_Decision, "vars", 1)* where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add constraints automatically using *prob.FcnConstraint.Size* or create a function with *Function con1(Function_Type_Constraint, "con", 1)* (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling *prob.Functions.Add(con)* to add the function to the Problem.

The second constructor, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type.Minimize*, *Solver_Type.Maximize* or *Solver_Type.Simulate*) the number of variables (*nvars*) and the number of constraints (*ncons*). An example of the second constructor is: *using (Problem problem = new Problem(Solver_Type.Maximize, nvars, ncons))*. Note that we are utilizing the C# *using* statement which allows all properties/methods/functions under the *CProblem* class to be available for the remainder of the *Problem*.

```
using (Problem prob = new Problem())
```

These two lines of code pass the number of decision variables (*prob.VarDecision.Size = nvars*) and constraints (*prob.FcnConstraint.Size = ncons*) that are included in the model.

```

prob.VarDecision.Size = nvars;
prob.FcnConstraint.Size = ncons;

```

As mentioned above, we could have added the variables using the two lines of code below. The first line of code creates a variable block of type *Variable_Type_Decision*, names the block "vars" and then sets the size of the block to 2. The second line of code adds the *vars* variable block to the Problem.

```

Variable vars (Variable_Type_Decision, "vars",2);
prob.Variables.Add(vars);

```

In addition, we could have added the constraints using the two lines of code below. The first line of code creates a function block of type *Variable_Type_Constraint*, names the block "cons" and then sets the size of the block to 3. The second line of code adds the *cons* function block to the Problem.

```

Function cons (Function_Type_Constraint, "cons",3);
prob.Functions.Add(cons);

```

To tell the SDK to maximize the model, we use the line of code below. Alternatively, *Solver_Type_Minimize* will result in the model being minimized.

```
prob.Solver.SolverType = Solver_Type.Maximize;
```

We pass the *ub_constraints* array to the SDK. Recall that this array holds the upper bounds of the constraint right hand sides.

```
prob.FcnConstraint.UpperBound = ub_constraints;
```

To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
prob.VarDecision.NonNegative();
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass upper and lower variable bounds to the SDK.

3. **Call AllLinear functions and solve the linear model.** The line of code, *prob.Model.AllLinear[Function_Type.Objective]*, passes the objective coefficient matrix and *prob.Model.AllLinear[Function_Type.Constraint]* passes the constraint coefficient matrix created and initialized above. Calling *prob.Model.AllLinear* sets the problem type to "linear" (*Problem_Type.OptLP*) and will automatically select the best linear engine to solve the problem. If no additional Large Scale "plug in" engines have been added, then the LP/Quadratic engine will be selected.

```
prob.Model.AllLinear[Function_Type.Objective] = objcoeff;  
prob.Model.AllLinear[Function_Type.Constraint] = concoeff;
```

Solve the optimization model by calling:

```
problem.Solver.Optimize();
```

4. **Obtain and display the solution to the linear model.** We obtain the solution status along with the final objective, constraint and decision variable values,

```
Optimize_Status status = problem.Solver.OptimizeStatus;  
double[] varValue = problem.VarDecision.FinalValue.Array;  
double[] conValue = problem.FcnConstraint.FinalValue.Array;  
double[] objValue = problem.FcnObjective.FinalValue.Array;
```

and use message boxes to display the answer.

```
MessageBox.Show("Optimize Status = " + status);  
MessageBox.Show("x1 = " + varvalue[0] + ", x2 = " + varvalue[1]);  
MessageBox.Show("Constraint 1 = " + conValue[0] + ", Constraint 2  
= " + conValue[1] + ", Constraint 3 = " + conValue[2]);  
MessageBox.Show("Objective = " + objValue[0]);
```

5. **Catch any errors.** The catch statement handles any exceptions.

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

6. **Build and run the application.** To compile and link this project, select Build | Build CSharpNETProject. Then use Debug | Start Debugging to run the program. Press the command button **Solve Linear** to solve the linear optimization example.

Full Code

```
private void ButtonLinear1_Click_1(object sender, EventArgs e)
{
    {
        /*
        Example 1: Solves the LP model:
        Maximize    2 x1 + 3 x2
        Subj to     9 x1 + 6 x2 <= 54
        6 x1 + 7 x2 <= 42
        5 x1 + 10 x2 <= 50
        x1, x2 non-negative
        LP solution: x1 = 2.8, x2 = 3.6
        Objective = 16.4
        Matrix is passed rowwise, dense.
        */
        try
        {
            int nvars = 2; // '2 decision variables
            int ncons = 3; // '3 constraint functions

            double[] rhsValue = { 54, 42, 50 };

            DoubleMatrix objCoeff = new DoubleMatrix(1, nvars);
            objCoeff[0, 0] = 2;
            objCoeff[0, 1] = 3;

            DoubleMatrix conCoeff = new DoubleMatrix(ncons, nvars);
            conCoeff.NZBgn();
            conCoeff[0, 0] = 9;
            conCoeff[0, 1] = 6;
            conCoeff[1, 0] = 6;
            conCoeff[1, 1] = 7;
            conCoeff[2, 0] = 5;
            conCoeff[2, 1] = 10;
            conCoeff.NZEnd();

            double[] ub_constraints = { 54, 42, 50 };

            //create a problem
            using (Problem problem = new Problem())
            {
                problem.VarDecision.Size = nvars;
                problem.FcnConstraint.Size = ncons;

                problem.Solver.SolverType = Solver_Type.Maximize;

                problem.FcnConstraint.UpperBound.Array = ub_constraints;
                problem.VarDecision.NonNegative();

                /*linear model; pass the objective coefficients*/
                problem.Model.AllLinear[Function_Type.Objective] =
                    objCoeff;

                /*linear model; pass the constraint coefficients*/
                problem.Model.AllLinear[Function_Type.Constraint] =
                    conCoeff;
            }
        }
    }
}
```

```

        problem.Solver.Optimize(); // solve model

        Optimize_Status status = problem.Solver.OptimizeStatus;
        // optimization status

        double[] varvalue = problem.VarDecision.FinalValue.
        Array; //final variable values

        double[] conValue = problem.FcnConstraint.FinalValue.
        Array; //final constraint values

        double[] objValue = problem.FcnObjective.FinalValue.
        Array; //final objective value

        //Note how the use of 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 1 block of each.

        MessageBox.Show("Optimize Status = " + status);
        MessageBox.Show("x1 = " + varvalue[0] + ", x2 = " +
        varvalue[1]);
        MessageBox.Show("Constraint 1 = " + conValue[0] + ",
        Constraint 2 = " + conValue[1] + ", Constraint 3 = " +
        conValue[2]);
        MessageBox.Show("Objective = " + objValue[0]);
    }
}
catch (SolverException ex)
{
    MessageBox.Show(
        "Exception " + ex.ExceptionType + "\r\n" + "Exception "
        + ex.Message, "SDK Exception Occurred");
}
}
}

```

Creating a Nonlinear Problem

1. **Basic first steps.** Follow the steps above in the section, *Creating a New C# Project*, and then double click on the *SOLVE NONLINEAR* button. Doing so will open the *buttonNonlinear_Click* event which is where we will write our code to solve the nonlinear optimization problem.
2. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constraint ($x + y$) forces the total of the two variables to equal to 1 and the second constraint ($x * y$) forces the product of the two variables to be greater than zero. Both final variable values must be zero or greater.

```

Minimize  $x^2 + y^2$ 
Subject to:
 $x + y = 1$ 
 $x * y \geq 0$ 
 $x, y$  are non-negative

```

First, we initialize the number of variables (*nvars*) and the number of constraints (*ncons*).

```
int nvars = 2, ncons = 2;
```

Next, we create two arrays of size *ncons* (2 constraints) to hold the upper and lower bounds on the constraint right hand sides. Notice that the upper and lower bound of the first constraint is 1 to denote an equality constraint. *Constants.PINF* sets the upper bound of the second constraint to positive infinity.

```
double[] ub_constraints = { 1.0, Constants.PINF };
double[] lb_constraints = { 1.0, 0 };
```

The next two lines of code create two arrays, *lb* and *ub*, to hold the lower and upper variable bounds, respectively.

```
double[] lb = { 0, 0 };
double[] ub = { 1, 1 };
```

This next line of code (*Problem prob = new problem(Solver_Type.Minimize, nvars, ncons)*) creates an instance of the Problem class. There are two different constructors which create a *Problem*. The first constructor (*Problem prob = new Problem()*) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call *prob.VarDecision.Size* to automatically add a variable block or create a variable block using *CVariable vars(Variable_Type_Decision, "vars", 1)* where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add constraints automatically using *prob.FcnConstraint.Size* or create a function with *Function con1(Function_Type_Constraint, "con", 1)* (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling *prob.Functions.Add(con)* to add the function to the Problem.

The second constructor, as shown here, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type.Minimize*, *Solver_Type.Maximize* or *Solver_Type.Simulate*) the number of variables (*nvars*) and the number of constraints (*ncons*). Note that we are utilizing the C# *using* statement which allows all properties/methods/functions under the *CProblem* class to be available for the remainder of the *Problem*.

```
using (Problem prob = new problem(Solver_Type.Minimize, nvars,
ncons))
```

Here we pass the constraint upper and lower bounds to the SDK,

```
prob.FcnConstraint.UpperBound.Array = ub_constraints;
prob.FcnConstraint.LowerBound.Array = lb_constraints;
```

following up with the variable upper and lower bounds.

```
prob.VarDecision.UpperBound.Array = ub;
prob.VarDecision.LowerBound.Array = lhs;
```

3. **Create the nonlinear evaluator.** In a linear model or quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator (*Eval_Type.Function*) to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied. (It is possible to pass these derivatives directly to the SDK using an evaluator of type *Eval_Type.Gradient*. For more

information see the subheading, *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter, *Solver SDK Evaluators*.)

C# requires that an evaluator appear above the function where it is called so this code will be placed above the *buttonNonlinear_Click* event.

We create and pass the evaluator (as described above) to the SDK.

```
prob.Evaluators[Eval_Type.Function].OnEvaluate += new
EvaluateEventHandler(Nonlinear_OnEvaluate)
```

We will write the code for our *eval_type.function* evaluator inside of this event handler.

4. **Write the nonlinear evaluator.** Inside of our *Nonlinear_OnEvaluate* event handler, we first create a convenient pointer, *p*, to the Problem,

```
Problem p = evaluator.Problem;
```

and another pointer, *pVar*, to the variables.

```
double[] pVar = p.VarDecision.Value.Array;
```

Next, we obtain a pointer to the constraint functions and calculate the constraints.

```
Double[] pFcn = p.FcnConstraint.Value.Array;
```

Here we pass the first constraint, $x + y$,

```
pFcn[0] = pVar[0] + pVar[1];
```

and here we pass the second, $x * y$.

```
pFcn[1] = pVar[0] * pVar[1];
```

Then we compute the objective function, $x * x + y * y$.

```
p.FcnObjective.Value[0] = pVar[0] * pVar[0] + pVar[1] * pVar[1];
```

In C#, we must pass the constraint values back to the SDK.

```
p.FcnConstraint.Value.Array = pFcn;
```

Finally, we tell the SDK to continue optimizing by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action.Evaluate* and *Engine_Action..Stop*. *Engine_Action.Evaluate* forces the SDK to call the evaluator of type *Eval_Type.Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action.Stop* will indicate to the SDK that an error has occurred during the calculation of the functions (objective or constraints). The Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine (for solving nonsmooth problems) will disregard the current point and continue solving.

```
return Engine_Action.Continue;
```

An alternative way to write this evaluator without using the *problem*, *pVar* and *pFcn* pointers is below.

```
Engine_Action Form1_OnEvaluate(Evaluator evaluator)
```

```
{
    evaluator.Problem.FcnConstraint.Value[0] =
        evaluator.Problem.VarDecision.Value[0] +
        evaluator.Problem.VarDecision.Value[1];
    evaluator.Problem.FcnConstraint.Value[1] =
        evaluator.Problem.VarDecision.Value[0] *
```



```

        evaluator.Problem.VarDecision.Value[1];
evaluator.Problem.FcnObjective.Value[0] =
    evaluator.Problem.VarDecision.Value[0] *
    evaluator.Problem.VarDecision.Value[0] +
    evaluator.Problem.VarDecision.Value[1] *
    evaluator.Problem.VarDecision.Value[1];
return Engine_Action.Continue;
}

```

Our evaluator is complete. We will now finish the code inside of the *button_NonlinearClick* event.

5. **Solve the nonlinear problem.** Back inside the *button_NonlinearClick* event, we provide the problem type as nonlinear (*Problem_Type_OptNLP*) which directs the SDK to select the GRG engine to solve the model. Alternatively, we also could have set the GRG engine manually (in either the SDK Platform or the SDK Pro) by calling:

prob.Engine = problem.Engines.GRGName.

```
prob.ProblemType = Problem_Type.OptNLP;
```

To solve the optimization problem, we call:

```
prob.Solver.Optimize();
```

6. **Obtain the solution status and final answer.** We obtain the solution status, along with the final objective and variable values,

```
Optimize_Status status = prob.Solver.OptimizeStatus;
```

```
double[] varValue = problem.VarDecision.FinalValue.Array;
```

```
double[] fcnValue = problem.FcnConstraint.FinalValue.Array;
```

```
double[] objValue = problem.FcnObjective.FinalValue.Array;
```

and display the answer with multiple message boxes.

```
MessageBox.Show("Status = " + status);
```

```
MessageBox.Show("x1 = " + varvalue[0] + ", x2 = " + valValue[1]);
```

```
MessageBox.Show("Constraint 1 = " + fcnValue[1] + ", Constraint 2  
= " + fcnValue[2]);
```

```
MessageBox.Show("Obj = " + objvalue[0]);
```

7. **Catch any errors.**

```
Catch (SolverException ex)
```

```
{MessageBox.Show(
```

```
"Exception " + ex.ExceptionType + "\r\n" + "Exception " +  
ex.Message, "SDK Exception Occurred");
```

```
}
```

8. **Build and run the application.** To compile and link this project, select Build | Build CSharpNETProject. Then use Debug | Start Debugging to run the program. Press the command button **Solve Nonlinear** to solve the nonlinear optimization example.

Full Code

```
private Engine_Action Nonlinear_OnEvaluate(Evaluator evaluator)
{

```

```

        Problem p = evaluator.Problem;

        double[] pVar = p.VarDecision.Value.Array;
        double[] pFcn = p.FcnConstraint.Value.Array;

        pFcn[0] = pVar[0] + pVar[1];
        pFcn[1] = pVar[0] + pVar[1];

        p.FcnObjective.Value[0] = pVar[0] * pVar[0] + pVar[1] *
        pVar[1];
        p.FcnConstraint.Value.Array = pFcn;

        return Engine_Action.Continue;
    }

    private void buttonNonlinear2_Click_2(object sender,
    EventArgs e)
    {
        /*
        Solves the problem:
        Minimize  $X^2 + Y^2$ 
        Subject to:
         $X + Y = 1$ 
         $X * Y \geq 0$ 
        (Solution is  $X = Y = 0.5$ , Objective = 0.5)
        */
        try
        {
            int nvars = 2, ncons = 2;

            double[] ub_constraints = { 1.0, Constants.PINF };
            double[] lb_constraints = { 1.0, 0 };

            double[] lb = { 0, 0 };
            double[] ub = { 1, 1 };

            using (Problem problem = new Problem(Solver_Type.Minimize,
            nvars, ncons))
            {
                problem.FcnConstraint.UpperBound.Array = ub_constraints;
                problem.FcnConstraint.LowerBound.Array = lb_constraints;

                problem.VarDecision.UpperBound.Array = ub;
                problem.VarDecision.LowerBound.Array = lb;

                problem.Evaluators[Eval_Type.Function].OnEvaluate += new
                EvaluateEventHandler(Nonlinear_OnEvaluate);

                problem.ProblemType = Problem_Type.OptNLP;

                problem.Solver.Optimize();

                Optimize_Status status = problem.Solver.OptimizeStatus;

                double[] objValue = problem.FcnObjective.FinalValue.Array;
                double[] varValue = problem.VarDecision.FinalValue.Array;
            }
        }
    }

```

```

        double[] fcnValue = problem.FcnConstraint.FinalValue.Array;

        MessageBox.Show("Status = " + status );
        MessageBox.Show("x1 = " + varValue[0] + ", x2 = " +
            varValue[1]);
        MessageBox.Show("Constraint 1 = " + fcnValue[0] + ",
            Constraint 2 = " + fcnValue[1] );
        MessageBox.Show("Objective = " + objValue[0]);
    }
}
catch (SolverException ex)
{
    MessageBox.Show(
        "Exception " + ex.ExceptionType + "\r\n" + "Exception " +
        ex.Message, "SDK Exception Occurred");
}
}

```

Creating a Simulation Model

1. **Basic first steps.** Follow the steps in the chapter, *Creating a New C# Project*, and then double click on the *SOLVE SIMULATION* button. Doing so will open the *buttonSimulate_Click* event which is where we will write our code to solve the simulation problem.
2. **Create a simulation problem.** The simulation model that we will be solving in this example contains two uncertain variables, X and Y. The uncertain variable X has a uniform distribution with a lower parameter of 0 and an upper parameter of 2. The uncertain variable Y has a Normal distribution with a mean and standard deviation equal to 1. In addition, there are two uncertain functions. The first uncertain function is the total of the two uncertain variables (X + Y). The second is the product of the two uncertain variables (X * Y).

```
X = PsiUniform(0,2)   and   Y = PsiNormal(1,1)
```

```
Function[1]: X + Y
```

```
Function[2]: X * Y
```

First, we initialize the number of uncertain variables (*nvars*) and functions (*nfcns*).

```
int nvars = 2, nfcns = 2;
```

This example uses a correlation matrix which measures the degree to which the movements of the two uncertain variables are related. This is optional. The correlation matrix is:

1	.5
.5	1

The elements are being passed row-wise.

```
double[] correl = new double[] { 1.0, 0.5, 0.5, 1.0 };
```

This next line of code (*Problem prob = new problem(Solver_Type.Simulate, nvars, nfcns)*) creates an instance of the Problem class. There are two different constructors which create a Problem. The first constructor (*Problem prob = new Problem()*) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call *prob.VarUncertain.Size* to automatically add a variable block or create a variable block using *Variable*

`vars(Variable_Type_Uncertain, "vars", 1)` where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling `prob.Variables.Add(vars)`. This is true for functions as well. A user can either add uncertain functions automatically using `prob.FcnConstraint.Size` or create a function with `Function con1(Function_Type_Uncertain, "con", 1)` (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling `prob.Functions.Add(con)` to add the function to the Problem.

The second constructor, as shown here, not only creates the Problem, but also passes the `Solver_Type`, (`Solver_Type.Minimize`, `Solver_Type.Maximize` or `Solver_Type.Simulate`) the number of variables (`nvars`) and the number of uncertain functions (`nfcns`). Note that we are utilizing the C# `using` statement which allows all properties/methods/functions under the `CProblem` class to be available for the remainder of the `Problem`.

```
using (Problem problem = new Problem(Solver_Type.Simulate, nvars,
nfcns))
```

Next, we pass the distributions of the two uncertain variables, Uniform and Normal.

```
problem.Model.VarDistribution[0] = new
Distribution("PsiUniform(0,2)");

problem.Model.VarDistribution[1] = new
Distribution("PsiNormal(1,1)");
```

Here, we create a new double matrix of size `nvars` x `nvars` (2 x 2) to hold the correlation matrix. We are passing the `correl` array row-wise.

```
problem.Model.DistCorrelations = new
DoubleMatrix(Array_Order.ByRow, nvars, nvars, correl);
```

3. **Create the simulation evaluator.** When running a simulation, the evaluator `Eval_Type_Sample` is called so the user can calculate the uncertain functions, much like `Eval_Type_Function` is called to compute the objective function and constraints in a nonlinear optimization problem.

We create the `Eval_Type.Sample` evaluator (as described above) and pass the evaluator to the SDK.

```
problem.Evaluators[Eval_Type.Sample].OnEvaluate += new
EvaluateEventHandler(SimEvaluator_OnEvaluate);
```

We will write the code for our `eval_type.function` evaluator inside of this event handler.

4. **Complete the simulation evaluator.** Inside of our `Simulation_OnEvaluate` event, we first create a convenient pointer, `p`, to the Problem,

```
Problem p = evaluator.Problem;
```

and a second pointer, `pVar`, to the uncertain variable values .

```
double[] pVar = p.VarUncertain.Value.Array;
```

Next, we create a pointer, `pFcn`, to the uncertain functions.

```
double[] pFcn = p.FcnUncertain.Value.Array;
```

We compute the uncertain function values given the current variable values.

```
pFcn[0] = pVar[0] + pVar[1];
```

```
pFcn[1] = pVar[0] * pVar[1];
```

In C#, we must pass the uncertain function values back to the SDK.

```
p.FcnUncertain.Value.Array = pFcn;
```

Finally, we tell the SDK to continue simulating.

```
return Engine_Action.Continue;
```

Finally, we tell the SDK to continue simulating by returning *Engine_Action.Continue*. There are two additional *Engine_Action* functions, *Engine_Action.Evaluate* and *Engine_Action.Stop*. *Engine_Action.Evaluate* forces the SDK to call the evaluator of type *Eval_Type.Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action.Stop* will indicate to the SDK that an error has occurred during the calculation of the uncertain functions. The SDK will mark the current trial as an error trial and continue with the remaining trials.

An alternative way to write this evaluator without using the *p*, *pVar* and *pFcn* pointers is below.

```
Engine_Action Simulation_OnEvaluate(Evaluator evaluator)
{
    evaluator.Problem.FcnUncertain.Value[0] =
        evaluator.Problem.VarUncertain.Value[0] +
        evaluator.Problem.VarUncertain.Value[1];
    evaluator.Problem.FcnUncertain.Value[1] =
        evaluator.Problem.VarUncertain.Value[0] *
        evaluator.Problem.VarUncertain.Value[1];
    return Engine_Action.Continue;
}
```

Our evaluator is complete. We will now finish the code inside of the *button_SimulationClick* event.

5. **Solve the simulation problem.** Back inside the *button_SimulationClick* event, we call the following line of code to run the simulation.

```
prob.Solver.Simulate();
```

6. **Display various simulation statistics.** We obtain the simulation status and various uncertain variable and uncertain function statistics using message boxes.

```
MessageBox.Show("Simulation Example 1");
```

```
MessageBox.Show("Status = " + prob.Solver.SimulateStatus);
```

Display statistics on the two uncertain variables.

```
MessageBox.Show("\r\nStatistics on Uncertain Variables");
```

```
    for (int i = 0; i < nvars; i++)
```

```
    {
```

```
        MessageBox.Show("\r\nVariable " + i + "\r\n");
```

```
        MessageBox.Show("Minimum = "
            + prob.VarUncertain.Statistics.Minimum[i]);
```

```
        MessageBox.Show("Maximum = " +
            prob.VarUncertain.Statistics.Maximum[i]);
```

```
        MessageBox.Show("Mean = " +
            prob.VarUncertain.Statistics.Mean[i]);
```

```
        MessageBox.Show("StdDev = " +
            prob.VarUncertain.Statistics.StdDev[i]);
```

```
        MessageBox.Show("Variance = " +
            prob.VarUncertain.Statistics.Variance[i]);
```

```

    MessageBox.Show("Skewness = " +
    prob.VarUncertain.Statistics.Skewness[i]);
    MessageBox.Show("Kurtosis = " +
    prob.VarUncertain.Statistics.Kurtosis[i]);
    MessageBox.Show("10th Percentile = " +
    prob.VarUncertain.Percentiles[i, 10]);
    MessageBox.Show("90th Percentile = " +
    prob.VarUncertain.Percentiles[i, 90]);
}

```

Display statistics on the two uncertain functions.

```

MessageBox.Show("\r\nStatistics on Uncertain Functions");
for (int i = 0; i < nfcns; i++)
{
    MessageBox.Show("\r\nFunction " + i);
    MessageBox.Show("Minimum = " +
    prob.FcnUncertain.Statistics.Minimum[i]);
    MessageBox.Show("Maximum = " +
    prob.FcnUncertain.Statistics.Maximum[i]);
    MessageBox.Show("Mean = " +
    prob.FcnUncertain.Statistics.Mean[i] );
    MessageBox.Show("StdDev = " +
    prob.FcnUncertain.Statistics.StdDev[i]);
    MessageBox.Show("Variance = " +
    prob.FcnUncertain.Statistics.Variance[i]);
    MessageBox.Show("Skewness = " +
    prob.FcnUncertain.Statistics.Skewness[i]);
    MessageBox.Show("Kurtosis = " +
    prob.FcnUncertain.Statistics.Kurtosis[i]);
    MessageBox.Show("10th Percentile = " +
    prob.FcnUncertain.Percentiles[i, 10]);
    MessageBox.Show("90th Percentile = " +
    prob.FcnUncertain.Percentiles[i, 90]);
}

```

7. Catch any errors. The catch statement will handle any exceptions.

```

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

```

8. **Build and run the application.** To compile and link this project, select Build | Build *UserGuideExample*. Then use Debug | Start Debugging to run the program. Press the command button **Solve Simulation** to solve the simulation example.

Full Code

```
private Engine_Action SimEvaluator_OnEvaluate(Evaluator
evaluator)
{
    // First, we ask for the current uncertain variable values
    double[] Var = evaluator.Problem.VarUncertain.Value.Array;

    MessageBox.Show("Evaluation x1 = " + Var[0] + ", x2 = " +
Var[1] + ", Trial = " + evaluator.Problem.Engine.Stat.
FunctionEvals );

    // We compute 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;
}
private void buttonSimulate_Click(object sender, EventArgs e)
{
    /*
    Example program calling the Monte Carlo Simulation engine.
    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
    */

    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.VarDistribution[0] = new
            Distribution("PsiUniform(0,2)");

            problem.Model.VarDistribution[1] = new
            Distribution("PsiInvNormal(1,1)");

            problem.Model.DistCorrelations = new
            DoubleMatrix(Array_Order.ByRow, nvars, nvars, correl);

            problem.Engine.Params["NumThreads"].Value = 1;
        }
    }
}
```

```

problem.Solver.Simulate();

MessageBox.Show("Status = " +
problem.Solver.SimulateStatus);
MessageBox.Show("Statistics on Uncertain
Variables");

for (int i = 0; i < nvars; i++)
{
    MessageBox.Show("Variable " + i);
    MessageBox.Show("Minimum = " + problem.VarUncertain.
Statistics.Minimum[i]);
    MessageBox.Show("Maximum = " + problem.VarUncertain.
Statistics.Maximum[i]);
    MessageBox.Show("Mean = " + problem.VarUncertain.
Statistics.Mean[i]);
    MessageBox.Show("StdDev = " + problem.
VarUncertain.Statistics.StdDev[i]);
    MessageBox.Show("Variance = " + problem.
VarUncertain.Statistics.Variance[i]);
    MessageBox.Show("Skewness = " + problem.
VarUncertain.Statistics.Skewness[i]);
    MessageBox.Show("Kurtosis = " + problem.
VarUncertain.Statistics.Kurtosis[i]);
    MessageBox.Show("10th Percentile = " + problem.
VarUncertain.Percentiles[i, 10] + "\r\n");
    MessageBox.Show("90th Percentile = " + problem.
VarUncertain.Percentiles[i, 90]);
}
MessageBox.Show("Statistics on Uncertain Functions");

for (int i = 0; i < nfcns; i++)
{
    MessageBox.Show("Function " + i );
    MessageBox.Show("Minimum = " + problem.
FcnUncertain.Statistics.Minimum[i]);
    MessageBox.Show("Maximum = " + problem.
FcnUncertain.Statistics.Maximum[i]);
    MessageBox.Show("Mean = " + problem.
FcnUncertain.Statistics.Mean[i]);
    MessageBox.Show("StdDev = " + problem.
FcnUncertain.Statistics.StdDev[i]);
    MessageBox.Show("Variance = " + problem.
FcnUncertain.Statistics.Variance[i]);
    MessageBox.Show("Skewness = " + problem.
FcnUncertain.Statistics.Skewness[i]);
    MessageBox.Show("Kurtosis = " + problem.
FcnUncertain.Statistics.Kurtosis[i]);
    MessageBox.Show("10th Percentile = " +
problem.FcnUncertain.Percentiles[i, 10]);
    MessageBox.Show("90th Percentile = " +
problem.FcnUncertain.Percentiles[i, 90]);
}
}
}
catch (SolverException ex)
{

```



```

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

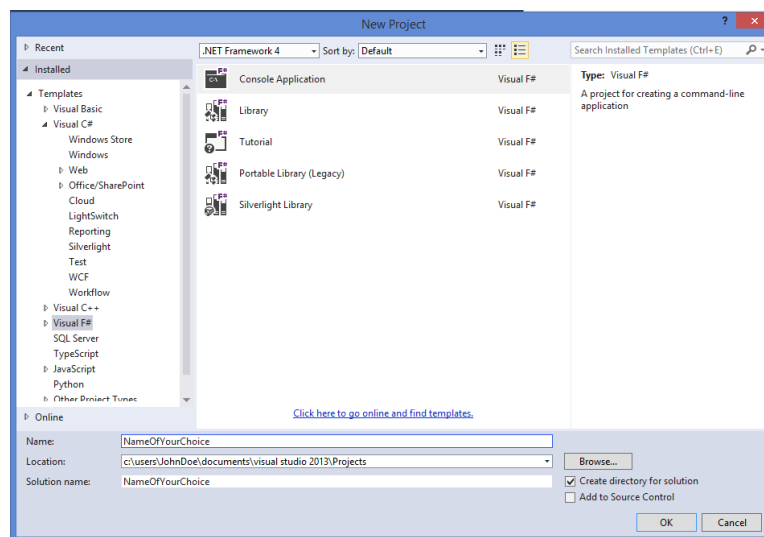
```

Creating a New F# Project

Users can utilize F# to call the Solver Platform SDK to solve optimization, simulation or simulation optimization models using the F# API. This API “wraps” around the SDK and allows users to take advantage of unique advantages offered by F#. Although the F# wrapper does not support all functionality provided in the SDK, this wrapper does support the most common uses such as solving linear, nonlinear and simulation models. The example code shown below is installed into C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\F#.

First, follow the steps in the "Installation" chapter, which will install all required SDK files, including Example files, into the Frontline Systems directory located at C:\Program Files. Note: This example can only be executed in *Microsoft Visual Studio 2013*. The F# wrapper is compatible with VS 2012 and VS2013.

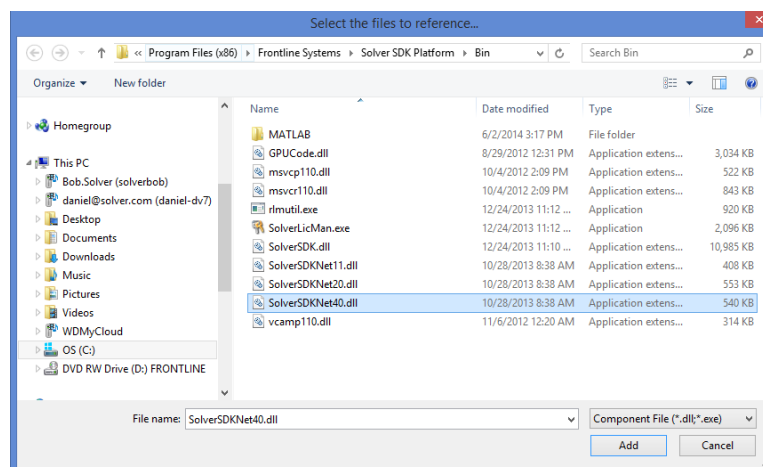
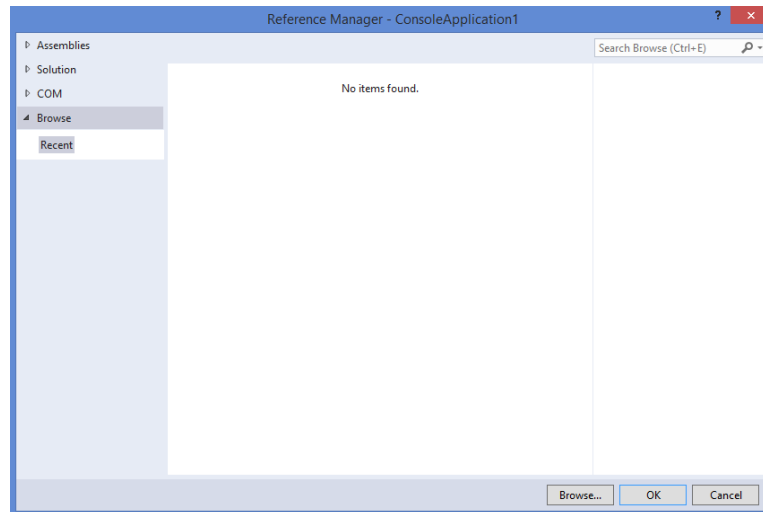
1. **Create a new project.** Start Microsoft Visual Studio, select File | New Project then select Console Application. Type a name of your choice in the Project name field, and then click OK.



A new blank console application will open.

2. **Set two references and add an (optional) directive.** In order to access the methods and properties in the Solver SDK when using F#, you must set two references. The SDK itself is a C++ unmanaged code DLL, but we can set a reference to an F# "wrapper" DLL. During installation of the SDK, four .NET DLLs are installed into the Bin directory located at C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK: SolverSDKNET12.dll for .NET1.1, SolverSDKNET20.dll for NET 2.0/3.0/3.5, and SolverSDKNET40.dll for .NET4.0. An F# “wrapper” is also installed, SolverSDKFsharp.dll. You must also set a reference to this file in order to easily call the SDK from F#. Note: F# programmers *can* use C# if desired. The F# wrapper is simply intended to make it easier for F# programmers to call the SDK for common scenarios. After you set the appropriate reference to the .NET DLL (depending on your version of .NET), your F# code will call the F# “wrapper”, which will call the .NET DLL which in turn will call the SolverSDK.dll. At runtime, both the F# wrapper and the SolverSDKNETXX.dll **must** be able to find the SolverSDK.dll. Simply copying the SolverSDK.dll into the same directory as the SolverSDKNETXX.dll and F# wrapper will ensure that the SolverSDK.dll is found.

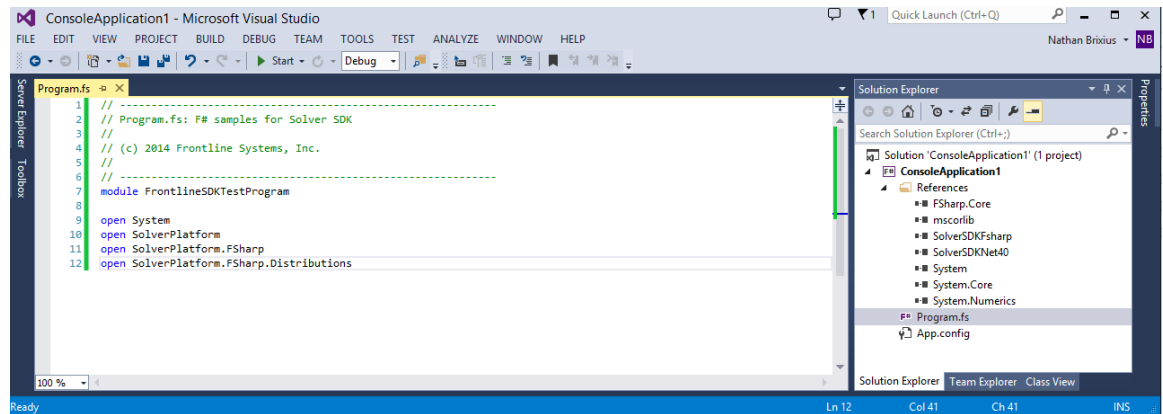
To add a reference to the .NET DLL and FSharpDLL, right-click **References** in the Solution Explorer and select **Add Reference**. Select **Browse** in the left pane and then the **Browse** button in the lower right corner, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Bin, select SolverSDKFsharp.dll and the appropriate **SolverSDKNetXX.dll**, then click **OK**.



Adding the directives, *open SolverPlatform* and *open SolverPlatform.FSharp* is optional in F#. If you choose to forgo the directive, then you must add "*SolverPlatform.FSharp*" to the beginning of each SDK function call. For example:

```
let model = SolverPlatform.FSharp.LinearModel(maximize, 2, 3)
```

This example uses the *open SolverPlatform* *open SolverPlatform.FSharp* directives which are shown in the screenshot below.



From here you can solve either a linear, nonlinear, or simulation model. For more details on each, please read on.

Creating a Linear Model in F#

1. **First steps.** Follow the steps in the section, *Creating a New F# Project*. In this example, we will name the function that will hold the linear example, *FSharpLinearExample()*.
2. **Create the linear model.** In this section, we will solve a linear optimization problem with two variables (x_1 and x_2) and three constraints. Both final variable values must be zero or larger.

```
Maximize 2 x1 + 3 x2
Subject To:
9 x1 + 6 x2 <= 54
6 x1 + 7 x2 <= 42
5 x1 + 10 x2 <= 50
x1, x2 non-negative
```

To begin, the example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model.

```
let nvars = 2
let ncons = 3
```

This next line of code (*let prob = LinearModel(maximize, nvars, ncons)*) creates an instance of the *CProblem* class, sets the problem type to "linear" and automatically selects the best linear engine to solve the problem. If no additional Large Scale "plug in" engines have been added, then the LP/Quadratic engine will be selected. Note: This is the only linear *problem* constructor available in the F# wrapper at this time.

LinearModel(Maximize, nvars, ncons) also passes the *Solver_Type* (*Minimize*, *Maximize* or *Simulate*), the number of variables (*nvars*) and the number of constraints (*ncons*). Note that we are utilizing the F# *let* statement which allows all properties/methods/functions under the *CProblem* class to be available for the remainder of the *Problem*.

```
let prob = LinearModel(Maximize, nvars, ncons)
```

To specify a matrix (for linear constraints and the objective) you simply specify a one-dimensional F# array with the appropriate size. For example, if you want to specify a 2x3 matrix then you would use `[| 1.0; 2.0; 3.0; 4.0; 5.0; 6.0 |]` where the first row coefficients would be the first three (1.0, 2.0, 3.0) and the second row coefficients would be the 2nd three (4.0, 5.0, 6.0). In this example, we'll use the following line of code to specify the objective coefficients. Recall the objective $2x_1 + 3x_2$ where 2 and 3 are the coefficients.

```
prob.Objective <- [2.; 3.]
```

The constraints are passed row-wise.

```
prob.Constraint <- [9.; 6.; 6.; 7.; 5.; 10.]
```

Next, we create an array to hold the upper bounds of the constraint right hand sides.

```
prob.ConstraintUpper = [ 54.; 42.; 50.];
```

To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
prob.NonNegativeVariables();
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass upper and lower variable bounds to the SDK.

3. Solve the linear model.

Solve the optimization model by calling:

```
prob.Solve();
```

4. Catch any errors. The catch statement handles any exceptions.

```
with | :? SolverPlatform.SolverException as ex -> printfn  
"Exception: %s " ex.Message)
```

5. Obtain and display the solution to the linear model. We obtain the solution status along with the final objective and decision variable values.

```
printfn ""  
printfn "Linear optimization example"  
if model.Status <> optimal then  
    printfn "Optimize Status: %A" model.Status  
    printfn "Objective = %.2f" model.ObjectiveValue  
    printfn "%A" model.Variables
```

6. Build and run the application. To compile and link this project, select Build | Build. Then use Debug | Start Debugging to run the program. An output window will appear.

Complete Code

```
let linearexample() =  
    //Creates a linear model with two vars and three constraints.  
    let model = LinearModel(maximize, 2, 3)  
  
    // The coefficients for the objective function.  
    model.Objective <- [ 2.; 3. ]  
  
    // The coefficients for the linear cons, given row-by-row.  
    model.Constraint <-  
        [ 9.; 6.;  
          6.; 7.;  
          5.; 10. ]  
  
    // The upper bounds for the constraints.  
    model.ConstraintUpper <- [ 54.; 42.; 50. ]  
    model.NonNegativeVariables()  
  
    try  
        model.Solve();  
    with  
        | :? SolverPlatform.SolverException as ex -> printfn "There  
            was an error solving the model: %s " ex.Message
```

```

printfn ""
printfn "Linear optimization example"
if model.Status <> optimal then
    printfn "Status: %A" model.Status
printfn "Objective = %.2f" model.ObjectiveValue
printfn "%A" model.Variables

```

Creating a Nonlinear Problem

1. **Basic first steps.** Follow the steps above in the section, *Creating a New F# Project*. In this example, we will name the function that will hold the linear example, *FSharpNonlinearExample()*.
2. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constraint ($x + y$) forces the total of the two variables to equal to 1 and the second constraint ($x * y$) forces the product of the two variables to be greater than zero. Both final variable values must be zero or greater.

```

Minimize x^2 + y^2
Subject to:
x + y = 1
x * y >= 0
x, y are non-negative

```

First, we initialize the number of variables (*nvars*) and the number of constraints (*ncons*).

```

let nvars = 2,
let ncons = 2;

```

This next line of code (*let prob = NonlinearModel(minimize, nvars, ncons)*) creates an instance of the *CProblem* class, sets the problem type to "nonlinear" and automatically selects the best nonlinear engine to solve the problem. If no additional Large Scale "plug in" engines have been added, then the LSGRG engine will be selected. Note: This is the only nonlinear *problem* constructor available in the F# wrapper at this time.

NonlinearModel(minimize, nvars, ncons) also passes the *Solver_Type* (*Minimize*, *Maximize* or *Simulate*), the number of variables (*nvars*) and the number of constraints (*ncons*). Note that we are utilizing the F# *let* statement which allows all properties/methods/functions under the *CProblem* class to be available for the remainder of the *Problem*.

```

let prob = new NonlinearModel(Minimize, nvars, ncons)

```

Here we pass the constraint upper and lower bounds to the SDK, (note "inf" denotes positive infinity)

```

prob.ConstraintLower <- [1.; 0.]
prob.ConstraintUpper <- [1.; inf]

```

following up with the variable upper and lower bounds.

```

prob.VariableLower[0.; 0.]
prob.VariableUpper[1.; 1.]

```

3. **Create the nonlinear evaluator.** In a linear model or quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to

compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied.

We create and pass the evaluator (as described above) to the SDK. The “c” array is the constraint array and the “x” arguments are the variables. The last function below ($x.[0] * x.[0] + x.[1] * x.[1]$) is the objective function. (In C++, C, VB.NET and VB6, the evaluator must appear in a separate class or module, however in F#, the evaluator resides in the same module as the rest of the model.)

```
prob.Evaluate <- fun x c ->
    c.[0] <- x.[0] + x.[1]
    c.[1] <- x.[0] * x.[1]
    x.[0] * x.[0] + x.[1] * x.[1]
```

4. **Solve the model.** We obtain the solution status, along with the final objective and variable values,

To solve the optimization problem, we call:

```
prob.Solve();
```

5. **Catch any errors.**

with

```
| :? SolverPlatform.SolverException as ex -> printfn
"Exception: %s " ex.Message
```

6. **Obtain the solution status and final answer.** We obtain the solution status, along with the final objective and variable values and display the answer with multiple message boxes.

```
printfn ""
printfn "Nonlinear optimization example"
If model.Status <> optimal then
    printfn "Status: %A" model.Status
    printfn "Objective: %A" model.ObjectiveValue
    printfn "Variables: %A" model.Variables
```

7. **Build and run the application.** To compile and link this project, select Build | Build *UserGuideExample*. Then use Debug | Start Debugging to run the program. An output window will appear.

Complete Code

```
let nonlinearexample() =
    // Creates a new nonlinear model with two vars and two cons
    let model = new NonlinearModel(minimize, 2, 2)

    // The evaluation function.
    model.Evaluate <- fun x c ->
        c.[0] <- x.[0] + x.[1]
        c.[1] <- x.[0] * x.[1]
        x.[0] * x.[0] + x.[1] * x.[1]

    // The lower and upper bounds for the nonlinear constraints.
    model.ConstraintLower <- [ 1.; 0. ]
    model.ConstraintUpper <- [ 1.; inf ]
    // The lower and upper bounds for the variables.
    model.VariableLower <- [ 0.; 0. ]
    model.VariableUpper <- [ 1.; 1. ]
```

```

try
    model.Solve();
with
    | :? SolverPlatform.SolverException as ex -> printfn "There was
        an error solving the model: %s " ex.Message

printfn ""
printfn "Nonlinear optimization example"
if model.Status <> optimal then
    printfn "Status: %A" model.Status
printfn "Objective = %A" model.ObjectiveValue
printfn "Variables = %A" model.Variables

```

Creating a Simulation Model

1. **Basic first steps.** Follow the steps in the chapter, *Creating a New F# Project*. Adding the directive, *open SolverPlatform.FSharp.Distributions* is optional in F#. If you choose to forgo this directive, then you must add "SolverPlatform.FSharp" to the beginning of each SDK uncertain variable. For example:

```

prob.Distributions <- [ SolverPlatform.FSharp.Distributions.Uniform 0. 2. ;
    SolverPlatform.FSharp.Distributions.Normal 1. 1.]

```

In this example, we will name the function that will hold the linear example, *FSharpSimulationExample()*.

2. **Create a simulation problem.** The simulation model that we will be solving in this example contains two uncertain variables, X and Y. The uncertain variable X has a uniform distribution with a lower parameter of 0 and an upper parameter of 2. The uncertain variable Y has a Normal distribution with a mean and standard deviation equal to 1. In addition, there are two uncertain functions. The first uncertain function is the total of the two uncertain variables (X + Y). The second is the product of the two uncertain variables (X * Y).

```

X = PsiUniform(0,2)    and    Y = PsiNormal(1,1)

```

```

Function[1]: X + Y

```

```

Function[2]: X * Y

```

This example uses a correlation matrix which measures the degree to which the movements of the two uncertain variables are related. This is optional. The correlation matrix is:

1	.5
.5	1

The elements are being passed row-wise.

```

double[] correl = new double[] { 1.0, 0.5, 0.5, 1.0 };

```

First, we initialize the number of uncertain variables (*nvars*) and functions (*nfcns*).

```

let nvars = 2

```

```

let nfcns = 2

```

This next line of code (*let prob = new SimulationModel(nvars, nfcns)*) creates an instance of the Problem class and sets the problem type to "nonlinear" and automatically selects the Risk Solver to solve the model even if additional Large Scale "plug in" engines have been added. Note: This is the only simulation *problem* constructor available in the F# wrapper at this time.

The constructor also passes the number of variables (*nvars*) and the number of uncertain functions (*nfcns*). Note that we are utilizing the F# *let* statement which allows all properties/methods/functions under the *CProblem* class to be available for the remainder of the *Problem*.

```
let prob = new SimulationModel(nvars, nfcns)
```

Next, we pass the distributions of the two uncertain variables, Uniform and Normal.

```
prob.Distributions <- [ uniform 0. 2.; normal 1. 1. ]
```

Here, we create a new array of size *nvars* x *nvars* (2 x 2) to hold the correlation matrix. We are passing the *correl* array row-wise.

```
prob.Correlations <- [| 1.; 0.5; 0.5; 1.0 |]
```

- 3. Create the simulation evaluator.** When running a simulation, an evaluator *is* called so the user can calculate the uncertain functions, much like an evaluator was called to compute the objective function and constraints in the nonlinear optimization example.

We create and pass the evaluator (as described above) to the SDK. The “f” array is the uncertain function array and the “var” arguments are the uncertain variables. (In C++, C, VB.NET and VB6, the evaluator must appear in a separate class or module, however in F#, the evaluator resides in the same module as the rest of the model.)

We create the evaluator and pass it to the SDK.

```
prob.Sample <- fun var f ->
  f.[0] <- var.[0] + var.[1]
  f.[1] <- var.[0] * var.[1]
```

- 4. Run a simulation.** To run a simulation, we call:

```
prob.Simulate();
```

- 5. Catch any errors.**

with

```
| :? SolverPlatform.SolverException as ex -> printfn
"Exception: %s " ex.Message
```

- 6. Display various simulation statistics.** We obtain the simulation status and various uncertain variable and uncertain function statistics using message boxes.

```
printfn "Values = %A" prob.FunctionValues
printfn "Statistics on Uncertain Variables"
let stats = prob.VariableStatistics
for i = 0 to 1 do
  printfn "Variable %A" i
  printfn "Minimum = %A" stats.Minimum.[i]
  printfn "Maximum = %A" stats.Maximum.[i]
  printfn "Mean = %A" stats.Mean.[i]
  printfn "StdDev = %A" stats.StdDev.[i]
  printfn "Variance = %A" stats.Variance.[i]
  printfn "Skewness = %A" stats.Skewness.[i]
  printfn "Kurtosis = %A" stats.Kurtosis.[i]
  printfn ""
```

- 7. Build and run the application.** To compile and link this project, select Build | Build *NonlinearUserGuideExample*. Then use Debug | Start Debugging to run the program. An output window will appear.

[Complete Code](#)


```

let example3() =
    // Creates a new sim model with two unc vars and two unc fcns
    let model = new SimulationModel(2, 2)
    // Defines the prob distributions associated with each unc var
    model.Distributions <- [ uniform 0. 2.; normal 1. 1. ]
    // The correlation matrix between the variables, row-by-row.
    model.Correlations <- [| 1.; 0.5; 0.5; 1.0 |]
    // The sample function. Args are variables
    model.Sample <- fun var f ->
        f.[0] <- var.[0] + var.[1]
        f.[1] <- var.[0] * var.[1]
    try
        model.Simulate();
    with
        | :? SolverPlatform.SolverException as ex -> printfn "There
            was an error solving the model: %s " ex.Message

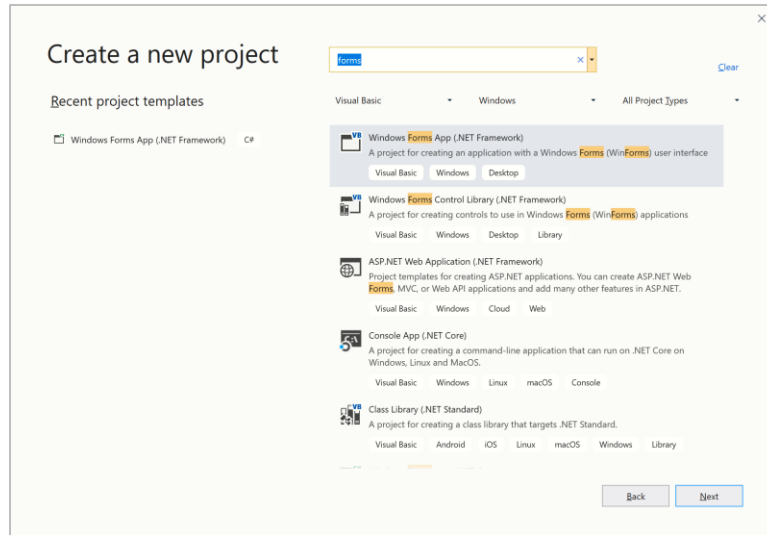
    printfn ""
    printfn "Simulation example"
    printfn "Values = %A" model.FunctionValues
    printfn "Statistics on Uncertain Variables"
    let stats = model.VariableStatistics
    for i = 0 to 1 do
        printfn "Variable %A" i
        printfn "Minimum   = %8.2f" stats.Minimum.[i]
        printfn "Maximum   = %8.2f" stats.Maximum.[i]
        printfn "Mean       = %8.2f" stats.Mean.[i]
        printfn "StdDev    = %8.2f" stats.StdDev.[i]
        printfn "Variance  = %8.2f" stats.Variance.[i]
        printfn "Skewness  = %8.2f" stats.Skewness.[i]
        printfn "Kurtosis  = %8.2f" stats.Kurtosis.[i]
    printfn ""

```

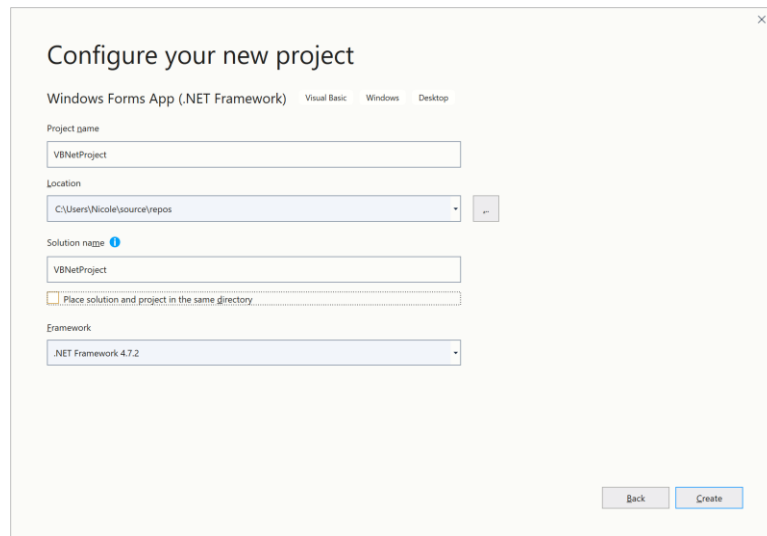
Creating a New VB.NET Project

First, follow the steps in the "Installation" chapter, which will install all required SDK files, including Example files, into the Frontline Systems directory located at C:\Program Files\Analytic Solver SDK\SolverSDK.

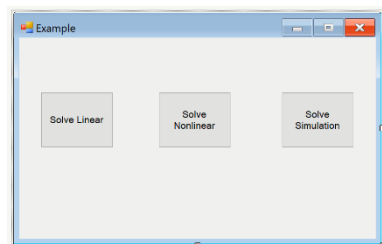
1. **Create a new project.** Start Microsoft Visual Studio, select File | New Project, select Visual Basic for Language and Windows Forms App (.NET Framework) for Template. Then click Next.



Type a name of your choice in the Project name field, and then click OK. This example uses the name VBNETProject.



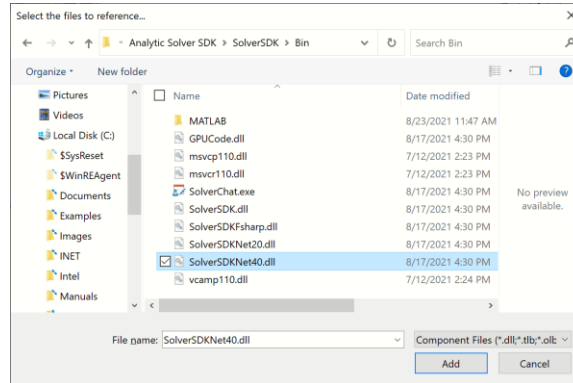
A new form will open. Construct a form of your choice using the Visual Studio Toolbox. In this example, the simple form shown below will be referenced.



8. **Set a reference and add an (optional) directive.** In order to access the methods and properties in the Solver SDK, you must set a reference. The SDK itself is a C++ unmanaged code DLL, but we can set a reference to a "wrapper" DLL. During installation of the SDK, two .NET DLLs are installed into the Bin directory located at C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK: SolverSDKNET20.dll for NET 2.0/3.0/3.5 and SolverSDKNET40.dll for .NET4.0. After you set the appropriate reference (depending on your version of .NET), your C#

code will call the .NET DLL which in turn will call the SolverSDK.dll. At runtime, the SolverSDKNETXX.dll **must** be able to find the SolverSDK.dll. Simply copying the SolverSDK.dll into the same directory as the SolverSDKNETXX.dll will ensure that the SolverSDK.dll is found.

To add a reference, click **References** in the Solution Explorer and select **Add Reference**. Select **Browse**, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Bin, select the appropriate **SolverSDKNetXX.dll** and press **Add**.



Adding the directive, *Import SolverPlatform*, is optional in VB.NET. If you choose to forgo the directive, then you must add "SolverPlatform" to the beginning of each SDK function call. For example:

```
using (SolverPlatform.Problem prob = new
SolverPlatform.Problem())

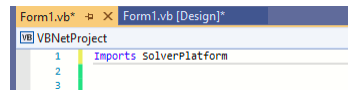
SolverPlatform.DoubleMatrix objcoeff = new
SolverPlatform.DoubleMatrix(1, nvars);

prob.Solver.SolverType = SolverPlatform.Solver_Type.Maximize;

prob.Model.AllLinear[SolverPlatform.Function_Type.Objective] =
objcoeff;

prob.Model.AllLinear[SolverPlatform.Function_Type.Constraint] =
concoeff;
```

This example uses the *Imports SolverPlatform* directive which is shown in the screenshot below.



From here you can solve either a linear, nonlinear, or simulation model. For more details on each, please read on.

Creating a Linear Model

1. **First steps.** Follow the steps in the section, *Creating a New C# Project*, and then double click on the *SOLVE LINEAR* button. Doing so will open the *buttonLinear_Click* event which is where we will write our code to solve the linear optimization problem.
2. **Create the linear model.** In this section, we will solve a linear optimization problem with two variables (x1 and x2) and three constraints. Both final variable values must be zero or larger.

```
Maximize 2 x1 + 3 x2
Subject To:
9 x1 + 6 x2 <= 54
```

```

6 x1 + 7 x2 <= 42
5 x1 + 10 x2 <= 50
x1, x2 non-negative

```

To begin, the example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model.

```

Dim nvars as Integer, ncons as Integer
nvars = 2 '2 decision variables
ncons = 3 '3 constraints

```

The next few lines of code set up the objective and constraint coefficient matrices using two separate double matrices: *objcoeff* – for the objective coefficients and *concoeff* – for the constraint coefficients. Since there is only one objective, we pass the double matrix, *objcoeff*, of size *nvars* (number of variables) x 1 for the objective coefficients and the second double matrix, *concoeff*, of size *nvars* x *ncons* (number of constraints) for the constraint coefficients. Notice that we are using *NZBgn()* and *NZEnd()* when passing the constraint matrix (*concoeff*) as explained in the subsection *Passing Sparse Matrices* in the chapter *Designing Your Application*. As a result, the Solver SDK will not update the *concoeff* DoubleMatrix object until *NZEnd()* is executed.

```

Dim objCoeff as new DoubleMatrix
With objCoeff
.Create(Rows:=1, Columns:=2)
.Value(0,0) = 2
.Value(0,1) = 3
End With

Dim conCoeff as new DoubleMatrix
With conCoeff
.Create(Rows:=3, Columns:=2)
.Value(0,0) = 9
.Value(0,1) = 6
.Value(1,0) = 6
.Value(1,1) = 7
.Value(2,0) = 5
.Value(2,1) = 10
End With

```

Next, we create an array to hold the upper bounds of the constraint right hand sides.

```

Dim ub_constraints(2) as Double
ub_constraints(0) = 54
ub_constraints(1) = 42
ub_constraints(2) = 50

```

The first line of code inside the Try event creates a new Problem, *prob*, and sets it to the null reference of the variable type Problem.

```

Dim prob as Problem = Nothing

```

This next line of code (*using (Problem prob = new Problem())*) creates an instance of the CProblem class. There are two different constructors which create a *Problem*. The first

constructor (*Problem prob = new Problem()*) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call *prob.VarDecision.Size* to automatically add a variable block or create a variable block using *Variable vars(Variable_Type_Decision, "vars", 1)* where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add constraints automatically using *prob.FcnConstraint.Size* or create a function with *Function con1(Function_Type_Constraint, "con", 1)* (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling *prob.Functions.Add(con)* to add the function to the Problem.

The second constructor, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type.Minimize*, *Solver_Type.Maximize* or *Solver_Type.Simulate*) the number of variables (*nvars*) and the number of constraints (*ncons*). An example of the second constructor is: *using (Problem problem = new Problem(Solver_Type.Maximize, nvars, ncons))*. Note that we are utilizing the C# *using* statement which allows all properties/methods/functions under the *CProblem* class to be available for the remainder of the *Problem*.

```
prob = New SolverPlatform.Problem
```

These two lines of code pass the number of decision variables (*prob.VarDecision.Size = nvars*) and constraints (*prob.FcnConstraint.Size = ncons*) that are included in the model.

```
prob.VarDecision.Size = nvars;
prob.FcnConstraint.Size = ncons;
```

As mentioned above, we could have added the variables using the two lines of code below. The first line of code creates a variable block of type *Variable_Type_Decision*, names the block "vars" and then sets the size of the block to 2. The second line of code adds the *vars* variable block to the Problem.

```
Variable vars (Variable_Type_Decision, "vars", 2);
prob.Variables.Add(vars);
```

In addition, we could have added the constraints using the two lines of code below. The first line of code creates a function block of type *Variable_Type_Constraint*, names the block "cons" and then sets the size of the block to 3. The second line of code adds the *cons* function block to the Problem.

```
Function cons (Function_Type_Constraint, "cons", 3);
prob.Functions.Add(cons);
```

We pass the *ub_constraints* array to the SDK. Recall that this array holds the upper bounds of the constraint right hand sides.

```
prob.FcnConstraint.UpperBound.Array = ub_constraints;
```

To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
prob.VarDecision.NonNegative();
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass upper and lower variable bounds to the SDK.

To tell the SDK to maximize the objective function, we use the line of code below. Alternatively, *Solver_Type.Minimize* will result in the model being minimized.

```
prob.Solver.SolverType = Solver_Type.Maximize;
```

3. **Call AllLinear functions and solve the linear model.** The line of code, *prob.Model.AllLinear[Function_Type.Objective, Variable_Type.Decisions]*, passes the objective coefficient matrix and *prob.Model.AllLinear[Function_Type.Constraint, Variable_Type.Decision]* passes the constraint coefficient matrix created and initialized above. Calling *prob.Model.AllLinear* sets the problem type to "linear" (Problem_Type.OptLP) and will automatically select the best linear engine to solve the problem. If no additional Large Sclae "plug in" engines have been added, then the LP/Quadratic engine will be selected.

```
prob.Model.AllLinear(Function_Type.Objective,
Variable_Type.Decisions) = objCoeff;
prob.Model.AllLinear(Function_Type.Constraint,
Variable_Type.Decision) = conCoeff;
```

Solve the optimization model by calling:

```
problem.Solver.Optimize();
```

4. **Obtain and display the solution to the linear model.** We obtain the solution status along with the final objective, constraint and decision variable values,

```
Dim status as Optimize_Status
status = prob.Solver.Optimization 'optimization status
Dim varvalue() as Double
Varvalue=prob.VarDecision.FinalValue.Array
Dim conValue() as Double
conValue = prob.FcnConstraint.FinalValue.Array
Dim objValue() as Double
objValue = prob.FcnObjective.FinalValue.Array
```

and use message boxes to display the answer.

```
MessageBox.Show("Optimize Status = " & status)
MessageBox.Show("x1 = " & varvalue(0) & ", x2 = " & varvalue(1))
MessageBox.Show("Constraint 1 = " & conValue(0) & ", Constraint 2
= " & conValue(1) & ", Constraint 3 = " & conValue(2))
MessageBox.Show("Obj = " & objValue(0))
```

5. **Catch any errors.** The catch statement handles any exceptions.

```
Catch ex As SolverException
    MsgBox ("Exception " + ex.ExceptionType & ex.Message,
    MsgBoxStyle.Critical, "Exception")
End Try
```

6. **Build and run the application.** To compile and link this project, select Build | Build CSharpNETProject. Then use Debug | Start Debugging to run the program. Press the command button **Solve Linear** to solve the linear optimization example.

7. **Dispose of Problem.** Use prob.Dispose to dispose of the problem.

```
If Not prob is Nothing Then
    prob.Dispose()
End If
```

Full Code

```

Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
Button1.Click
'    Example 1: Solve the LPmodel:
'    Maximize    2 x1 +  3 x2
'    Subj to     9 x1 +  6 x2 <= 54
'               6 x1 +  7 x2 <= 42
'               5 x1 + 10 x2 <= 50
'    x1, x2 non-negative
'    LP solution: x1 = 2.8, x2 = 3.6
'    Objective = 16.4
'
'    Matrix is passed rowwise, dense.

Dim nvars As Integer, ncons As Integer
nvars = 2 '2 decision variables
ncons = 3 '3 constraint functions

Dim objCoeff As New DoubleMatrix
With objCoeff
.Create(Rows:=1, Columns:=2)
.Value(0, 0) = 2
.Value(0, 1) = 3
End With

Dim conCoeff As New DoubleMatrix
With conCoeff
.Create(Rows:=3, Columns:=2)
.Value(0, 0) = 9
.Value(0, 1) = 6
.Value(1, 0) = 6
.Value(1, 1) = 7
.Value(2, 0) = 5
.Value(2, 1) = 10
End With

Dim ub_constraints(2) As Double
ub_constraints(0) = 54
ub_constraints(1) = 42
ub_constraints(2) = 50

Try
Dim prob As SolverPlatform.Problem = Nothing
prob = New SolverPlatform.Problem ' create a problem

prob.VarDecision.Size = nvars ' 2 variables
prob.FcnConstraint.Size = ncons ' 3 constraints

prob.FcnConstraint.UpperBound.Array = ub_constraints

prob.VarDecision.NonNegative() 'variables >= 0

'linear model; we pass the coefficients for the objective
function(s)
prob.Model.AllLinear(Function_Type.Objective,
Variable_Type.Decision) = objCoeff

```

```

'linear model; we pass the coefficients for the constraint
function(s)
prob.Model.AllLinear(Function_Type.Constraint,
Variable_Type.Decision) = conCoeff

prob.Solver.SolverType = Solver_Type.Maximize

prob.Solver.Optimize() 'solve model

Dim status As Optimize_Status
status = prob.Solver.OptimizeStatus 'optimization status

Dim varvalue() As Double
varvalue = prob.VarDecision.FinalValue.Array 'final variable
values

Dim conValue() As Double
conValue = prob.FcnConstraint.FinalValue.Array

Dim objValue() As Double
objValue = prob.FcnObjective.FinalValue.Array

'Note how the use of 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 1 block of each.

MessageBox.Show("Optimize Status = " & status & vbCrLf)
MessageBox.Show("x1 = " & varvalue(0) & ", x2 = " & varvalue(1))
MessageBox.Show("Constraint 1 = " & conValue(0) & ", Constraint 2 =
" & conValue(1) & ", Constraint 3 = " & conValue(2))
MessageBox.Show("Obj = " & objValue(0) & vbCrLf)

'Dispose of problem
If Not prob Is Nothing Then
    prob.Dispose()
End If
objCoeff = Nothing
conCoeff = Nothing
ub_constraints = Nothing

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

End Sub

```

Creating a Nonlinear Problem

1. **Basic first steps.** Follow the steps above in the section, *Creating a New C# Project*, and then double click on the *SOLVE NONLINEAR* button. Doing so will open the *buttonNonlinear_Click* event which is where we will write our code to solve the nonlinear optimization problem.
2. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constant (x + y) forces the total of the

two variables to equal to 1 and the second constraint ($x * y$) forces the product of the two variables to be greater than zero. Both final variable values must be zero or greater.

```
Minimize x^2 + y^2
Subject to:
x + y = 1
x * y >= 0
x, y are non-negative
```

First, we initialize the number of variables (*nvars*) and the number of constraints (*ncons*).

```
int nvars = 2, ncons = 2;
```

Then, we create two arrays to hold the variable lower bounds (lb array) and the variable upper bounds (ub array).

```
Dim lb(1) As Double
lb(0) = 0
lb(1) = 0
Dim ub(1) As Double
ub(0) = 1
ub(1) = 1
```

Next, we create two more arrays to hold the upper (ub_constraints) and lower bounds (lb_constraints) on the constraint right hand sides. Notice that the upper and lower bound of the first constraint is 1 to denote an equality constraint. $1.0E+30$ sets the upper bound of the second constraint to positive infinity.

```
Dim ub_constraints(1) as Double
ub_constraints(0) = 1.0#
ub_constraints(1) = 1.0E+30
Dim lb_constraints(1) as Double
lb_constraints(0) = 1.0#
lb_constraints(1) = 0
```

The first line of code inside the Try event creates a new Problem, prob, and sets it to the null reference of the variable type Problem.

```
Dim prob as Problem = Nothing
```

This next line of code (*Problem prob = new problem(Solver_Type.Minimize, nvars, ncons))* creates an instance of the Problem class. There are two different constructors which create a Problem. The first constructor (*Problem prob = new Problem()*) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call *prob.VarDecision.Size* to automatically add a variable block or create a variable block using *CVariable vars(Variable_Type_Decision, "vars", 1)* where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add constraints automatically using *prob.FcnConstraint.Size* or create a function with *Function con1(Function_Type_Constraint, "con", 1)* (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling *prob.Functions.Add(con)* to add the function to the Problem.

The second constructor, as shown here, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type.Minimize*, *Solver_Type.Maximize* or *Solver_Type.Simulate*) the number of variables (*nvars*) and the number of constraints (*ncons*).

```
Dim prob as Problem = Nothing
prob = new Problem(Solver_Type.Minimize, nvars, ncons)
```

Here we pass the constraint upper and lower bounds to the SDK,

```
prob.FcnConstraint.UpperBound.Array = ub_constraints;
prob.FcnConstraint.LowerBound.Array = lb_constraints;
```

following up with the variable upper and lower bounds.

```
prob.VarDecision.UpperBound.Array = ub;
prob.VarDecision.LowerBound.Array = lb;
```

3. **Create the nonlinear evaluator.** In a linear model or quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator (*Eval_Type.Function*) to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied. (It is possible to pass these derivatives directly to the SDK using an evaluator of type *Eval_Type.Gradient*. For more information see the subheading, *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter, *Solver SDK Evaluators*.)

We create and pass the evaluator (as described above) to the SDK.

```
NLPEvaluator = New Evaluator
prob.Evaluators.Item(Eval_Type.Function) = NLPEvaluator
```

We will write the code for our *eval_type.function* evaluator inside of this event handler.

4. **Write the nonlinear evaluator.** Inside of our *Nonlinear_OnEvaluate* event handler, we first dimension, *NLPeval*, as a *problem* and then set *NLPeval* as an *evaluator*.

```
Dim NLPeval as Problem
NLPeval = evaluator.Problem
```

Dimension a new array *Var* (as a double) to hold the variable values being passed between the SDK and the evaluator.

```
Dim evalVar() as Double
evalVar = NLPeval.VarDecision.Value.Array
```

Then we store the computed constraint values in an array, *evalFcn*.

```
Dim evalFcn(1) as Double
evalFcn(0) = evalVar(0) + evalVar(1)
evalFcn(1) = evalVar(0) * evalVar(1)
```

Then we compute the objective function, $x * x + y * y$.

```
Dim evalObj as Double
evalObj = evalVar[0] * evalVar[0] + evalVar[1] * evalVar[1];
```

Afterwards, we must pass the function values and the objective value back to Solver SDK.

```
NLPeval.FcnConstraint.Value.Array = evalFcn
NLPeval.FcnObjective.Value(0) = evalObj
```

Finally, we tell the SDK to continue optimizing by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action.Evaluate* and *Engine_Action..Stop*. *Engine_Action.Evaluate* forces the SDK to call the evaluator of type *Eval_Type.Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action.Stop* will indicate to the SDK that an error has occurred during the calculation of the functions (objective or constraints). The Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine (for solving nonsmooth problems) will disregard the current point and continue solving.

```
return Engine_Action.Continue;
```

Our evaluator is complete. We will now finish the code inside of the *button_NonlinearClick* event.

5. **Solve the nonlinear problem.** Back inside the *button_NonlinearClick* event, we provide the problem type as nonlinear (*Problem_Type_OptNLP*) which directs the SDK to select the GRG engine to solve the model. Alternatively, we also could have set the GRG engine manually (in either the SDK Platform or the SDK Pro) by calling:

```
prob.Engine = problem.Engines.GRGName.
```

```
prob.ProblemType = Problem_Type.OptNLP
```

To solve the optimization problem, we call:

```
prob.Solver.Optimize()
```

6. **Obtain the solution status and final answer.** We obtain the solution status, along with the final objective and variable values,

```
Optimize_Status status = prob.Solver.OptimizeStatus
```

```
double[] varValue = prob.VarDecision.FinalValue.Array
```

```
double[] conValue = prob.FcnConstraint.FinalValue.Array
```

```
double[] fcnValue = prob.FcnConstraint.FinalValue.Array;le[]
```

```
objValue = prob.FcnObjective.FinalValue.Array;
```

and display the answer with multiple message boxes.

```
MessageBox.Show("Status = " + status);
```

```
MessageBox.Show("x1 = " + varvalue[0] + ", x2 = " + valValue[1]);
```

```
MessageBox.Show("Constraint 1 = " + conValue[1] + ", Constraint 2  
= " + conValue[2]);
```

```
MessageBox.Show("Objective = " + objvalue[0]);
```

7. **Catch any errors.**

```
Catch ex as SolveException
```

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

8. **Build and run the application.** To compile and link this project, select Build | Build VBNETProject. Then use Debug | Start Debugging to run the program. Press the command button **Solve Nonlinear** to solve the nonlinear optimization example.

Full Code

```

Private Function NLPEvaluator_OnEvaluate(ByVal evaluator As
SolverPlatform.Evaluator) As SolverPlatform.Engine_Action Handles
NLPEvaluator.OnEvaluate

'This evaluator computes the values of the objective and
constraint functions for a given set of variable values.
    Dim NLPeval As Problem
    NLPeval = evaluator.Problem

    Dim evalVar() As Double

    ' First, we must ask what the current variable values are
    evalVar = NLPeval.VarDecision.Value.Array

    'We store the computed function values in an array, Fcn.
    Dim evalFcn(1) As Double
    evalFcn(0) = evalVar(0) + evalVar(1)
    evalFcn(1) = evalVar(0) * evalVar(1)

    Dim evalObj As Double
    ' We compute the objective
    evalObj = evalVar(0) * evalVar(0) + evalVar(1) * evalVar(1)

    'And we pass the function values to the Solver.
    NLPeval.FcnConstraint.Value.Array = evalFcn
    'And we pass the objective value to the Solver.
    NLPeval.FcnObjective.Value(0) = evalObj

    NLPEvaluator_OnEvaluate = Engine_Action.[Continue]

End Function

Private Sub buttonNonlinear_Click(sender As Object, e As
EventArgs) Handles buttonNonlinear.Click

' Solves the problem:
'
' Minimize  $X^2 + Y^2$ 
' Subject to:
'    $XY = 1$ 
'    $X * Y \geq 0$ 
'
' (Solution is  $X = Y = 0.5$ , Objective = 0.5)

Dim nvars = 2

```

```

Dim ncons = 2

Dim ub_constraints(1) As Double
ub_constraints(0) = 1.0#
ub_constraints(1) = 1.0E+30

Dim lb_constraints(1) As Double
lb_constraints(0) = 1.0#
lb_constraints(1) = 0

Dim lb(1) As Double
lb(0) = 0
lb(1) = 0

Dim ub(1) As Double
ub(0) = 1
ub(1) = 1

Dim start(1) As Double
start(0) = 0.5
start(1) = 0.5

Try
    Dim prob As SolverPlatform.Problem = Nothing
    prob = New Problem(Solver_Type.Minimize, nvars, ncons)

    prob.FcnConstraint.UpperBound.Array = ub_constraints
    prob.FcnConstraint.LowerBound.Array = lb_constraints

    prob.VarDecision.UpperBound.Array = ub
    prob.VarDecision.LowerBound.Array = lb
    prob.VarDecision.InitialValue.Array = start

    NLPEvaluator = New Evaluator
    prob.Evaluators.Item(Eval_Type.Function) = NLPEvaluator

    prob.ProblemType = Problem_Type.OptNLP
    prob.Solver.Optimize()

    Dim convalue() As Double
    convalue = prob.FcnConstraint.FinalValue.Array
    Dim varvalue() As Double
    varvalue = prob.VarDecision.FinalValue.Array
    Dim objValue As Double
    objValue = prob.FcnObjective.FinalValue(0)
    Dim status As Optimize_Status

    status = prob.Solver.OptimizeStatus
    MessageBox.Show("Optimize Status = " & status & vbCrLf)
    MessageBox.Show("x1 = " & varvalue(0) & ", x2 = " & varvalue(1))
    MessageBox.Show("Constraint 1 = " & convalue(0) & ", Constraint
    2 = " & convalue(1))
    MessageBox.Show("Objective = " & objValue)

    'dispose of the problem and the evaluator
    If Not prob Is Nothing Then
        prob.Dispose()
    End If
Catch
    'Error handling
End Try

```

```

        End If
        prob = Nothing
        NLPEvaluator = Nothing

Catch ex As SolverException

    MsgBox("Exception " & ex.ExceptionType & vbCrLf & ex.Message,
        MsgBoxStyle.Critical, "Exception")

End Try

End Sub

```

Creating a Simulation Model

1. **Basic first steps.** Follow the steps in the chapter, *Creating a New C# Project*, and then double click on the *SOLVE SIMULATION* button. Doing so will open the *buttonSimulate_Click* event which is where we will write our code to solve the simulation problem.
9. **Create a simulation problem.** The simulation model that we will be solving in this example contains two uncertain variables, X and Y. The uncertain variable X has a uniform distribution with a lower parameter of 0 and an upper parameter of 2. The uncertain variable Y has a Normal distribution with a mean and standard deviation equal to 1. In addition, there are two uncertain functions. The first uncertain function is the total of the two uncertain variables (X + Y). The second is the product of the two uncertain variables (X * Y).

```
X = PsiUniform(0,2)   and   Y = PsiNormal(1,1)
```

```
Function[1]: X + Y
```

```
Function[2]: X * Y
```

First, we initialize the number of uncertain variables (*nuncvars*) and functions (*nuncfcns*).

```
Dim nuncvars as Integer = 2
```

```
Dim nuncfcns as Integer = 2
```

This example uses a correlation matrix which measures the degree to which the movements of the two uncertain variables are related. This is optional. The correlation matrix is:

1	.5
.5	1

The elements are being passed row-wise.

```
Dim correl as Double = new double() { 1.0, 0.5, 0.5, 1.0 };
```

The first line of code inside the Try event creates a new Problem, prob, and sets it to the null reference of the variable type Problem.

```
Dim prob as Problem = Nothing
```

The next line of code, *prob = New Problem(Solver_Type.Simulate, nuncvars, nuncfcns)*, creates an instance of the Problem class. There are two different constructors which create a Problem. The first constructor (*Problem prob = new Problem()*) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call *prob.VarUncertain.Size* to automatically add a variable block or create a variable block using *Variable vars(Variable_Type_Uncertain, "vars", 1)* where the first argument is the variable type, the 2nd

argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add uncertain functions automatically using *prob.FcnConstraint.Size* or create a function with *Function con1(Function_Type_Uncertain, "con", 1)* (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling *prob.Functions.Add(con)* to add the function to the Problem.

The second constructor, as shown here, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type.Minimize*, *Solver_Type.Maximize* or *Solver_Type.Simulate*) the number of uncertain variables (*nuncvars*) and the number of uncertain functions (*nuncfcns*).

```
prob = New Problem(Solver_Type.Simulate, nuncvars, nuncfcns)
```

Next, we pass the distributions of the two uncertain variables, Uniform and Normal.

```
prob.Model.VarDistribution[0] = new
Distribution("PsiUniform(0,2)");

problem.Model.VarDistribution[1] = new
Distribution("PsiNormal(1,1)");
```

Here, we create a new double matrix of size *nvars* x *nvars* (2 x 2) to hold the correlation matrix. We are passing the *correl* array row-wise.

```
prob.Model.DistCorrelations = new DoubleMatrix(Array_Order.ByRow,
nuncvars, nuncvars, correl);
```

10. **Create the simulation evaluator.** When running a simulation, the evaluator *Eval_Type_Sample* is called so the user can calculate the uncertain functions, much like *Eval_Type_Function* is called to compute the objective function and constraints in a nonlinear optimization problem.

We create the *Eval_Type.Sample* evaluator (as described above) and pass the evaluator to the SDK.

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

We will write the code for our *eval_type.function* evaluator inside of this event handler.

11. **Complete the simulation evaluator.** The first line in the simulation evaluator asks for the variable values from Solver SDK.

```
Dim evalUncVar() As Double =
evaluator.Problem.VarUncertain.Value.Array
```

Then, the uncertain function values are computed given the current variable values from the line above.

```
Dim evalUncFcn() As Double =
evaluator.Problem.FcnUncertain.Value.Array

evalUncFcn(0) = evalUncVar(0) + evalUncVar(1)    ' X+Y
evalUncFcn(1) = evalUncVar(0) * evalUncVar(1)    ' X*Y
```

Finally, we tell the SDK to continue simulating.

```
return Engine_Action.Continue;
```

Finally, we tell the SDK to continue simulating by returning *Engine_Action.Continue*. There are two additional *Engine_Action* functions, *Engine_Action.Evaluate* and *Engine_Action.Stop*. *Engine_Action.Evaluate* forces the SDK to call the evaluator of type *Eval_Type.Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action.Stop* will indicate to the SDK that an error has occurred during the calculation of the uncertain functions. The SDK will mark the current trial as an error trial and continue with the remaining trials.

Our evaluator is complete. We will now finish the code inside of the *button_SimulationClick* event.

12. **Solve the simulation problem.** Back inside the *button_SimulationClick* event, we call the following line of code to run the simulation.

```
prob.Solver.Simulate();
```

13. **Display various simulation statistics.** We obtain the simulation status and various uncertain variable and uncertain function statistics using message boxes.

```
MessageBox.Show("Status = " & prob.Solver.SimulateStatus & vbCrLf)
MessageBox.Show("Statistics on Uncertain Variables" & vbCrLf)
```

```
Dim i As Integer
For i = 0 To nuncvars - 1
    MessageBox.Show("Variable " & i & vbCrLf)
    MessageBox.Show("Minimum = " &
        prob.VarUncertain.Statistics.Minimum(i) & vbCrLf)
    MessageBox.Show("Maximum = " &
        prob.VarUncertain.Statistics.Maximum(i) & vbCrLf)
    MessageBox.Show("Mean = " &
        prob.VarUncertain.Statistics.Mean(i) & vbCrLf)
    MessageBox.Show("StdDev = " &
        prob.VarUncertain.Statistics.StdDev(i) & vbCrLf)
    MessageBox.Show("Variance = " &
        prob.VarUncertain.Statistics.Variance(i) & vbCrLf)
    MessageBox.Show("Skewness = " &
        prob.VarUncertain.Statistics.Skewness(i) & vbCrLf)
    MessageBox.Show("Kurtosis = " &
        prob.VarUncertain.Statistics.Kurtosis(i) & vbCrLf)
    MessageBox.Show("10th Percentile = " &
        prob.VarUncertain.Percentiles(i, 10) & vbCrLf)
    MessageBox.Show("90th Percentile = " &
        prob.VarUncertain.Percentiles(i, 90) & vbCrLf)
```

```
Next
```

```
MessageBox.Show(vbCrLf & "Statistics on Uncertain Functions" & vbCrLf)
```

```
For i = 0 To nuncfcns - 1
    MessageBox.Show("Function " & i & vbCrLf)
    MessageBox.Show("Minimum = " &
        prob.FcnUncertain.Statistics.Minimum(i) & vbCrLf)
    MessageBox.Show("Maximum = " & \
        prob.FcnUncertain.Statistics.Maximum(i) & vbCrLf)
    MessageBox.Show("Mean = " &
        prob.FcnUncertain.Statistics.Mean(i) & vbCrLf)
    MessageBox.Show("StdDev = " &
        prob.FcnUncertain.Statistics.StdDev(i) & vbCrLf)
    MessageBox.Show("Variance = " &
        prob.FcnUncertain.Statistics.Variance(i) & vbCrLf)
    MessageBox.Show("Skewness = " &
        prob.FcnUncertain.Statistics.Skewness(i) & vbCrLf)
    MessageBox.Show("Kurtosis = " &
        prob.FcnUncertain.Statistics.Kurtosis(i) & vbCrLf)
    MessageBox.Show("10th Percentile = " &
```



```

        prob.FcnUncertain.Percentiles(i, 10) & vbCrLf)
        MsgBox.Show("90th Percentile = " &
        prob.FcnUncertain.Percentiles(i, 90) & vbCrLf)
    Next

```

14. **Catch any errors.** The catch statement will handle any exceptions.

```

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

```

15. **Build and run the application.** To compile and link this project, select Build | Build *VBNetProject*. Then use Debug | Start Debugging to run the program. Press the command button **Solve Simulation** to solve the simulation example.

Full Code

```

Private Function SimEvaluator_OnEvaluate(ByVal evaluator As
SolverPlatform.Evaluator) As SolverPlatform.Engine_Action Handles
SimEvaluator.OnEvaluate

    ' First, we ask for the current uncertain variable values
    Dim evalUncVar() As Double =
        evaluator.Problem.VarUncertain.Value.Array

    'We compute the function values given the current variable
    values
    Dim evalUncFcn() As Double =
        evaluator.Problem.FcnUncertain.Value.Array
    evalUncFcn(0) = evalUncVar(0) + evalUncVar(1)    ' X+Y
    evalUncFcn(1) = evalUncVar(0) * evalUncVar(1)    ' X*Y

    SimEvaluator_OnEvaluate = Engine_Action.[Continue]

End Function

Private Sub Button4_Click(sender As Object, e As EventArgs)
Handles Button4.Click

    Dim nuncvars As Integer = 2
    Dim nuncfcns As Integer = 2
    Dim correl() As Double = New Double() {1.0, 0.5, 0.5, 1.0}

    Try
        Dim prob As SolverPlatform.Problem = Nothing
        prob = New Problem(Solver_Type.Simulate, nuncvars, nuncfcns)

        prob.Model.VarDistribution(0) = New
        Distribution("PsiUniform(0,2)")
        prob.Model.VarDistribution(1) = New
        Distribution("PsiNormal(1,1)")

        prob.Model.DistCorrelations = New
        DoubleMatrix(Array_Order.ByRow, nuncvars, nuncvars, correl)

        SimEvaluator = New Evaluator
        prob.Evaluators(Eval_Type.Sample) = SimEvaluator

        prob.Solver.NumTrials = 10

        prob.Model.DistCorrelations = New

```

```

DoubleMatrix(Array_Order.ByRow, nuncvars, nuncvars, correl)

prob.Solver.Simulate()

MessageBox.Show("Status = " & prob.Solver.SimulateStatus)
MessageBox.Show("Statistics on Uncertain Variables")

Dim i As Integer
For i = 0 To nuncvars - 1
    MessageBox.Show("Variable " & i & vbCrLf)
    MessageBox.Show("Minimum = " &
        prob.VarUncertain.Statistics.Minimum(i) & vbCrLf)
    MessageBox.Show("Maximum = " &
        prob.VarUncertain.Statistics.Maximum(i) & vbCrLf)
    MessageBox.Show("Mean = " &
        prob.VarUncertain.Statistics.Mean(i) & vbCrLf)
    MessageBox.Show("StdDev = " &
        prob.VarUncertain.Statistics.StdDev(i) & vbCrLf)
    MessageBox.Show("Variance = " &
        prob.VarUncertain.Statistics.Variance(i) & vbCrLf)
    MessageBox.Show("Skewness = " &
        prob.VarUncertain.Statistics.Skewness(i) & vbCrLf)
    MessageBox.Show("Kurtosis = " &
        prob.VarUncertain.Statistics.Kurtosis(i) & vbCrLf)
    MessageBox.Show("10th Percentile = " &
        prob.VarUncertain.Percentiles(i, 10) & vbCrLf)
    MessageBox.Show("90th Percentile = " &
        prob.VarUncertain.Percentiles(i, 90) & vbCrLf)
Next
MessageBox.Show(vbCrLf & "Statistics on Uncertain Functions" &
vbCrLf)
For i = 0 To nuncfcns - 1
    MessageBox.Show("Function " & i & vbCrLf)
    MessageBox.Show("Minimum = " &
        prob.FcnUncertain.Statistics.Minimum(i) & vbCrLf)
    MessageBox.Show("Maximum = " &
        prob.FcnUncertain.Statistics.Maximum(i) & vbCrLf)
    MessageBox.Show("Mean = " &
        prob.FcnUncertain.Statistics.Mean(i) & vbCrLf)
    MessageBox.Show("StdDev = " &
        prob.FcnUncertain.Statistics.StdDev(i) & vbCrLf)
    MessageBox.Show("Variance = " &
        prob.FcnUncertain.Statistics.Variance(i) & vbCrLf)
    MessageBox.Show("Skewness = " &
        prob.FcnUncertain.Statistics.Skewness(i) & vbCrLf)
    MessageBox.Show("Kurtosis = " &
        prob.FcnUncertain.Statistics.Kurtosis(i) & vbCrLf)
    MessageBox.Show("10th Percentile = " &
        prob.FcnUncertain.Percentiles(i, 10) & vbCrLf)
    MessageBox.Show("90th Percentile = " &
        prob.FcnUncertain.Percentiles(i, 90) & vbCrLf)
Next

'dispose of problem and evaluator
If Not prob Is Nothing Then
    prob.Dispose()
End If

SimEvaluator = Nothing

```

```
    Catch ex As SolverException

        MsgBox("Exception " & ex.ExceptionType & vbCrLf & ex.Message,
            MsgBoxStyle.Critical, "Exception")

    End Try

End Sub
```

Calling the Solver SDK from JAVA

Introduction

This chapter details the steps required to write and run a simple linear, nonlinear, and simulation example using Solver SDK and 32/64-bit JAVA. Examples that illustrate how to call the SDK using the JAVA wrapper may be opened from a JAVA IDE, (i.e., JavaBeans or Eclipse), or using a script editor such as Notepad++.

The JAVA interpreter reads and parses the example script, compiles it into bytecode, and executes in run-time. To work with Solver SDK in JAVA, you must have the JAVA SE Development Kit installed. It is free, and it may be downloaded from the Oracle website (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>).

To run the JAVA Solver SDK Examples, follow the steps in the Installation chapter to install all required SDK files, including Example files, into the Frontline Systems directory located at C:\Program Files.

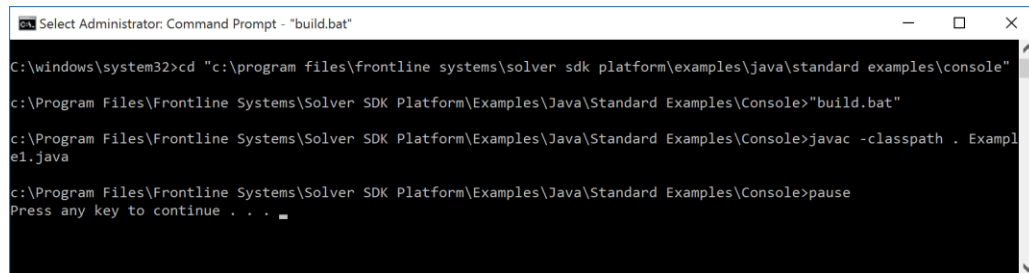
To open example code for JAVA, browse to

- C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\JAVA

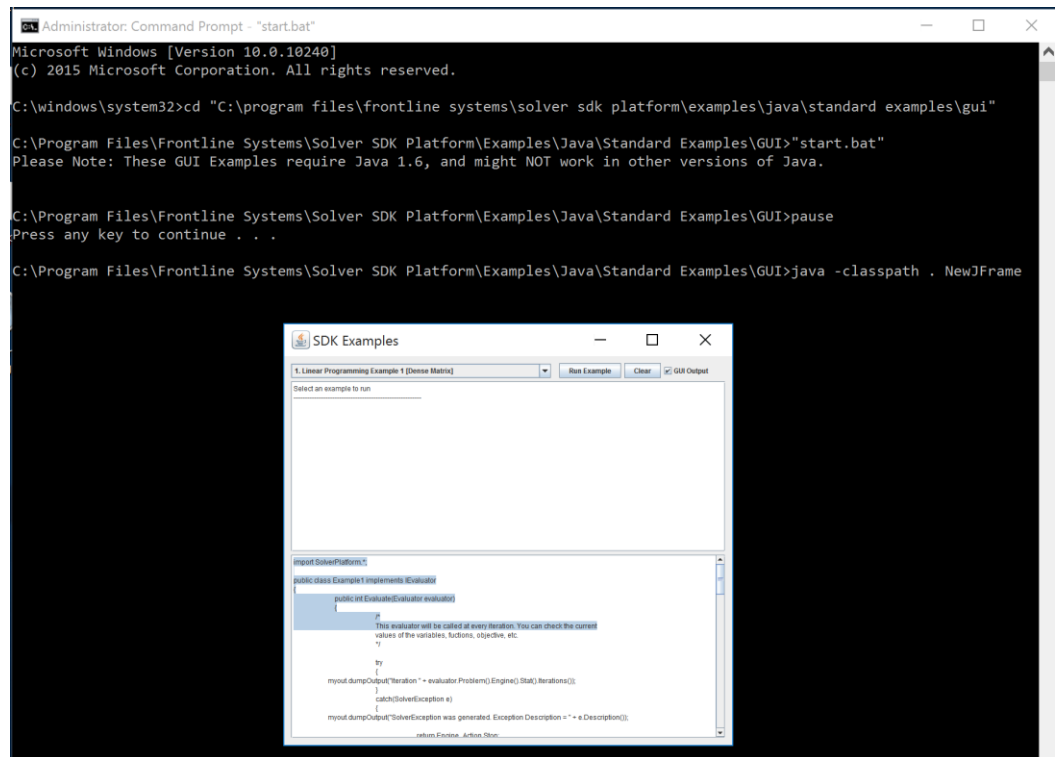
Over 30 files with the name **ExamplesXX.java** are installed into ...Analytic Solver SDK\SolverSDK\Examples\Java\Standard Examples containing definitions for functions that illustrate sample usage of Solver SDK. In addition, application specific examples such as a simulation model simulating saving money for college, a product mix example, a cutting stock example and a traveling salesman example are installed into ...Analytic Solver SDK\Solver SDK\Java\Examples

There are two ways to view and run the JAVA examples: by command prompt executing sequential commands, or by running the JAVA GUI.

Command Prompt—open a Command Prompt (right-click and select **Run as Administrator**), navigate to the Java/Console directory and type "build.bat" at the prompt. Using this method, single commands are executed in succession.



Graphical User Interface—open a Command Prompt (right click and select **Run as Administrator**), navigate to the Java/GUI directory, and type "start.bat" at the prompt. The following dialog opens. Select the desired example using the drop-down menu at the top, then click **Run Example** to run the example, or **Clear** to delete the contents of the window.



To call Solver SDK from JAVA, you will need **SolverSDK.jar** and **SolverSDK.dll** along with any other binaries that **SolverSDK.dll** depends upon, which are installed into the **Bin** folder (see above for complete path). The path to the **SolverSDK dynamic libraries** is added to the system PATH during installation.

Creating a Linear Model

1. **Installation.** Follow the steps in the Installation chapter to install all required SDK files, including Example files, into the Frontline Systems directory located at C:\Program Files.
2. **Example Step – Thru.** This section steps through Example1.java. This file is installed into ...\\Frontline Systems\\Analytic Solver SDK\\SolverSDK\\Examples\\Java\\StandardExamples\\. The first line must begin with an *import* statement to allow JAVA to access the SolverSDK.dll.

```
Import SolverPlatform.*;
```
3. **Create the linear model.** In this section, we will solve a linear optimization problem with two variables (x1 and x2) and three constraints. Both final variable values must be integers and zero or larger.

```
Maximize 2 x1 + 3 x2
Subject To:
9 x1 + 6 x2 <= 54
6 x1 + 7 x2 <= 42
5 x1 + 10 x2 <= 50
x1, x2 non-negative integers
```
4. **Initialize variables and arrays.** The example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model. In addition, three arrays are created to hold the objective coefficients, constraint coefficients, and constraint right sides. Note that the constraint coefficients are being passed row-wise.

```

int nvars = 2;
int ncons = 3;
double [] objcoeff = new double []{ 2, 3 };
double [] concoeff = new double []{ 9, 6, 6, 7, 5, 10 };
double [] rhsvalue = new double []{ 54, 42, 50 };

```

5. **Create a Problem.** There are two different constructors that create a Problem. The first constructor (*problem = new Problem()*) shown here, will only create the problem. This constructor will not create any variable or function blocks, nor will it pass any additional information such as whether you are solving a minimization optimization, maximization optimization, or simulation problem.

If this constructor is used, a user must either call *problem.VarDecision().Size(nvars)* to automatically add a variable block or create a variable block using *Variable decVars = Variable(Variable_Type.Decision, "Vars", 2)*, where the first argument is the variable type, the second argument is the name given to the variable block, and the third argument is the number of variables in the block. Additionally, the user must add the variable to the problem by calling *problem.Variables().Add(decVars)*. This is true for functions as well. A user can either add constraints automatically using *problem.FcnConstraint().Size(ncons)*, or create a function with *Function constraints = new Function(Function_Type.Constraint, "fcns", 3)*, then call *problem.Functions().Add(constraints)* to add the function to the Problem.

The second constructor, used in the following non-linear example, not only creates the Problem, but also passes the type of model to be solved, the number of variables (*nvars*) and the number of constraints (*ncons*).

```
problem = new Problem();
```

6. **Add a variable and function block.** Add a variable block containing two variables and a constraint block containing three functions to the Problem.

```

problem.VarDecision().Size(nvars);
problem.FcnConstraint().Size(ncons);

```

7. **Set the Problem "sense" to Maximize.** Use *Solver_Type.Minimize* if solving a minimization.

```
Problem.Solver().SolverType(Solver_Typ.Maximize);
```

8. **Assign the constraint right-hand sides and add the non-negativity constraint.** The first line of code below passes the constraint right-hand sides (54, 42, and 50) contained in the *rhsvalue* array.

```
problem.FcnConstraint().UpperBound(rhsvalue);
```

To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
Problem.VarDecision().NonNegative();
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass non-zero upper and lower variable bounds to the SDK.

9. **Call AllLinear functions.** In this example, AllLinear is used to pass the objective function and constraint coefficients. Doing so sets the problem type to "linear" which triggers Solver SDK to automatically select the best linear engine to solve the problem. If no additional Large Scale "plug in" engines have been added, the LP/Quadratic engine will be selected.

Since *problem.Model().AllLinear()* is expecting a Double Matrix, we must convert both the *concoeff* and *objcoeff* arrays to Double Matrices. For example, in the first line of code below, a new Double Matrix, containing a single row and nvars (2) columns, is created and populated with the objective function coefficients contained within the *objcoeff* array. Since Solver SDK only supports the use of one objective function per model, it does not matter if the values in the *objcoeff* array are passed row-wise or column-wise.

```
problem.Model().AllLinear(Function_Type.Objective, new
DoubleMatrix(Array_Order.ByRow, 1, nvars, objcoeff));
```

In the next line of code, a double matrix containing *ncons* (3) rows and *nvars* (2) columns, passes the constraint function coefficient values (using *Function_Type.Constraint*) contained in the *concoeff* array, row-wise, to the *AllLinear* function. If the *concoeff* array held the coefficient values passed by column, we would use *Array_Order.ByCol* as the first argument to *DoubleMatrix*.

```
problem.Model().AllLinear(Function_Type.Constraint, new
DoubleMatrix(Array_Order.ByRow, ncons, nvars, concoeff));
```

10. **Pass integer constraints.** Use *problem.VarDecision().IntegerType()* to add integer constraints to both variables.

```
problem.VarDecision().IntegerType( new int []{
Integer_Type.Integer, Integer_Type.Integer });
```

11. **Solve the optimization model.** Solve the optimization model by calling:

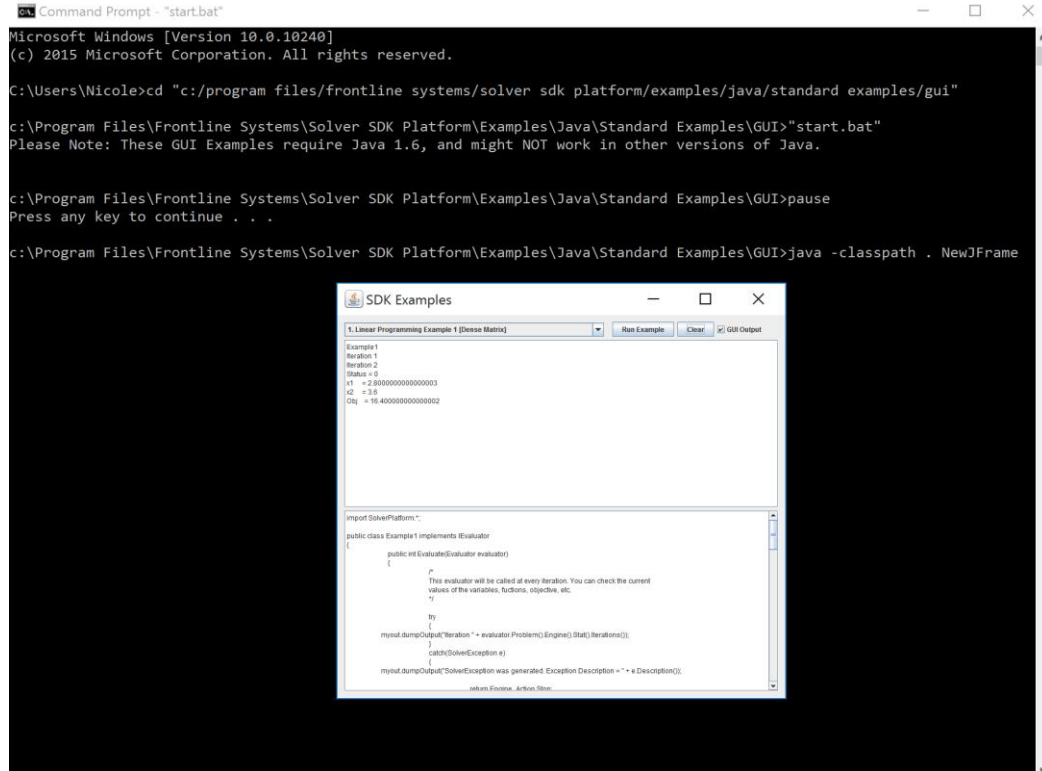
```
problem.Solver().Optimize();
```

12. **Obtain the status and final variable/constraint values.**

We capture the status of the optimization and final variable and constraint values, then display each using the *System.out.println* command. Note the use of the *VarDecision* and *FcnObjective* properties both point to the "current" variable block or objective block. In this example, there is only one block of each.

```
int nOptStatus = problem.Solver().OptimizeStatus();
System.out.println("Status = " + nOptStatus);
System.out.println("x1 = " +
problem.VarDecision().FinalValue(0));
System.out.println("x2 = " +
problem.VarDecision().FinalValue(1));
System.out.println("Obj = " +
problem.FcnObjective().FinalValue(0));
```

13. **Solve the example.** To solve this example using the GUI console, open a prompt (as an Administrator), navigate to the Java\GUI directory and type "start.bat" at the prompt. The following dialog opens. Select **Example 1** using the drop-down menu at the top, then click **Run Example**. To run the console example, see the Introduction to this chapter for instructions.



Creating a Nonlinear Model

1. **Installation.** Follow the steps in the Installation chapter to install all required SDK files, including Example files, into the Frontline Systems directory located at C:\Program Files.
2. **Example.** This section steps through **Example4.java**, which is installed into ...\\Frontline Systems\\Analytic Solver SDK\\SolverSDK\\Examples\\Java\\StandardExamples\\. The first line must begin with an *import* statement to allow Java to access the SolverSDK.dll.

```
Import SolverPlatform.*;
```

3. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constraint ($x + y$) forces the total of the two variables to be equal to 1, and the second constraint ($x * y$) forces the product of the two variables to be zero or larger. Both final variable values must be non-negative (0 or larger).

```
Maximize x^2 + y^2
Subject To:
    x + y = 1
    x * y >= 0
    x, y non-negative
```

4. **Initialize variables.** Initialize the number of variables ($nvars$) and constraints ($ncons$).

```
int nvars = 2;
int ncons = 2;
```

5. **Create arrays to hold the constraint right-hand sides.** Create two arrays of size $ncons$ (2 constraints) to hold the upper and lower bounds on the constraint right-hand sides. Notice the upper and lower bound of the first constraint is 1 to denote an equality constraint. **Constants.PINF** sets the upper bound of the second constraint to positive infinity.

```
double [] rhs = new double []{ 1.0, Constants.PINF() };
```



```
double [] lhs = new double []{ 1.0, 0 };
```

6. **Create arrays to hold variable upper and lower bounds.** The next two lines of code create two arrays to hold the lower and upper variable bounds, *lb* and *ub*, respectively.

```
double [] lb = new double []{ 0, 0 };
double [] ub = new double []{ 1, 1 };
```

7. **Create a Problem.** This next line of code creates an instance of the Problem class, passes the “sense” of the problem, the number of decision variables, and the number of constraints. For more information on creating a new Problem, see the step 5 explanation in the *Creating a Linear Model* section.

```
problem = new Problem(Solver_Type.Maximize, nvars, ncons);
```

8. **Pass the constraint right-hand sides to the Problem.** Pass the constraint upper (*ub_constraints*) and lower (*lb_constraints*) bounds to the SDK,

```
problem.FcnConstraint().UpperBound(rhs);
problem.FcnConstraint().LowerBound(lhs);
```

and follow up with the variable upper (*ub*) and lower (*lb*) bounds.

```
problem.VarDecision().UpperBound(ub);
problem.VarDecision().LowerBound(lb);
```

9. **Create the nonlinear evaluator.** In a linear model or quadratic model, the Solver SDK is supplied with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator to compute the function (objective and constraint) values. If a Jacobian evaluator is not supplied, this evaluator will be called at each iteration to compute the values of the objective and constraints. It is possible to pass these derivatives directly to the SDK using the *Eval_Type.Gradient* evaluator.

For more information on supplying this type of evaluator, see the subheading *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter *Solver SDK Evaluators*.

As described above, we create and pass the evaluator of type *Eval_Type.Function* to the SDK. The name of this evaluator is *NLP Evaluator*.

```
problem.Evaluators().Item(Eval_Type.Function).Evaluator(new
NLPEvaluator());
```

10. **Write the nonlinear evaluator.** Next, step into the nonlinear evaluator.

- a. Inside the evaluator *evalNLP*, a convenient pointer to the current values of the variables is created.

```
double [] Var = evaluator.Problem().VarDecision().Value();
```

- b. Next, calculate the constraints.

```
evaluator.Problem().FcnConstraint().Value(new Double []
{
    Var[0] + Var[1],
    Var[0] * Var[1],
});
```

- c. Next, compute the objective function, $x * x + y * y$.

```
evaluator.Problem().FcnObjective().Value( new double [] {
Var[0] * Var[0] + Var[1] * Var[1] });
```

The following line of code prints the current values of the variables and the iteration number each time the *NLPEvaluator* function is called (optional.)

```
System.out.println("Evaluation x1 = " + Var[0] + ", x2 = " +  
evaluator.Problem().Engine().Stat().FunctionEvals());
```

Finally, tell the SDK to continue optimizing by returning *Engine_Action.Continue*. There are two additional *Engine_Action* functions: *Engine_Action.Evaluate* and *Engine_Action.Stop*.

Engine_Action.Evaluate forces the SDK to call the evaluator of type *Eval_Type.Function* a second time, which can be useful if you need to inspect the variable values.

Engine_Action.Stop communicates to the SDK that an error has occurred during the calculation of the functions (objective or constraints).

Note: If an error does occur during the calculation of the functions, the Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine (for solving nonsmooth problems) will disregard the current point and continue solving.

```
return Engine_Action.Continue;
```

11. **Solve the nonlinear problem.** Back inside the **Example4** class, we provide the problem type as nonlinear, which directs the SDK to select the GRG engine to solve the model. Alternatively, we could have set the GRG engine manually (in either the SDK Platform or SDK Pro) by calling:
problem.Engine(problem.Engines().Item(Engine.LPName()));

```
problem.ProblemType(Problem_Type.OptNLP);
```

To solve the nonlinear optimization problem, call:

```
problem.Solver().Optimize();
```

12. **Obtain the solution status.** Obtain and display the solution status, along with the final objective and variable values. Note the use of the *VarDecision*, and *FcnObjective* properties that both point to the "current" variable or objective block. In this example, there is only one block of each.

```
int nOptStatus = problem.Solver().OptimizeStatus();  
double [] varvalue = problem.VarDecision().FinalValue();  
double [] objval = problem.FcnObjective().FinalValue();  
  
System.out.println("Status = " + nOptStatus);  
System.out.println("x1 = " + varvalue[0]);  
System.out.println("x2 = " + varvalue[1]);  
System.out.println("Obj = " + objval[0]);
```

Solve the example. To solve this example using the GUI console, open a command prompt (right-click and select Run as Administrator), navigate to the Java\GUI directory and type "start.bat" at the prompt. The following dialog opens. Select **Non-Linear Programming with GRG** using the drop down menu at the top, then click **Run Example**. To run the console example, see the Introduction to this chapter for instructions.

X	1	.5
Y	.5	1

The correlation coefficients in the *correl* array are being passed row-wise.

```
double [] correl = new double []{ 1.0, 0.5, 0.5, 1.0 };
```

6. **Create an instance of the Problem class.** Two different constructors create a *Problem*. See step 5 in *Creating a Linear Model* for details on both. This example uses the second constructor, which not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type.Minimize*, *Solver_Type.Maximize*, or *Solver_Type.Simulate*), the number of uncertain variables (*nvars*), and the number of functions (*nfcns*).

```
problem = new problem(Solver_Type.Simulate, nvars, nfcns);
```

7. **Set the number of trials.** The default value for the number of trials is 1000. In this example, we set the number of trials to 10.

```
problem.Solver().NumTrials(10);
```

8. **Set the distributions for the uncertain variables.** These two lines of code pass the distributions of the two uncertain variables, Uniform and Normal.

```
problem.Model().VarDistribution(0, new  
Distribution("PsiUniform(0,2)"));
```

```
problem.Model().VarDistribution(1, new  
Distribution("PsiNormal(1,1)"));
```

9. **Pass the Correlation Matrix.** Create a new double matrix of size *nvars* x *nvars* (2 x 2) to hold the correlation matrix. We are passing the *correl* array row-wise.

```
problem.Model().DistCorrelations(new  
DoubleMatrix(Array_Order.ByRow, nvars, nvars, correl));
```

10. **Create the simulation evaluator.** When running a simulation, the evaluator *Eval_Type.Sample* is called to calculate the uncertain functions, much like *Eval_Type.Function* is called to compute the objective function and constraints in a nonlinear optimization problem.

The evaluator, *Eval_Type.Sample*, is passed to the SDK. The name of the evaluator is *simEval*.

```
problem.Evaluators().Item(Eval_Type.Sample).Evaluator(new  
simEval());
```

11. **Write the simulation evaluator.**

- i. Inside the evaluator *evalSim*, we create two convenient pointers, *Var* and *Fcn*, to hold the current values of the variables and uncertain functions, respectively.

```
double var[] = evaluator.Problem().VarUncertain().Value();
```

```
double Fcn[] = evaluator.Problem().FcnUncertain().Value();
```

- ii. Compute the uncertain function values given the current variable values.

```
Fcn[0] = Var[0] + Var[1];
```

```
Fcn[1] = Var[0] + Var[1];
```

- iii. Next, pass the uncertain function values back to the SDK.

```
evaluator.Problem().FcnUncertain().Value(Fcn);
```

Finally, tell the SDK to continue simulating by returning *Engine_Action.Continue*. There are two additional *Engine_Action* functions: *Engine_Action.Evaluate* and *Engine_Action.Stop*. *Engine_Action.Evaluate* forces the SDK to call the evaluator of type *Eval_Type.Function* a second time, which can be useful if you need to inspect the variable values.

Engine_Action.Stop communicates to the SDK that an error has occurred during the calculation of the uncertain functions.

Note: If an error occurs during the calculation of the uncertain functions, the SDK marks the current trial as an error trial and continues with the remaining trials.

```
return Engine_Action.Continue
```

Our evaluator is complete. We will now finish the code inside of the main function.

12. **Solve the simulation model.** Back inside the *simulation()* function, call the following line of code to run the simulation.

```
problem.Solver().Simulate();
```

13. **Obtain and display the simulation status, as well as various simulation statistics.**

```
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, 10));  
    System.out.println("90th Percentile = " +  
problem.VarUncertain().Percentiles().Value(i, 90));  
}
```

```
System.out.println("");  
System.out.println("Statistics on Uncertain Functions");
```

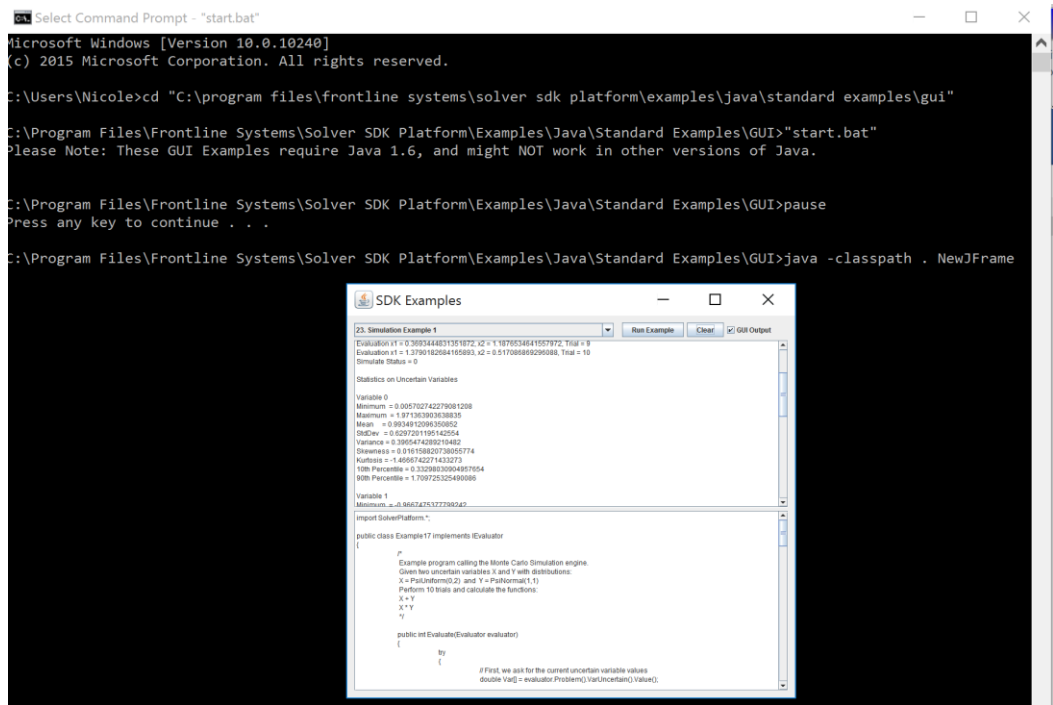
```
for (int i = 0; i < nfcns; i++)  
{  
    System.out.println("");  
    System.out.println("Function " + i);  
    System.out.println("Minimum = " +  
problem.FcnUncertain().Statistics().Minimum(i));  
    System.out.println("Maximum = " +  
problem.FcnUncertain().Statistics().Maximum(i));  
    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, 10));
System.out.println("90th Percentile = " +
problem.FcnUncertain().Percentiles().Value(i, 90));
}

```

14. **Solve the example.** To solve this example using the GUI console, open a prompt (right-click and select Run as Administrator), navigate to the Java\GUI directory and type "start.bat" at the prompt. The following dialog opens. Select **Simulation Example 1** using the drop-down menu at the top, then click **Run Example**. To run the console example, see the Introduction to this chapter for instructions.



Calling the Solver SDK from Python

Introduction

This chapter details the steps required to write and run a simple linear, nonlinear, and simulation example using Solver SDK and 64-bit Python.

The script editor Notepad++ is used to edit the example code in **Examples.py**. The Python interpreter reads and parses the example script, compiles it into bytecode, and executes in run-time. To run the Python example, the required Python libraries must be downloaded and installed. Currently Solver SDK supports the following versions of Python: 3.7, 3.8 and 3.9.

The XLMiner SDK Python SDK consists of native binaries compiled with certain configurations. As a result, it is important to install the appropriate version of Python. Before calling XLMiner SDK, confirm that Python DLLs are accessible using the system path.

To run the PYTHON Solver SDK Examples, first, follow the steps in the "Installation" chapter to install all required SDK files, including Example files, into the Frontline Systems directory located at C:\Program Files. There are two ways to view and run the Python examples: by command prompt executing sequential commands, or by writing a full script and executing the full code using a Python notebook or a full Python IDE.

Command Prompt—if using a command prompt, simply open a **python.exe** interpreter (or call it for each command) to work in "interactive shell" mode. Using this method, single commands are executed in succession.

Using a Python IDE or Notebook—open a Python IDE to execute a full script such as **PyCharm** by JetBrains, Visual Studio 2019 (with Python tools installed), or a Python Notebook such as **IPython**. (The example code was written using **IPython**.)

To call Solver SDK from Python, you will need **SolverPlatform.pyd** and **SolverSDK.dll** (along with any other binaries that **SolverSDK.dll** depends upon) which are installed into the **Bin** folder (see above for complete path). Both files must be on the PYTHONPATH or in the same folder. **SolverSDK.dll** and the **Bin** folder (C:\Program Files\Frontline System\Analytic Solver SDK\SolverSDK\Bin) are added to the system path during installation.

Various editors can be used to write a Python script. In this example, Python's Integrated Development Environment (IDLE) will be used.

Creating a Linear Model

1. **First steps.** Open a new document in Notepad++. In this example, we will name the script, *PythonLinearExample()*. The first line of any script calling SolverSDK.dll must begin with import SolverPlatform. This line of code allows Python to access the Solver SDK Platform.

```
import SolverPlatform
```

2. **Create the linear model.** In this section, we will solve a linear optimization problem with two variables (x1 and x2) and three constraints. Both final variable values must be zero or larger.

```
Maximize 2 x1 + 3 x2
Subject To:
9 x1 + 6 x2 <= 54
6 x1 + 7 x2 <= 42
5 x1 + 10 x2 <= 50
x1, x2 non-negative integers
```

3. To begin, the example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model. In addition, three arrays are created to hold the objective coefficients, constraint coefficients, and constraint right sides.

```
let nvars = 2
let ncons = 3
objcoeff = [ 2, 3 ]
concoeff = [ 9, 6, 6, 7, 5, 10 ]
rhsvalue = [ 54, 42, 50 ]
```

4. There are two different constructors which create a *Problem*. The first constructor (*Problem prob = new Problem()*) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call *prob.VarDecision.Size* to automatically add a variable block or create a variable block using *Variable vars(Variable Type Decision, "vars", 1)* where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. Additionally, the User must add the variable to the problem by calling *prob.Variables.Add(vars)*. This is true for functions as well. A user can either add constraints automatically using *prob.FcnConstraint.Size* or create a function with *Function con1(Function Type Constraint, "con", 1)* (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling *prob.Functions.Add(con)* to add the function to the Problem.

The second constructor, used in this example, not only creates the Problem, but also passes the type of model to be solved, the number of variables (*nvars*) and the number of constraints (*ncons*).

```
problem =
SolverPlatform.Problem(SolverPlatform.Solver_Type_Maximize,
nvars, ncons
```

5. **Call AllLinear functions.**

The line of code, *problem.Model.AllLinear[SolverPlatform.Function Type Objective]*, passes the objective coefficient matrix and *problem.Model.AllLinear[Function Type Constraint]* passes the constraint coefficient matrix. Calling *prob.Model.AllLinear* sets the problem type to "linear" (Problem Type.OptLP) and will automatically select the best linear engine to solve the problem. If no additional Large Scale "plug in" engines have been added, then the LP/Quadratic engine will be selected.

Since both objcoeff and concoeff are arrays, and problem.Model.AllLinear is expecting a DoubleMatrix, we must convert both arrays to a DoubleMatrix. First, we create the double matrix,

obj, containing 1 row, *nvars* columns, and the values contained in the *objcoeff* array passed column-wise. Then we pass this DoubleMatrix object to `problem.Model.AllLinear[SolverPlatform.Function_Type_Objective]`. Next, we create the double matrix *constraints*, containing *ncons* rows, *nvars* columns, and the values contained in the *concoeff* array passed column-wise. (We could also pass the values in the arrays row-wise using *Array_Order_ByRow*.)

```
obj =
SolverPlatform.DoubleMatrix(SolverPlatform.Array_Order_ByRow, 1,
nvars, objcoeff)

problem.Model.AllLinear[SolverPlatform.Function_Type_Objective] =
obj

constraints =
SolverPlatform.DoubleMatrix(SolverPlatform.Array_Order_ByRow,
ncons, nvars, concoeff)

problem.Model.AllLinear[SolverPlatform.Function_Type_Constraint]
= constraints
```

6. **Assign the constraint right hand sides and add the non-negativity constraint.**

We pass the *ub_constraints* array which holds the upper bounds of the constraint right hand sides.

```
problem.FcnConstraint.UpperBound.Array = rhsvalue
```

To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
problem.VarDecision.NonNegative()
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass non-zero upper and lower variable bounds to the SDK.

7. **Pass integer constraints.**

Use `problem.VarDecision.IntegerType[0]` to add integer constraints to both variables.

```
problem.VarDecision.IntegerType[0] =
SolverPlatform.Integer_Type_Integer

problem.VarDecision.IntegerType[1] =
SolverPlatform.Integer_Type_Integer
```

Solve the optimization model.

Solve the optimization model by calling:

```
problem.Solver.Optimize()
```

8. **Obtain the status and final variable/constraint values.**

We capture the status of the optimization and final variable and constraint values, then display each using the print command.

```
status = problem.Solver.OptimizeStatus
convalue = problem.FcnConstraint.FinalValue.Array
varvalue = problem.VarDecision.FinalValue.Array
print("Linear Python Example ")
print("Status = %i" % status)
print("x1 = %d" % varvalue[0])
print("x2 = %d" % varvalue[1])
print("Obj = %d" % problem.FcnObjective.FinalValue[0])
```

9. **Run the script.** To run the script, click **Run – Run Module**. An output window appears displaying the result of the print commands.

Creating a Non-Linear Model

1. **Basic Steps. First steps.** Open a new document in Notepad++. In this example, we will name the script, *PythonNonlinearExample()*. The first line of any PYTHON script calling SolverSDK.dll must begin with import SolverPlatform. This line of code allows Python to access the Solver SDK Platform.

```
import SolverPlatform
```

2. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constraint (x + y) forces the total of the two variables to be equal to 1 and the second constraint (x * y) forces the product of the two variables to be zero or larger. Both final variable values must be non-negative (0 or larger).

```
Minimize x^2 + y^2
Subject To:
x + y = 1
x * y >= 0
x, y non-negative
```

- i. To begin, we initialize the number of variables (*nvars*) and constraints (*ncons*).

```
nvars = 2
ncons = 2
```

- ii. Next, we create two arrays of size *ncons* (2 constraints) to hold the upper and lower bounds on the constraint right hand sides. Notice the upper and lower bound of the first constraint is 1 to denote an equality constraint. *CConstants::PINF* sets the upper bound of the second constraint to positive infinity.

```
rhs = [ 1.0, SolverPlatform.Constants().PINF ]
lhs = [ 1.0, 0 ]
```

- iii. The next two lines of code create two arrays to hold the lower and upper variable bounds.

```
lb = [ 0, 0 ]
ub = [ 1, 1 ]
```

3. This next line of code creates an instance of the Problem class, passes the “sense” of the problem, the number of decision variables and the number of constraints. For more information on creating this class, see the Linear example above.

```
problem =
SolverPlatform.Problem(SolverPlatform.Solver_Type_Minimize,
nvars, ncons);
```

Here we pass the constraint upper (*ub_constraints*) and lower (*lb_constraints*) bounds to the SDK,

```
prob.FcnConstraint.UpperBound.Array = rhs;
prob.FcnConstraint.LowerBound.Array = lhs;
```

following up with the variable upper (*ub*) and lower (*lb*) bounds.

```
prob.VarDecision.UpperBound.Array = ub;
prob.VarDecision.LowerBound.Array = lb;
```

4. **Create the nonlinear evaluator.** In a linear model or quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to

compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied. (It is possible to pass these derivatives directly to the SDK using the *Eval_Type_Gradient* evaluator. For more information on supplying this type of evaluator, see the subheading *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter, *Solver SDK Evaluators*.)

We create and pass the evaluator of type *Eval_Type_Function* (as described above) to the SDK. The name of this evaluator is *evalNLP*.

```
prob.Evaluators[SolverPlatform.Eval_Type_Function].Evaluator =
evalNLP
```

5. **Write the nonlinear evaluator.** Inside of the evaluator *evalNLP*, we first create a convenient pointer, *var*, to the variables.

```
def evalNLP(ev) :
var = ev.prob.VarDecision.Value.Array
```

Here, we pass the first constraint, $x + y$,

```
ev.prob.FcnConstraint.Value[0] = var[0] + var[1]
```

and now the second constraint, $x * y$.

```
ev.prob.FcnConstraint.Value[0] = var[0] * var[1]
```

Then we compute the objective function, $x * x + y * y$.

```
ev.prob.FcnObjective.Value[ev.Prob.ObjectiveIndex] = var[0] *
var[0] + var[1] * var[1]
```

The line of code below prints the current values of the variables and the iteration number each time the *evalNLP* function is called. (This is optional.)

```
print("Evaluation x1 = %f, x2 = %f, Iteration = %i" % (var[0],
var[1], ev.Problem.Engine.Stat.FunctionEvals))
```

Finally, we tell the SDK to continue optimizing by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the functions (objective or constraints). The Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine (for solving nonsmooth problems) will disregard the current point and continue solving.

```
return SolverPlatform.Engine_Action_Continue
```

6. **Solve the nonlinear problem.** Back inside the *nonlinear()* function, we provide the problem type as nonlinear which directs the SDK to select the GRG engine to solve the model. Alternatively, we also could have set the GRG engine manually (in either the SDK Platform or SDK Pro) by calling: *prob.Engine = prob.Engines.[Engines.GRGName]*;

```
prob.ProblemType = SolverPlatform.Problem_Type_OptNLP
```

To solve the nonlinear optimization problem, we call:

```
prob.Solver.Optimize()
```

7. **Obtain the solution status.** We obtain the solution status, along with the final objective and variable values,

```
status = prob.Solver.OptimizeStatus
objvalue = prob.FcnObjective.FinalValue[0]
varvalue = prob.VarDecision.FinalValue.Array
convalue = prob.FcnDecision.FinalValue.Array
```

```

print("Example 4:")
print ("Status = %i % status)
print("x1 + %f" % varvalue[0])
print("x2 = %f" % varvalue[1])
print("Constraint 1 + %f" % conValue[0])
print("Constraint 2 + %f" % conValue[1])
print("Obj = %f" % objvalue)

```

8. **Run the script.** To run the script, click Run – Run Module. An output window appears displaying the result of the print commands.

```

Python 2.7.10 Shell
File Edit Shell Debug Options Window Help
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ----- RESTART -----
>>>
Evaluation x1 = 0.000000, x2 = 0.000000, Iteration = 1
Evaluation x1 = 0.000000, x2 = 0.000000, Iteration = 2
Evaluation x1 = 0.000000, x2 = 0.000000, Iteration = 3
Evaluation x1 = 0.000001, x2 = 0.000000, Iteration = 3
Evaluation x1 = 0.000000, x2 = 0.000001, Iteration = 3
Evaluation x1 = 0.000000, x2 = 0.000000, Iteration = 4
Evaluation x1 = 0.000000, x2 = 0.000000, Iteration = 5
Evaluation x1 = 0.000001, x2 = 0.000000, Iteration = 5
Evaluation x1 = 0.000000, x2 = 0.000001, Iteration = 5
Evaluation x1 = 0.050000, x2 = 0.050000, Iteration = 6
Evaluation x1 = 0.050000, x2 = 0.050000, Iteration = 7
Evaluation x1 = 0.100000, x2 = 0.100000, Iteration = 8
Evaluation x1 = 0.100000, x2 = 0.100000, Iteration = 9
Evaluation x1 = 0.200000, x2 = 0.200000, Iteration = 10
Evaluation x1 = 0.200000, x2 = 0.200000, Iteration = 11
Evaluation x1 = 0.400000, x2 = 0.400000, Iteration = 12
Evaluation x1 = 0.400000, x2 = 0.400000, Iteration = 13
Evaluation x1 = 0.800000, x2 = 0.800000, Iteration = 14
Evaluation x1 = 0.800000, x2 = 0.800000, Iteration = 15
Evaluation x1 = 0.500000, x2 = 0.500000, Iteration = 16
Evaluation x1 = 0.500000, x2 = 0.500000, Iteration = 17
Evaluation x1 = 0.500000, x2 = 0.500000, Iteration = 18
Evaluation x1 = 0.500001, x2 = 0.500000, Iteration = 18
Evaluation x1 = 0.500000, x2 = 0.500001, Iteration = 18
Iteration 1
Objective = 0.500000
Evaluation x1 = 0.500000, x2 = 0.500000, Iteration = 19
Evaluation x1 = 0.500000, x2 = 0.500000, Iteration = 20
Example 4:
Status = 0
x1 = 0.500000
x2 = 0.500000
Obj = 0.500000
>>>
Ln: 38 Col: 4

```

Solving a Simulation Problem

1. **Basic Steps. First steps.** Open a new document in Notepad++. In this example, we will name the script, *PythonNonlinearExample()*. The first line of any PYTHON script calling SolverSDK.dll must begin with import SolverPlatform. This line of code allows Python to access the Solver SDK Platform.

```
import SolverPlatform
```

2. **Create a simulation problem.** The simulation model that we will be solving in this example contains two uncertain variables, x and y. The uncertain variable x has a uniform distribution with a lower parameter of 0 and an upper parameter of 2. The uncertain variable y has a normal distribution with a mean and standard deviation equal to 1. In addition, there are two uncertain functions. The first uncertain function is the total of the two uncertain variables. The second is the product of the two uncertain variables.

Given two uncertain variables X and Y with distributions:

```
x = PsiUniform(0,2) and y = PsiNormal(1,1)
```

```
Function[1]: x + y
```

```
Function[2]: x * y
```

This example uses a correlation matrix which measures the degree to which the movements of the two uncertain variables are related. This is optional. The correlation matrix is:

	X	Y
X	1	.5
Y	.5	1

The correlation coefficients in the *correl* array are being passed row-wise.

```
correl = [ 1.0, 0.5, 0.5, 1.0 ]
```

To begin, we create an instance of the Problem class. There are two different constructors which create a *Problem*. The first constructor (as used in this example) will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving an optimization or simulation problem. The second constructor, which is used in the Linear and Non-Linear examples above, not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type_Minimize*, *Solver_Type_Maximize*, or *Solver_Type_Simulate*) the number of variables (*nvars*) and the number of functions (*nfcns*).

```
myprob = SolverPlatform.Problem();
```

The next line of code sets the problem type to “Simulate”.

```
myprob.Solver.SolverType = SolverPlatform.Solver_Type_Simulate
```

Next we create two uncertain variables and two uncertain functions. Notice the three arguments to *SolverPlatform.Variable* and *SolverPlatform.Function*. The first argument is the Variable or Function type, uncertain, the second is the name of the variable or function block and the third is the number of variables and functions contained within the block.

```
uncVarUniform =
SolverPlatform.Variable(SolverPlatform.Variable_Type_Uncertain,
"Var1", 1)
uncVarNormal =
SolverPlatform.Variable(SolverPlatform.Variable_Type_Uncertain,
"Var2", 1)

uncFcnAdd =
SolverPlatform.Function(SolverPlatform.Function_Type_Uncertain,
"Fcn1", 1)
uncFcnMult =
SolverPlatform.Function(SolverPlatform.Function_Type_Uncertain,
"Fcn2", 1)
```

then add each of them to the problem, *myprob*.

```
myprob.Variables.add(uncVarUniform)
myprob.Variables.add(uncVarNormal)

myprob.Functions.Add(uncFcnAdd)
myprob.Functions.Add(uncFcnMult)
```

These two lines of code pass the distributions of the two uncertain variables, Uniform and Normal.

```
myprob.Model.VarDistribution[0] =
SolverPlatform.Distribution("PsiUniform(0,2)");

myprob.Model.VarDistribution[1] =
SolverPlatform.Distribution("PsiNormal(1,1)");
```

We can also specify a lower and upper bound for *uncVarNormal*, the uncertain variable with the Normal distribution. *ControlParam[1]* specifies the lower bound for the distribution and *ControlParam[2]* specifies the upper bound for the distribution. For a complete list of control

parameters, please see the Distribution Control Parameters section that occurs later on in this guide.

```
myprob.Model.VarDistribution[1].ControlParams[1] = 0;
myprob.Model.VarDistribution[1].ControlParams[2] = 3;
```

Here we create a new double matrix of size nvars x nvars (2 x 2) to hold the correlation matrix. We are passing the *correl* array row-wise.

```
myprob.Model.DistCorrelations =
SolverPlatform.DoubleMatrix(SolverPlatform.Array_Order_ByRow, 2,
2, correl )
```

3. **Create the simulation evaluator.** When running a simulation, the evaluator *Eval_Type_Sample* is called so the user can calculate the uncertain functions, much like *Eval_Type_Function* is called to compute the objective function and constraints in a nonlinear optimization problem.

We create the *Eval_Type_Sample* evaluator and pass the evaluator to the SDK. The name of the evaluator is *evalSim()*.

```
myprob.Evaluators[SolverPlatform.Eval_Type_Sample].Evaluator =
evalSim
```

4. **Write the simulation evaluator.** Inside of the evaluator *evalSim*, we first create a convenient pointer, *p*, to the Problem.

```
def evalSim(ev) :
p = ev.Problem
```

We compute the uncertain function values given the current variable values.

```
myprob.Functions[0].Value[0] = p.Variables[0].Value[0] +
p.Variables[1].Value[0]    # X+Y

myprob.Functions[1].Value[1] = p.Variables[0].Value[0] *
p.Variables[1].Value[0]    # X*Y
```

Finally, we tell the SDK to continue simulating by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the uncertain functions. The SDK will mark the current trial as an error trial and continue with the remaining trials.

```
return SolverPlatform.Engine_Action_Continue
```

Our evaluator is complete. We will now finish the code inside of the *simulation()* function.

5. **Solve the simulation model.** Back inside the *simulation()* function, we call the following line of code to run the simulation.

```
myprob.Solver.Simulate();
```

6. **Obtain and display various simulation statistics.** We obtain the simulation status.

```
print("The status of the solver after simulation is: %d" %
myprob.Solver.SimulateStatus)

# display the output statistics
print("Statistics on the sampled uncertain variables:\n")
print("Uncertain Variable X:")
print("Max value: %f" %
      myprob.Variables[0].Statistics.Maximum[0])
```

```

print("Min value: %f" %
      myprob.Variables[0].Statistics.Minimum[0])
print("Mean: %f" % myprob.Variables[0].Statistics.Mean[0])
print("Variance: %f\n" %
      myprob.Variables[0].Statistics.Variance[0])

print("Uncertain Variable Y:")
print("Max value: %f" %
      myprob.Variables[1].Statistics.Maximum[1])
print("Min value: %f" %
      myprob.Variables[1].Statistics.Minimum[1])
print("Mean: %f" % myprob.Variables[1].Statistics.Mean[1])
print("Variance: %f\n" %
      myprob.Variables[1].Statistics.Variance[1])

print("Statistics on the 1st uncertain function, x + y")
print("Max Value: %f" %
      myprob.Functions[0].Statistics.Maximum[0])
print("Min Value: %f" %
      myprob.Functions[0].Statistics.Minimum[0])
print("Mean: %f" % myprob.Functions[0].Statistics.Mean[0])
print("Variance: %f" %
      myprob.Functions[0].Statistics.Variance[0])

percentiles = myprob.Functions[0].Percentiles

print("The 50th percentile is: %f" % percentiles.Value[0][50])
print("The 5th percentile is: %f" % percentiles.Value[0][5])
print("The 95th percentile is: %f" % percentiles.Value[0][95])
print("The mode is: %f\n"
      %myprob.Functions[0].Statistics.Mode[0])

```

7. **Build and run the application.** To run the script, click Run – Run Module. An output window appears displaying the result of the print commands.

```

File Edit Shell Debug Options Window Help
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>> The status of the solver after simulation is: 0
Statistics on the sampled uncertain variables:

Uncertain Variable X:
Max value: 1.998181
Min value: 0.000307
Mean: 1.000036
Variance: 0.333628

Uncertain Variable Y:
Max value: 2.987689
Min value: 0.000564
Mean: 1.229609
Variance: 0.520324

Statistics on the 1st uncertain function (x + y)
Max Value: 4.966930
Min Value: 0.013999
Mean: 2.229645
Variance: 1.259569
The 50th percentile is: 2.195283
The 5th percentile is: 0.452644
The 95th percentile is: 4.242995
The mode is: 2.197009

>>>

```

Calling the Solver SDK from R

Introduction

This chapter details the steps required to write and run simple linear, nonlinear, and simulation examples by calling Solver SDK from the R programming language. Currently, Solver SDK only supports the use of R!, version 4 (version 4.x).

To open example code for R, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\R\StandardExamples, if using 64-bit Solver SDK. Thirty-two scripts, one for each example model, are contained in the R StandardExamples folder: Example1.R, Example2.R, etc. Each of these scripts contain definitions for R functions that illustrate sample usage of Solver SDK.

Creating a Linear Model

1. **First steps.** Open a new document in Notepad++. In this example, we will name the script, *RLinearExample()*. The first line of any script calling SolverSDK.dll must begin with "library(SolverPlatform)". This line of code allows R to access the Solver SDK Platform.
2. **Create the linear model.** In this section, we will solve a linear optimization problem with two variables (x1 and x2) and three constraints. Both final variable values must be zero or larger.

```
Maximize 2 x1 + 3 x2
Subject To:
9 x1 + 6 x2 <= 54
6 x1 + 7 x2 <= 42
5 x1 + 10 x2 <= 50
x1, x2 non-negative integers
```

3. **Specify the number of variables and constraints, pass the coefficient matrix and the constraint right-hand sides.**

To begin, the example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model. In addition, three arrays are created to hold the objective coefficients, constraint coefficients, and constraint right sides. Note that the coefficients for the constraint array, *concoeff*, are passed by row.

Note: The `concoeff()` array is passed row-wise.

```
nvars <- 2
ncons <- 3
objcoeff <- c( 2, 3 )
concoeff <- ( 9, 6, 6, 7, 5, 10 )
rhsvalue <- ( 54, 42, 50 )
```

4. Instantiate the Problem and pass the number of variables and constraints to the Problem.

There are two different constructors which create a *Problem*. The first constructor, `problem <- Problem$new()` as used in this example, will only create the problem. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. If this constructor is used, a user must either call `problem$VarDecision$Size` to automatically add a variable block or create a variable block using `Variable$new(Variable_Type_Decision, "VariableBlockName", numvars)` where the first argument is the variable type, the 2nd argument is a name chosen by the user, and the third argument is the size of the variable block. (Additionally, the User must add the variable to the problem by calling `prob$Variables$Add(vars)`.) This is true for functions as well. A user can either add constraints automatically using `prob$FcnConstraint$Size` or create a function with `Function$new(Function_Type_Constraint, "FunctionBlockName", 1)` (where the first argument is the function type, the 2nd argument is a name chosen by the user, and the third argument is the size of the function block) and then calling `prob$Functions$Add(con)` to add the function to the Problem.

The second constructor, used in the example below, not only creates the Problem, but also passes the type of model to be solved, the number of variables (*nvars*) and the number of constraints (*ncons*).

In this example, the first line of code creates a *Problem*. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem. A user must call `problem$VarDecision$Size` to add a variable block and `problem$FcnConstraint$Size` to add a function block then `problem$Solver$SolverType` to designate the problem as a minimization or maximization. In this example, we pass *nvars* to pass 2 variables, *ncons* to pass our constraint coefficients and `Solver_Type_Maximize` to delineate the problem as a maximization. In this example, the body of code is contained within a "tryCatch" block in order to handle any exceptions that may occur during the execution of the program.

```
tryCatch({
  problem <- Problem$new()
  problem$VarDecision$Size <- nvars
  problem$FcnConstraint$Size <- ncons
  problem$Solver$SolverType <- Solver_Type_Maximize
})
```

5. Assign the constraint right hand sides and add the non-negativity constraint.

We pass the three constraint right hand sides separately by passing each element of the *rhsvalue* array.

```
problem$FcnConstraint$UpperBound[0] <- rhsvalue[1]
problem$FcnConstraint$UpperBound[1] <- rhsvalue[2]
problem$FcnConstraint$UpperBound[2] <- rhsvalue[3]
```

To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
problem$VarDecision$NonNegative()
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass non-zero upper and lower variable bounds to the SDK.

6. Call AllLinear functions.

Calling `prob$Model$AllLinear` sets the problem type to "linear" (Problem_Type_OptLP) and will automatically select the best linear engine to solve the problem. If no additional Large Scale "plug in" engines have been added, then the LP/Quadratic engine will be selected.

The second line of code below, `al[Function_Type_Objective] <- DoubleMatrix$new(Array_Order_ByRow, 1, nvars, objcoeff)`, passes the objective coefficient matrix row-wise.

The third line of code below, `al[Function_Type_Constraint] <- DoubleMatrix$new(Array_Order_ByRow, ncons, nvars, concoeff)`, passes the constraint coefficient matrix.

Since both `objcoeff` and `concoeff` are arrays, and `problem$Model$AllLinear` is expecting a `DoubleMatrix`, we must convert both arrays to a `DoubleMatrix`. First, we create a new double matrix containing 1 row, `nvars` columns, and the values contained in the `objcoeff` array passed row-wise. Then we pass this `DoubleMatrix` object to `al[Function_Type_Objective]`.

Next, we create a 2nd double matrix, containing `ncons` rows, `nvars` columns, and the values contained in the `concoeff` array passed row-wise. (We could also pass the values in the arrays row-wise using `Array_Order_ByCol`.)

```
al <- problem$Model$AllLinear
al[Function_Type_Objective] <-
DoubleMatrix$new(Array_Order_ByRow, 1, nvars, objcoeff)
al[Function_Type_Constraint] <-
DoubleMatrix$new(Array_Order_ByRow, ncons, nvars, concoeff)
```

7. Solve the optimization model.

Solve the optimization model by calling:

```
problem$Solver$Optimize()
```

8. Obtain the status and final variable/constraint values.

We capture the status of the optimization and final variable and constraint values, then display each using the print command.

```
status <- problem$Solver$OptimizeStatus
convalue <- problem$FcnConstraint$FinalValue$Array
varvalue <- problem$VarDecision$FinalValue$Array
writeLines("Linear R Example ")
writeLines(paste("Status = ", status))
writeLines(paste("x1 = ", varvalue[1]))
writeLines(paste("x2 = ", varvalue[2]))
writeLines(paste("Obj = ", problem$FcnObjective$FinalValue[0]))
```

9. The error handlers below generate warnings to the user if an error is encountered.

```
warning = function(war) {
  # warning handler picks up where error was generated
  writeLines(paste("WARNING: ", war))
  return(TRUE)
}, error = function(err) {
  # error handler picks up where error was generated
  writeLines(paste("Error: ", err))
  return(TRUE)
},
```

10. Finally, resources are freed and module variables are removed.

```
finally = {
```

```

# free allocated resources
problem$Free()

# removing module variables
rm(problem, nvars, ncons, objcoeff, concoeff, rhsvalue, al,
    ev, evalIter, status, varvalue)
})

```

This ends this section which explained how to solve a linear optimization model using the Solver SDK and the R programming language. See the next section for an explanation on how to solve a non-linear optimization model.

Complete Code

```

library(SolverPlatform)

# Example 1: Solves the LP model:
# Maximize    2 x1 + 3 x2
# Subj to     9 x1 + 6 x2 <= 54
#             6 x1 + 7 x2 <= 42
#             5 x1 + 10 x2 <= 50
# x1, x2 non-negative
# LP solution: x1 = 2.8, x2 = 3.6
# Objective = 16.4
#
# Matrix is passed rowwise, dense.

evalIter = function(ev)
{
  writeLines(paste("Iteration ", ev$Problem$Engine$Stat$Iterations))
  writeLines(paste("Objective = ",
    ev$Problem$FcnObjective$Value[0]))

  return (Engine_Action_Continue)
}

nvars <- 2
ncons <- 3

objcoeff <- c( 2, 3 )
concoeff <- c( 9, 6, 6, 7, 5, 10 ) # note that these are row-wise
rhsvalue <- c( 54, 42, 50 )

tryCatch({

  problem <- Problem$new()

  problem$VarDecision$Size <- nvars      # 2 decision variables
  problem$FcnConstraint$Size <- ncons    # 3 constraint functions

  problem$Solver$SolverType <- Solver_Type_Maximize

  problem$FcnConstraint$UpperBound[0] <- rhsvalue[1]
  problem$FcnConstraint$UpperBound[1] <- rhsvalue[2]
  problem$FcnConstraint$UpperBound[2] <- rhsvalue[3]

  problem$VarDecision$NonNegative()      # vars >= 0

```

```

al <- problem$Model$AllLinear
# linear model; we pass the coefficients for the objective fcn
al[Function_Type_Objective] <- DoubleMatrix$new(Array_Order_ByRow,
1, nvars, objcoeff)

# linear model; we pass the coefficients for the constraint fcns
al[Function_Type_Constraint] <-
DoubleMatrix$new(Array_Order_ByRow, ncons, nvars, concoeff)

ev <- problem$Evaluators[Eval_Type_Iteration]

ev$Evaluator <- evalIter # And pass the evaluator to the problem

problem$Solver$Optimize() # Solve model

status <- problem$Solver$OptimizeStatus # optimization status

varvalue <- problem$VarDecision$FinalValue$Array # final var values

writeLines("Example 1")
writeLines(paste("Status = ", status))
writeLines(paste("x1 = ", varvalue[1]))
writeLines(paste("x2 = ", varvalue[2]))
writeLines(paste("Obj = ", problem$FcnObjective$FinalValue[0]))

}, warning = function(war) {

# warning handler picks up where error was generated
writeLines(paste("WARNING: ",war))
return(TRUE)

}, error = function(err) {

# error handler picks up where error was generated
writeLines(paste("Error: ",err))
return(TRUE)

}, finally = {
# free allocated resources
problem$Free()

# removing module variables
rm(problem, nvars, ncons, objcoeff, concoeff, rhsvalue, al, ev,
evalIter, status, varvalue)
})

```

Creating a Non-Linear Model

1. **Basic Steps. First steps.** Open a new document in Notepad++. In this example, we will name the script, *RNonlinearExample()*. The first line of any R script calling SolverSDK.dll must begin with *library(SolverPlatform)*. This line of code allows R to access the Solver SDK Platform.

```
Library(SolverPlatform)
```

2. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constraint ($x + y$) forces the total of the two variables to be equal to 1 and the second constraint ($x * y$) forces the product of the two variables to be zero or larger. Both final variable values must be non-negative (0 or larger).

```
Minimize x^2 + y^2
Subject To:
x + y = 1
x * y >= 0
x, y non-negative
```

- i. To begin, we initialize the number of variables (*nvars*) and constraints (*ncons*).

```
nvars <- 2
ncons <- 2
```

- ii. Next, we create two arrays of size *ncons* (2 constraints) to hold the upper and lower bounds on the constraint right hand sides. Notice the upper and lower bound of the first constraint is 1 to denote an equality constraint. *CConstants::PINF* sets the upper bound of the second constraint to positive infinity.

```
rhs <- c( 1.0, Constants.PINF )
lhs <- ( 1.0, 0 )
```

- iii. The next two lines of code create two arrays to hold the lower and upper variable bounds.

```
lb <- c( 0, 0 )
ub <- c( 1, 1 ]
```

3. This next line of code creates an instance of the Problem class, passes the “sense” of the problem (maximize or minimize), the number of decision variables and the number of constraints. For more information on creating this class, see the Linear example above.

```
tryCatch({
  problem <- Problem$new(Solver_Type_Minimize, nvars, ncons)
```

Here we pass the constraint upper (*rhs*) and lower (*lhs*) bounds to the SDK,

```
problem$FcnConstraint$UpperBound$Array <- rhs
problem$FcnConstraint$LowerBound$Array <- lhs
```

following up with the variable upper (*ub*) and lower (*lb*) bounds.

```
problem$VarDecision$UpperBound$Array <- ub
problem$VarDecision$LowerBound$Array <- lb
```

4. **Create the nonlinear evaluator.** In a linear model or quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied. (It is possible to pass these derivatives directly to the SDK using the *Eval_Type_Gradient* evaluator. For more information on supplying this type of evaluator, see the subheading *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter, *Solver SDK Evaluators*.)

We create and pass the evaluator of type *Eval_Type_Function* (as described above) to the SDK. The name of this evaluator is *evalNLP*.

```
evalNLP <- problem$Evaluators[Eval_Type_Function]
```

```
evalNLP$Evaluator <- evalNLP
```

5. **Write the nonlinear evaluator.** Inside of the evaluator *evalNLP*, we first create a convenient pointer, *var*, to the variables.

```
evalNLP = function(evaluator)
{
  prob <- evaluator$Problem
```

First, we must ask for the values of the current variables.

```
var <- prob$VarDecision$Value$Array
```

Here, we pass the first constraint, $x + y$,

```
prob$FcnConstraint$Value[0] <- var[1] + var[2]
```

and now the second constraint, $x * y$.

```
prob$FcnConstraint$Value[1] <- var[1] * var[2]
```

Then we compute the objective function, $x * x + y * y$.

```
prob$FcnObjective$Value[0] <- var[1]*var[1]+var[2]*var[2]
```

The line of code below prints the current values of the variables and the iteration number each time the *evalNLP* function is called. (This is optional.)

```
writeLines(paste("Evaluation x1 = ", var[1], ", x2 = ",
var[2], ", Iteration = ", prob$Engine$Stat$FunctionEvals))
```

Finally, we tell the SDK to continue optimizing by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the functions (objective or constraints). The Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine (for solving nonsmooth problems) will disregard the current point and continue solving.

```
return (Engine_Action_Continue)
}
```

6. **Solve the nonlinear problem.** Back inside *RNonlinearExample()*, we provide the problem type as nonlinear which directs the SDK to select the GRG engine to solve the model. Alternatively, we also could have set the GRG engine manually (in either the SDK Platform or SDK Pro) by calling: *prob\$Engine = prob\$Engines[Engines\$GRGName]*;

```
problem$ProblemType <- Problem_Type_OptNLP
```

To solve the nonlinear optimization problem, we call:

```
prob$Solver$Optimize()
```

7. **Obtain the solution status and print results.** We obtain the solution status, along with the final variable and constraint values.

```
status <- problem$Solver$OptimizeStatus
convalue <- problem$FcnConstraint$FinalValue$Array
varvalue <- problem$VarDecision$FinalValue$Array
```

8. **Print the Results.**

```
writeLines("Nonlinear R Example")
```

```

writeLines(paste("Status = ", status))
writeLines(paste("x1 = ", varvalue[1]))
writeLines(paste("x2 = ", varvalue[2]))
writeLines(paste("Obj = ", problem$FcnObjective$FinalValue[0]))

```

9. **The error handlers below generate warnings to the user if an error is encountered.**

```

warning = function(war) {

  # warning handler picks up where error was generated
  writeLines(paste("WARNING:  ",war))
  return(TRUE)
}, error = function(err) {
  # error handler picks up where error was generated
  writeLines(paste("Error:  ",err))
  return(TRUE)
},

```

10. **Finally, resources are freed and module variables are removed.**

```

finally = {
  # free allocated resources
  problem$Free()

  # removing module variables
  rm(problem, nvars, ncons, varvalue, convalue, rhs, lhs, lb,
    ub,evalNLP, status)
})

```

Complete Code

```

library(SolverPlatform)
# Solves the problem:
#
# Minimize  $X^2 + Y^2$ 
# Subject to:
#  $X + Y = 1$ 
#  $X * Y \geq 0$ 
#
# (Solution is  $X = Y = 0.5$ , Objective = 0.5)

# This evaluator computes the values of the objective and
# constraint functions for
# a given set of variable values.

evalNLP = function(evaluator)
{
  prob <- evaluator$Problem

  # First, we must ask what the current variable values are
  var <- prob$VarDecision$Value$Array

  writeLines(paste("Evaluation x1 = ", var[1], ", x2 = ", var[2],
    ", Iteration = ", prob$Engine$Stat$FunctionEvals))

  # We set the constraint functions
  prob$FcnConstraint$Value[0] <- var[1] + var[2] # X+Y
  prob$FcnConstraint$Value[1] <- var[1] * var[2] # X*Y

```

```

# And we compute the objective,  $X^2 + Y^2$ 
prob$FcnObjective$Value[0] <- (var[1] * var[1] + var[2] *
var[2])

return(Engine_Action_Continue)
}

evalIter = function(ev)
{
  writeLines(paste("Iteration ",
ev$Problem$Engine$Stat$Iterations))
  writeLines(paste("Objective = ",
ev$Problem$FcnObjective$Value[0]))

  return (Engine_Action_Continue)
}

tryCatch({

  nvars <- 2
  ncons <- 2

  rhs <- c( 1.0, Constants.PINF )
  lhs <- c( 1.0, 0 )

  lb <- c(0, 0)
  ub <- c(1, 1)

  problem <- Problem$new(Solver_Type_Minimize, nvars, ncons)

  problem$FcnConstraint$UpperBound$Array <- rhs
  problem$FcnConstraint$LowerBound$Array <- lhs

  problem$VarDecision$UpperBound$Array <- ub
  problem$VarDecision$LowerBound$Array <- lb

  evNLP <- problem$Evaluators[Eval_Type_Function]
  evNLP$Evaluator <- evalNLP # pass evaluator.

  # Note that this evaluator is required.

  problem$ProblemType <- Problem_Type_OptNLP

  problem$Solver$Optimize()

  status <- problem$Solver$OptimizeStatus

  convalue <- problem$FcnConstraint$FinalValue$Array
  varvalue <- problem$VarDecision$FinalValue$Array

  writeLines("Example 4")
  writeLines(paste("Status = ", status))
  writeLines(paste("x1 = ", varvalue[1]))
  writeLines(paste("x2 = ", varvalue[2]))
  writeLines(paste("Obj = ", problem$FcnObjective$FinalValue[0]))

```



```

}, warning = function(war) {

  # warning handler picks up where error was generated
  writeLines(paste("WARNING:  ", war))
  return(TRUE)

}, error = function(err) {

  # error handler picks up where error was generated
  writeLines(paste("Error:  ", err))
  return(TRUE)

}, finally = {
  # free allocated resources
  problem$Free()

  # removing module variables
  rm(problem, nvars, ncons, rhs, lhs, lb, ub, evNLP, evIter,
status, convalue, varvalue)
})

```

Solving a Simulation Problem

1. **Basic Steps. First steps.** Open a new document in Notepad++. In this example, we will name the script, *RSimulationExample()*. The first line of any R script calling SolverSDK.dll must begin with import SolverPlatform. This line of code allows R to access the Solver SDK Platform.

```
library(SolverPlatform)
```

2. **Create a simulation problem.** The simulation model that we will be solving in this example contains two uncertain variables, x and y. The uncertain variable x has a uniform distribution with a lower parameter of 0 and an upper parameter of 2. The uncertain variable y has a Normal distribution with a mean and standard deviation equal to 1. In addition, there are two uncertain functions. The first uncertain function is the total of the two uncertain variables. The second is the product of the two uncertain variables.

Given two uncertain variables X and Y with distributions:

```

x = PsiUniform(0,2)   and   y = PsiNormal(1,1)
Function[1]:  x + y
Function[2]:  x * y

```

To begin, we create an instance of the Problem class. This example uses the 2nd constructor which labels the problem as a simulation model (Solver_Type_Simulate) with 2 uncertain variables and 1 uncertain function.

```
myprob <- Problem$new(Solver_Type_Simulate, 2, 1)
```

The next two lines of code pass the distributions of the two uncertain variables, Poisson and Uniform. Notice that the index for VarDistribution uses the "Flat Address Space" index. For more information on these distributions, see the Psi Function Reference chapter.

```

myProb$Model$VarDistribution$Item(0,
Distribution$new("PsiPoisson(3)"))

myProb$Model$VarDistribution$Item(1,
Distribution$new("PsiUniform(9,18)"))

```

We can also specify a lower and upper bound for *uncVarNormal*, the uncertain variable with the Normal distribution. ControlParam[1] specifies the lower bound for the distribution and ControlParam[2] specifies the upper bound for the distribution. For a complete list of control

parameters, please see the Distribution Control Parameters section that occurs later on in this guide.

```
myprob$Model$VarDistribution[1]$ControlParams[1] <- 0
myprob$Model$VarDistribution[1]$ControlParams[2] <- 3
```

This example uses a correlation matrix which measures the degree to which the movements of the two uncertain variables are related. This is optional. The correlation matrix is:

	X	Y
X	1	.5
Y	.5	1

Here we create a new double matrix of size nvars x nvars (2 x 2) to hold the correlation matrix. The correlation coefficients in the *correlationmatrix* array are being passed row-wise.

```
correlationmatrix <- c(1.0, 0.5, 0.5, 1.0)
corrmatrix <- DoubleMatrix$new(Array_Order_ByRow, 2, 2,
correlationmatrix)
```

3. **Create the simulation evaluator.** When running a simulation, the evaluator *Eval_Type_Sample* is called so the user can calculate the uncertain functions, much like *Eval_Type_Function* is called to compute the objective function and constraints in a nonlinear optimization problem.

We create the *Eval_Type_Sample* evaluator and pass the evaluator to the SDK. The name of the evaluator is *evalSim()*.

```
simEval <- myprob$Evaluators[Eval_Type_Sample]
simEval$Evaluator <- simulationEvaluator
```

4. **Write the simulation evaluator.**

```
simulationEvaluator <- function(e)
{
  problem <- e$Problem;
```

First, obtain the values of the uncertain variables.

```
var <- problem$Variables[0]
```

We compute the uncertain function values given the current variable values.

```
fcnoutput <- (var$Value[0] * var$Value[0] + -1.0 *
var$Value[1] * var$Value[1])
```

Pass the value of the computed function back to the Problem.

```
problem$Functions[0] <- fcnoutput
```

Finally, we tell the SDK to continue simulating by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the uncertain functions. The SDK will mark the current trial as an error trial and continue with the remaining trials.

```
return (Engine_Action_Continue)
}
```

Our evaluator is complete. We will now finish the code inside of the *simulation()* function.

5. **Solve the simulation model.** Back inside the *simulation()* function, we set the number of simulations to 1000 and run the simulation.

```
Myprob$Solver$NumTrials <- 1000
myprob$Solver$Simulate()
```

6. **Obtain and display** various simulation statistics. We obtain the simulation status and various statistics.

```
writeLines(paste("The status of the solver after simulation is:",
  myprob$Solver$SimulationStatus))
```

Display the output statistics for the first uncertain variable.

```
writeLines("Statistics on the sampled uncertain variables")
writeLines("Uncertain Demand")
writeLines(paste("Max value: ",
  myprob$Variables[0]$Statistics$Maximum[0]))
writeLines(paste("Min value: ",
  myprob$Variables[0]$Statistics$Minimum[0]))
writeLines(paste("Mean: ",
  myprob$Variables[0]$Statistics$Mean[0]))
writeLines(paste("Variance: ",
  myprob$Variables[0]$Statistics$Variance[0]))
```

Display the output statistics for the second uncertain variable.

```
writeLines("Uncertain Variable 1")
writeLines(paste("Max value: ",
  myprob$Variables[1]$Statistics$Maximum[0]))
writeLines(paste("Min value: ",
  myprob$Variables[1]$Statistics$Minimum[0]))
writeLines(paste("Mean: ",
  myprob$Variables[1]$Statistics$Mean[0]))
writeLines(paste("Variance: ",
  myprob$Variables[1]$Statistics$Variance[0]))
```

Display the output statistics for the first uncertain functions.

```
writeLines("Statistics on the first uncertain function")
writeLines(paste("Max Value: ",
  myprob$Functions[0]$Statistics$Maximum[0]))
writeLines(paste("Min Value: ",
  myprob$Functions[0]$Statistics$Minimum[0]))
writeLines(paste("Mean: ",
  myprob$Functions[0]$Statistics$Mean[0]))
writeLines(paste("Variance: ",
  myprob$Functions[0]$Statistics$Variance[0]))
percentiles <- myprob$Functions[0]$Percentiles
```

Display the output statistics for the second uncertain functions.

```
writeLines("Statistics on the first uncertain function")
writeLines(paste("Max Value: ",
  myprob$Functions[1]$Statistics$Maximum[0]))
writeLines(paste("Min Value: ",
  myprob$Functions[1]$Statistics$Minimum[0]))
writeLines(paste("Mean: ",
  myprob$Functions[1]$Statistics$Mean[0]))
writeLines(paste("Variance: ",
  myprob$Functions[1]$Statistics$Variance[0]))
percentiles <- myprob$Functions[0]$Percentiles
```

Complete Code

```
library(SolverPlatform)

# Example program calling the Monte Carlo Simulation engine.
# 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

evalSim <- function(evaluator) {

  problem <- evaluator$Problem

  # First, we ask for the current uncertain variable values
  var <- problem$VarUncertain$Value$Array

  writeLines(paste("Evaluation x1 = ", var[1], ", x2 = ", var[1], ",
    Trial = ", problem$Engine$Stat$FunctionEvals))

  # We get a convenient pointer to the uncertain functions

  # We compute the function values given the current variable values

  problem$FcnUncertain$Value[0] <- var[1] + var[2]   # X+Y
  problem$FcnUncertain$Value[1] <- var[1] * var[2]   # X*Y

  return (Engine_Action_Continue)
}

tryCatch({

  nvars <- 2
  nfcns <- 2

  problem <- Problem$new(Solver_Type_Simulate, nvars, nfcns)

  ev <- problem$Evaluators[Eval_Type_Sample]
  ev$Evaluator <- evalSim

  problem$Solver$NumTrials <- 10 # default is 1000

  vd <- problem$Model$VarDistribution
  vd[0] <- Distribution$new("PsiUniform(0,2)")
  vd[1] <- Distribution$new("PsiNormal(1,1)")

  correlationmatrix <- c(1.0, 0.5, 0.5, 1.0)
  corrmatrix <- DoubleMatrix$new(Array_Order_ByRow, 2, 2,
    correlationmatrix)

  problem$Solver$Simulate()

  status <- problem$Solver$SimulateStatus
  writeLines("Example 17")
  writeLines(paste("Status = ", status))
```

```

writeLines("Statistics on Uncertain Variables")

varStat <- problem$VarUncertain$Statistics
varPct <- problem$VarUncertain$Percentiles

for(i in 0:(nvars-1))
{
  writeLines(paste("Variable ", i))
  writeLines(paste("Minimum = ", varStat$Minimum[i]))
  writeLines(paste("Maximum = ", varStat$Maximum[i]))
  writeLines(paste("Mean = ", varStat$Mean[i]))
  writeLines(paste("StdDev = ", varStat$StdDev[i]))
  writeLines(paste("Variance = ", varStat$Variance[i]))
  writeLines(paste("Skewness = ", varStat$Skewness[i]))
  writeLines(paste("Kurtosis = ", varStat$Kurtosis[i]))
  writeLines(paste("10th Percentile = ", varPct$Values[i, 10]))
  writeLines(paste("90th Percentile = ", varPct$Values[i, 90]))
}

writeLines("Statistics on Uncertain Functions")

fcnStat <- problem$Functions[0]$Statistics;
fcnPct <- problem$Functions[0]$Percentiles;

for(i in 0:(nfcns-1))
{
  writeLines(paste("Function ", i))
  writeLines(paste("Minimum = ", fcnStat$Minimum[i]))
  writeLines(paste("Maximum = ", fcnStat$Maximum[i]))
  writeLines(paste("Mean = ", fcnStat$Mean[i]))
  writeLines(paste("StdDev = ", fcnStat$StdDev[i]))
  writeLines(paste("Variance = ", fcnStat$Variance[i]))
  writeLines(paste("Skewness = ", fcnStat$Skewness[i]))
  writeLines(paste("Kurtosis = ", fcnStat$Kurtosis[i]))
  writeLines(paste("10th Percentile = ", fcnPct$Values[i, 10]))
  writeLines(paste("90th Percentile = ", fcnPct$Values[i, 90]))
}

}, warning = function(war) {

# warning handler picks up where error was generated
writeLines(paste("WARNING: ", war))
return(TRUE)

}, error = function(err) {

# error handler picks up where error was generated
writeLines(paste("Error: ", err))
return(TRUE)

}, finally = {
  # free allocated resources
  problem$Free()

  # removing module variables
  rm(problem, nvars, nfcns, correl, status, ev, varStat, varPct,
    fcnStat, fcnPct, vd)
}

```

})

Calling the Solver SDK from C using the Procedural API

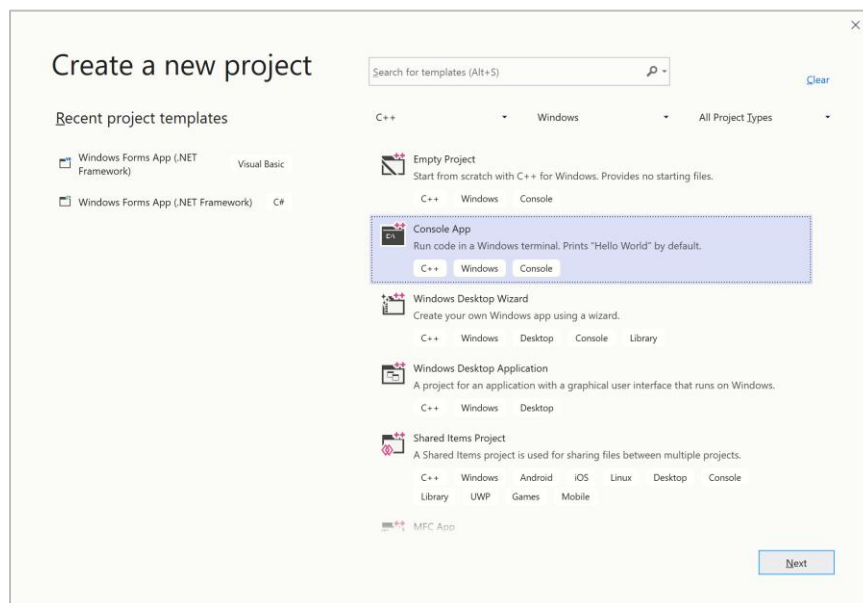
Introduction

The Solver SDK offers a **procedural API** for **C, Fortran, Pascal** and similar languages that provides access to the SDK's primary objects via conventional procedure calls. All procedural calls start with Solver... and the name or abbreviation of the SDK object, for example Prob, Mod, Var and Fcn.

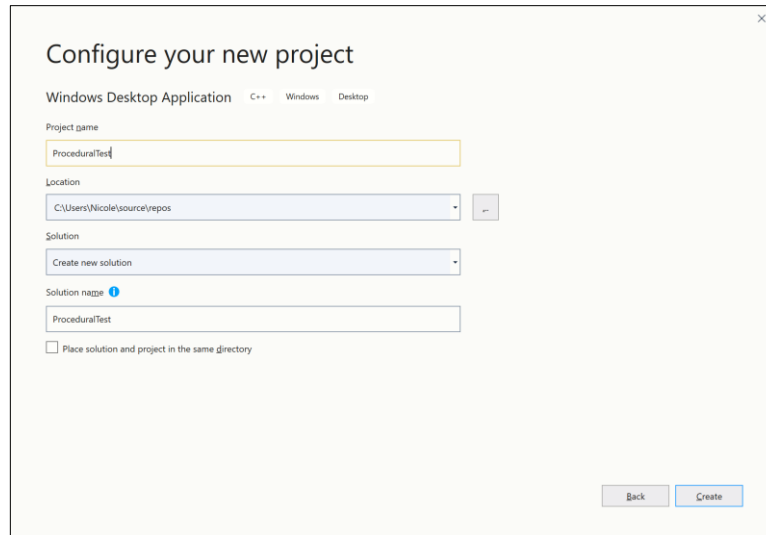
Creating a New C Project

First, follow the steps in the "Installation" chapter, which will install all required SDK files, including Example files, into the Frontline Systems directory (located at C:\Program Files) on your hard disk.

1. **Create a new project.** Start Microsoft Visual Studio, select File | New Project, select C on the left of the dialog and Console Application under Templates (on the right of the dialog). Type a name of your choice in the Project name field, and C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\C\ProceduralTest (or a path of your choosing) in the Location field. Then click OK.



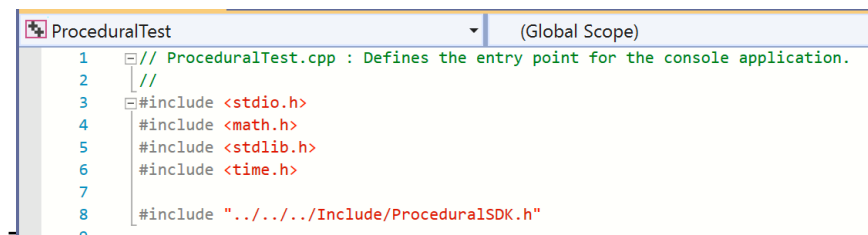
Click Finish on the next dialog.



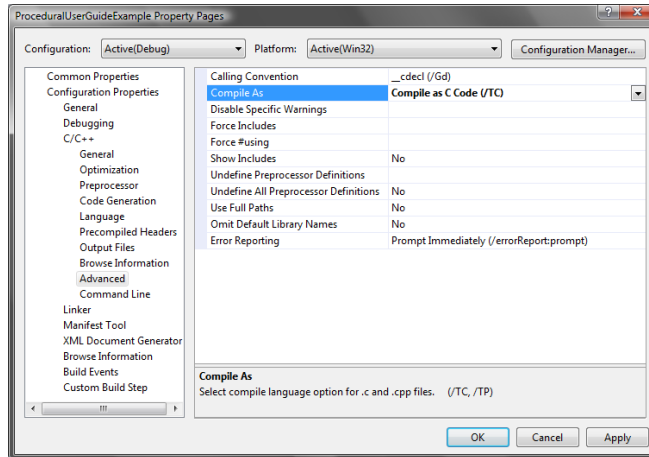
2. **Add the library file, SolverSDK.lib.** To create an application that calls the Solver SDK in C or C++, you must include the import library files, SolverSDK.lib in your project or your linker's response file. (Alternatively, you can write code to use "run-time dynamic linking" by calling Windows API routines such as LoadLibrary().)

Right click **Source Files** in the Solution Explorer and select **Add | Existing Item**. Navigate to C:\Program Files\Frontline Systems\Solver SDK Platform\SolverSDK\Lib and add the SolverSDK.lib file.

3. **Add an include statement.** In order to access the Solver SDK functions, you must add an *include* statement for the file, solversdk.h. To add the *include* statement, type `#include "C:/Program Files/Frontline Systems/Analytic Solver SDK/SolverSDK/Include/ProceduralSDK.h"` as shown in the screen print below.



4. **Set Language to C.** To illustrate how to call the Procedural API, the Compile As" property under "Advanced Properties" is set to "Compile as C Code". To open this dialog, go to View|Property



From here you can solve either a linear, nonlinear or simulation model. For more details on each, please read on.

Creating a linear model.

1. **First Steps.** Follow the steps in the section, Creating a New C Project. In this example, we will name the function that will hold the linear example, `linear()`.
2. **Create a linear model.** For this example, we will solve a linear programming model with two variables (x_1 and x_2) and three constraints. Both final variable values must be 0 or larger. The complete example model is shown below.

```
Maximize 2 x1 + 3 x2
Subject To:
9 x1 + 6 x2 <= 54
6 x1 + 7 x2 <= 42
5 x1 + 10 x2 <= 50
x1, x2 non-negative
```

To begin, the example code below first initializes two variables, *nvars* and *ncons*, to pass the number of decision variables (*nvars*) and constraints (*ncons*) in the model.

```
int nvars = 2, ncons = 3;
```

The next two lines of code set up two arrays to hold the objective (*objcoeff*) and constraint (*concoeff*) coefficients. Notice that the constraint coefficients in the *concoeff* array are passed row-wise. If the coefficients were being passed column-wise, the array contents would be: { 9, 6, 5, 6, 7, 10 }.

```
double objcoeff[] = { 2, 3 };
double concoeff[] = { 9, 6, 6, 7, 5, 10 };
```

Next, we create an array to hold the upper bounds of the constraint right hand sides.

```
double rhsvalue[] = { 54, 42, 50 };
```

Here we initialize the variable *objval* which will hold the final objective value,

```
double objval;
```

In the next three lines of code, we create an array to hold the final variable values (*x*), a problem handle (*prob*) and a variable of type `_Optimize_Status`, (*status*). The *x* array is of size two because there are two variables in the problem. *HProblem prob* creates the problem handle, *prob*, and `_Optimize_Status status` creates the variable, *status*, that will hold the status of the optimization.

```
double x[2];
HProblem prob;
_Optimize_Status status;
```

This next line of code creates a handle (*prob*) to the problem. In the remainder of the code, we will call additional routines, passing the problem handle as an argument.

```
SolverProblem(&prob);
```

Next, we set the problem type to *Problem_Type_Maximize* by calling *SolverProbTypeSet()*. To set the solver type to minimize, you would pass *Solver_Type_Minimize* instead. The third problem type, *Solver_Type_FindFeas* can be used to find a feasible solution to a system of constraints with no objective function.

```
SolverProbTypeSet(prob, Solver_Type_Maximize, Problem_Type_NA);
```

Notice the problem handle (*prob*) is passed as the first argument to the function. Optionally, you could pass the problem type for the third argument. In this case, the problem type would be: *Problem_Type_OptLP* which indicates to the solver that the problem is linear. The SDK will automatically select the best engine to solve this type of problem. Within the SDK, the best engine to solve a linear problem is the LP/Quadratic Solver Engine.

There are five additional types of optimization problems: *Problem_Type_OptQP* (quadratic optimization problem), *Problem_Type_OptQCP* (quadratically constrained optimization problem), *Problem_Type_OptNLP* (nonlinear optimization problem), and *Problem_Type_OptNSP* (non-smooth optimization problem). If the problem type is set, the Solver SDK will choose the best engine available to solve that problem type. For example, if we were to set the problem type to *Problem_Type_OptNSP*, then the Evolutionary Engine would be automatically selected to solve the model.

Now we create a variable block, *Vars*, by calling *SolverVariableAdd()* and passing the variable type, *Variable_Type_Decision*, along with the number of variables, *nvars*.

```
SolverVariableAdd(prob, L"Vars", Variable_Type_Decision, nvars);
```

There are four types of variables: *Variable_Type_All* (specifies variables of all types), *Variable_Type_Decision* (decision variables), *Variable_Type_Uncertain* (for use with simulation and simulation optimization problems) and *Variable_Type_Recourse* (recourse variables used in simulation optimization).

Here we create a function block, *Constraints*, by calling *SolverFunctionAdd()* and passing the function type, *Function_Type_Constraint*, along with the number of constraints, *ncons*.

```
SolverFunctionAdd(prob, L"Constraints", Function_Type_Constraint,
ncons);
```

There are five types of functions for optimization problems: *Function_Type_All* (specifies variables of all types), *Function_Type_Objective* (objective function), *Function_Type_Constraint* (constraints), *Function_Type_Uncertain* (uncertain functions for use with simulation or simulation optimization) and *Function_Type_Chance* (refers to chance constraints for use with simulation optimization).

The line of code below creates the objective function block, *Objective*, with the function type, *Function_Type_Objective*, and the number of objectives, 1. Currently, the SDK only takes one objective so this last argument is currently always 1.

```
SolverFunctionAdd(prob, L"Objective", Function_Type_Objective,
1);
```

Next, we pass the upper bounds of the constraint right hand sides. Alternatively we could also have passed the constraint right hand sides using the `Relation` method (see the nonlinear example below).

```
SolverFcnUpperBoundSet(prob, L"Constraints", ub_constraints);
```

To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
SolverVarNonNegative(prob, L"Vars");
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass upper and lower variable bounds to the SDK.

Note that we use a dense matrix, and pass the elements by row.

```
SolverModArrayOrder(prob, Array_Order_ByRow);
```

3. **Call `AllLinear` and solve the linear model.** The two lines of code below call `SolverModAllLinearSet` to pass the objective coefficient matrix, *objcoeff*, and the constraint coefficient matrix, *concoeff*. By calling the *AllLinear* functions, the problem type is updated to "linear". As a result, the SDK will automatically choose the best linear engine available to solve the problem. Within the SDK, the LP/Quadratic engine is the best engine to solve a linear problem.

```
SolverModAllLinearSet(prob, Function_Type_Objective,  
Variable_Type_All , objcoeff);
```

```
SolverModAllLinearSet(prob, Function_Type_Constraint,  
Variable_Type_All, concoeff);
```

Solve the optimization problem by calling `SolverOptimize()`. Recall from the beginning of this subsection that *status* is the variable that will hold the optimization status and *objval* is a variable that will hold the final objective value. A pointer to each of these variables (*status* and *objval*) must be passed as the second and third arguments to `SolverOptimize()`, respectively.

```
SolverOptimize(prob, &status, &objval);
```

4. **Obtain and display the solution to the linear model.** We obtain and display the final variable values along with the optimization status, *status*, and the objective function, *objval*, on the C console.

```
SolverVarFinalValue(prob, L"Vars", x);
```

```
printf("Example 1\n");
```

```
printf("x[0] = %g\n", x[0]);
```

```
printf("x[1] = %g\n", x[1]);
```

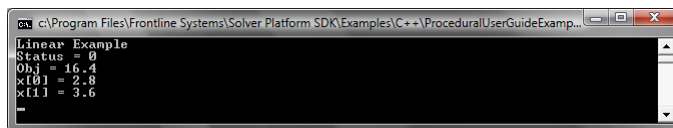
```
printf("Obj = %g\n", objval);
```

```
printf("Status = %d\n", status);
```

Finally, we release the problem from memory.

```
SolverProbFree(&prob);
```

5. **Build and run the application.** To compile and link this project, select `Build | Build UserGuideExample`. Then use `Debug | Start Debugging` to run the problem. An output window like the one below, should appear.



[Complete Code](#)

```

void linearExample()
{
    /*
    Example 1: Solves the LP model:
    Maximize    2 x1 +  3 x2
    Subj to     9 x1 +  6 x2 <= 54
               6 x1 +  7 x2 <= 42
               5 x1 + 10 x2 <= 50
    x1, x2 non-negative
    LP solution: x1 = 2.8, x2 = 3.6
    Objective = 16.4

    Matrix is passed rowwise, dense.
    */

    int nvars = 2, ncons = 3;    //2 dec vars, 3 constraints
    double objcoeff[] = { 2, 3 };
    double concoeff[] = { 9, 6, 6, 7, 5, 10 }; //row-wise
    double rhsvalue[] = { 54, 42, 50 };

    double objval;
    double x[2];
    HProblem prob;
    _Optimize_Status status;

    //creates problem
    SolverProblem(&prob);

    // tell Solver we want to maximize
    SolverProbTypeSet(prob, Solver_Type_Maximize,
    Problem_Type_NA);
    //Now create a variable block.
    SolverVariableAdd(prob, L"Vars", Variable_Type_Decision,
    nvars);
    // a Constraint block, and..
    SolverFunctionAdd(prob, L"Constraints",
    Function_Type_Constraint, ncons);
    // an Objective function block
    SolverFunctionAdd(prob, L"Objective", Function_Type_Objective,
    1);

    // pass the right hand sides of the constraints.
    SolverFcnUpperBoundSet(prob, L"Constraints", rhsvalue);

    // Variables have to be non-negative. Note that VarDecision
    // points to the "current" block of variables. In this example
    // we only have 1 block of variables.
    SolverVarNonNegative(prob, L"Vars");

    // Note that we use a dense matrix, and pass elements by row
    SolverModArrayOrder(prob, Array_Order_ByRow);

    // linear model; we pass the coefficients for the objective
    SolverModAllLinearSet(prob, Function_Type_Objective,
    Variable_Type_All , objcoeff);

    // linear model; we pass 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".

// Now we solve the model. Since we have not picked an engine
// to solve the model, Solver will determine best engine by
// checking the problem type.

SolverOptimize(prob, &status, &objval);

// We get the final value of the variables
SolverVarFinalValue(prob, L"Vars", x);

//Finally, we release the memory used, by freeing the problem.
SolverProbFree(&prob);

printf("Example 1\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;
}

```

Creating a Nonlinear Problem

1. **Basic Steps.** Follow the steps above in the section, Creating a New C Project. In this example, we will name the function that will hold the nonlinear example, *nonlinear()*.
2. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constraint ($x + y$) forces the total of the two variables to be equal to 1 and the second constraint ($x * y$) forces the product of the two variables to be zero or larger. Both final variable values must be non-negative (0 or larger).

```

Minimize  $x^2 + y^2$ 
Subject To:
 $x + y = 1$ 
 $x * y \geq 0$ 
 $x, y$  non-negative

```

To begin, the first few lines of code initialize the *prob*, *status*, and *objval* variables. *HProblem* *prob* initializes the problem handle, *prob*. The *status* variable will hold the optimization status and the *objval* variable will hold the final objective value.

```

HProblem prob;
_Optimize_Status status;
double objval;

```

These next two lines of code create and initialize two arrays of size 2 which is the number of variables. The *rhs* array holds the constraint right hand sides and the *x* array will hold the final variable values.

```

double rhs[] = {1, 0};

```

```
double x[2];
```

Next, we create an array of type *_Cons_Rel* (defined in the SDK) to hold the constraint "relation", *Cons_Rel_EQ* for equality and *Cons_Rel_GE* for greater than or equal to. There is one additional relation, *Cons_Rel_LE* for less than or equal to.

```
_Cons_Rel rel[] = {Cons_Rel_EQ, Cons_Rel_GE};
```

We now create a handle to the problem, *prob*. We will pass this handle as the first argument to the various Solver SDK routines.

```
SolverProblem(&prob);
```

Passing the problem type of OptNLP (nonlinear) by calling *SolverProbTypeSet()* will direct the Solver SDK to choose the best nonlinear engine available to solve the problem. Alternatively, we also could have set the GRG engine manually (in either the SDK Platform or the SDK Pro) by passing, *SolverProbEngineSet(prob, L"GRGNonlinear")*.

```
SolverProbTypeSet(prob,  
Solver_Type_Minimize, Problem_Type_OptNLP);
```

There are six types of optimization problems: *Problem_Type_OptLP* (linear), *Problem_Type_OptQP* (quadratic optimization problem), *Problem_Type_OptQCP* (quadratically constrained optimization problem), *Problem_Type_OptNLP* (nonlinear optimization problem), and *Problem_Type_OptNSP* (non-smooth optimization problem). If the problem type is set, the Solver SDK will choose the best engine available to solve that problem type. For example, if we were to set the problem type to *Problem_Type_OptNSP*, then the Evolutionary Engine would be automatically selected to solve the model.

SolverVariableAdd creates a variable block, *Vars*, passes the variable type, decision, and the number of variables, 2.

```
SolverVariableAdd(prob, L"Vars", Variable_Type_Decision, 2);
```

There are four types of variables: *Variable_Type_All* (specifies variables of all types), *Variable_Type_Decision* (decision variables), *Variable_Type_Uncertain* (for use with simulation and simulation/optimization problems) and *Variable_Type_Recourse* (recourse variables used in simulation optimization).

Next, we follow up with two *SolverFunctionAdd* function calls. The first *SolverFunctionAdd* creates a function block, passes the function type, constraint, and the number of constraints, 2. The second *SolverFunctionAdd* creates an objective function block, *Objective*, passes the function type, objective, and the number of objectives, which is currently always 1.

```
SolverFunctionAdd(prob, L"Constraints", Function_Type_Constraint,  
2);
```

```
SolverFunctionAdd(prob, L"Objective", Function_Type_Objective,  
1);
```

There are five types of functions for optimization problems: *Function_Type_All* (specifies variables of all types), *Function_Type_Objective* (objective function), *Function_Type_Constraint* (constraints), *Function_Type_Uncertain* (uncertain functions for use with simulation or simulation optimization) and *Function_Type_Chance* (refers to chance constraints for use with simulation optimization).

In this next line of code, we pass the constraint relations (*rel*) and the right hand sides (*rhs*).

```
SolverFcnRelation(prob, L"Constraints", rel, rhs);
```

3. **Create the nonlinear evaluator.** In a linear model or quadratic model, you supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the

values of the variables change. Therefore, we must supply an evaluator to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied. (It is possible to supply these derivatives directly to the SDK using an evaluator of type *Eval_Type_Gradient*. (For more information see the subheading, *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter, *Solver SDK Evaluators*.)

We create and pass the evaluator of type *Eval_Type_Function*, (as described above) to the SDK. The name of this evaluator is *NLPEvaluator*.

```
SolverEvaluatorAdd(prob, NLPEvaluator, NULL, Eval_Type_Function);
```

4. **Write the nonlinear evaluator.** Inside of the evaluator *NLPEvaluator*, we first create three arrays: *var*, *fcn*, and *obj*. The *var* array will hold the decision variable values. The *fcn* array will hold the computed constraint values, and the *obj* array will hold the computed objective function value. The *var* and *fcn* arrays are both of size two because there are two variables and two constraints. The *obj* array is of size 1 because there is only one objective function.

```
double var[2];
double fcn[2];
double obj[1];
```

Next, we ask for the current variable values from the SDK,

```
SolverVarValue(prob, L"Vars", var);
```

then store the computed constraint values in the *fcn* array and the computed objective value in the *obj* array.

```
fcn[0] = var[0] + var[1];
fcn[1] = var[0] * var[1];
obj[0] = var[0] * var[0] + var[1] * var[1];
```

We must pass the *fcn* and *obj* arrays back to the Solver SDK.

```
SolverFcnValueSet(prob, L"Constraints", fcn);
SolverFcnValueSet(prob, L"Objective", obj);
```

Finally, we tell the SDK to continue optimizing by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the functions (objective or constraints). The Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine (for solving nonsmooth problems) will disregard the current point and continue solving.

```
return Engine_Action_Continue;
```

Our evaluator is complete. We will now finish the code inside of the *nonlinear()* function.

5. **Solve the nonlinear problem.** Back inside the *nonlinear()* function, we call *SolverOptimize()* to solve the nonlinear optimization problem. As in the linear model above we must pass pointers to the *status* and *objval* variables as the second and third arguments to *SolverOptimize()*.

```
SolverOptimize(prob, &status, &objval);
```

6. **Obtain and display the solution to the linear model.** We obtain and display the final variable values as well as the optimization status and final objective function value.

We ask for the final variable values,

```
SolverVarFinalValue(prob, L"Vars", x);
```

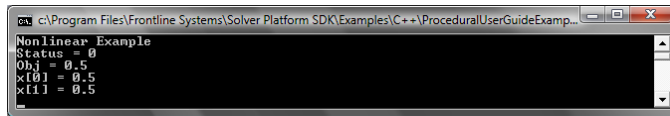
then display them, along with the optimization status and final objective value.

```
printf("Example 4\n");
printf("Status = %d\n", status);
printf("Obj = %g\n", objval);
printf("x[0] = %g\n", x[0]);
printf("x[1] = %g\n", x[1]);
```

Finally, we release the problem from memory.

```
SolverProbFree(&prob);
```

7. **Build and run the application.** To compile and link this project, select Build | Build userguideexample. Then use Debug | Start Debugging to run the program. An output window like the one below should appear.



Complete Code

```
_Engine_Action Evaluate(HProblem 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];
    // First, we must ask what the current variable values are.
    // Then, we store computed function values in the fcn array.
    SolverVarValue(prob, L"Vars", var);
    fcn[0] = var[0] + var[1];
    fcn[1] = var[0] * var[1];
    // We compute the objective
    obj[0] = var[0]*var[0] + var[1]*var[1];
    //We pass the function values to the Solver.
    SolverFcnValueSet(prob, L"Constraints", fcn);
    //We pass the objective value to the Solver.
    SolverFcnValueSet(prob, L"Objective", obj);
    return Engine_Action_Continue;
}

void nonlinearExample()
{
    /*
    Example C program calling the nonlinear Solver DLL.
    Solves the problem:

    Minimize X^2 + Y^2
    Subject to:
        X + Y = 1
        X * Y >= 0
    */
}
```



```

        (Solution is X = Y = 0.5, Objective = 0.5)
    */
    HProblem prob;
    _Optimize_Status status;
    double objval;
    double rhs[] = {1, 0};
    double x[2];
    _Cons_Rel rel[] = {Cons_Rel_EQ, Cons_Rel_GE};

    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, 2);
    SolverFunctionAdd(prob, L"Objective", Function_Type_Objective,
        1);

    SolverFcnRelation(prob, L"Constraints", rel, rhs);

    SolverEvaluatorAdd(prob, Evaluate, NULL, Eval_Type_Function);

    SolverOptimize(prob, &status, &objval);
    SolverVarFinalValue(prob, L"Vars", x);

    SolverProbFree(&prob);

    printf("Example 4\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);
}

```

Creating a Simulation Model

1. **Basic Steps.** Follow the steps in the chapter, *Creating a New C Project*. In this example, we will name the function that will hold the simulation example, *simulation()*.
2. **Create a simulation problem.** The simulation model that we will be solving in this example contains two uncertain variables, *x* and *y*. The uncertain variable, *x*, has a Uniform distribution with a lower parameter of 0 and an upper parameter of 2. The uncertain variable, *y*, has a Normal distribution with a mean and standard deviation equal to 1. In addition, there are two uncertain functions. The first uncertain function is the total of the two uncertain variables. The second is the product of the two uncertain variables.

```

X = PsiUniform(0,2)   and   Y = PsiNormal(1,1)
Function[1]: X + Y
Function[2]: X * Y

```

To begin, we create several variables, a variable, *status*, of type *_Simulate_Status*, a problem handle (*prob*), and three integer variables (*i*, *numvalues*, and *numerrors*). The *status* variable will hold the optimization status after the simulation has completed. The problem handle, *prob*, will be passed as an argument to the various Solver functions. The integer variable, *i*, will serve as a counter later on in the code. The integer variable *numvalues* will hold the number of successful trials. The integer variable *numerrors* will hold the number of trials that result in an error. The

number of trials equals *numvalues* plus *numerrors* (NumTrials = NumValue + NumErrors). For example, if you have 100 functions, and on trial 83, one of those functions returns an error, then all values for the 100 functions for trial 83 will be discarded. This means that when the Mean is computed for each function, the value for trial 83 will not be included.

```
_Simulate_Status status;
HProblem prob;
int i;
int numvalues, numerrors;
```

Next, we create several arrays which will hold various statistics once the simulation is complete. All but the last array is given a size of two, which is the number of variables. The *percentiles* array will hold all 100 percentiles for each uncertain variable.

```
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];
```

Next, we initialize the number of uncertain variables (*nvars*) and functions (*nfcns*).

```
int nvars = 2, nfcns = 2;
```

This example uses a correlation matrix which measures the degree to which the movements of the two uncertain variables are related. This is optional. The correlation matrix is:

1	.5
.5	1

The elements in the *correl* array below are in row-wise order.

```
double correl[] = { 1.0, 0.5, 0.5, 1.0 };
```

The next line of code creates a handle to the Problem, *prob*. We will pass this handle as the first argument to the various Solver SDK routines.

```
SolverProblem(&prob);
```

Next, we set the problem type to *Problem_Type_Simulate* by calling *SolverProbTypeSet()*. Notice the problem handle (*prob*) is passed as the first argument to the function. Optionally, you could pass the problem type for the third argument. In this case, the problem type would be *Problem_Type_SimCorrel* which indicates to the solver that we are solving a correlational simulation problem.

```
SolverProbTypeSet(prob, Solver_Type_Simulate, Problem_Type_NA);
```

There are two additional problem types for a simulation model, *Problem_Type_NA* which means that the problem type is unavailable, and *Problem_Type_InDep*, which indicates an independent simulation problem.

Now we create a variable block, *Vars*, by calling *SolverVariableAdd()* and passing the variable type, *Variable_Type_Uncertain*, along with the number of variables, *nvars*.

```
SolverVariableAdd(prob, L"Vars", Variable_Type_Uncertain, nvars);
```

There are four types of variables: *Variable_Type_All* (specifies variables of all types), *Variable_Type_Decision* (decision variables for use with optimization models), *Variable_Type_Uncertain* (for use with simulation and simulation optimization problems) and *Variable_Type_Recourse* (recourse variables used in simulation optimization).

Next, we create a function block, *Funcs*, by calling *SolverFunctionAdd()* and passing the function type, *Function_Type_Uncertain*, along with the number of uncertain functions, *nfcns*.

```
SolverFunctionAdd(prob, L"Funcs", Function_Type_Uncertain, nfcns);
```

There are five types of functions: *Function_Type_All* (specifies variables of all types), *Function_Type_Objective* (objective function), *Function_Type_Constraint* (constraints), *Function_Type_Uncertain* (uncertain functions for use with simulation or simulation optimization problems) and *Function_Type_Chance* (refers to chance constraints for use with simulation optimization problems).

The following two lines of code pass the distributions of the two uncertain variables, Uniform and Normal.

```
SolverModDistInit(prob, Dist_Var, 0, L"PsiUniform(0,2)" );
```

```
SolverModDistInit(prob, Dist_Var, 1, L"PsiNormal(1,1)" );
```

This next line of code, *SolverModArrayOrder()*, specifies that the correlation array is being passed "row wise". Alternatively, you could pass the coefficients "column wise" with *Array_Order_ByCol*.

```
SolverModArrayOrder(prob, Array_Order_ByRow);
```

Next, we pass the correlation matrix to the SDK using *SolverModDistCorrelationsSet()*.

```
SolverModDistCorrelationsSet(prob, 4, NULL, NULL, correl);
```

3. **Create the simulation evaluator.** When running a simulation, the evaluator of type *Eval_Type_Sample* is called so the user can calculate the uncertain functions, much like the evaluator of type *Eval_Type_Function* is called to compute the objective and constraints in a nonlinear optimization problem.

We create the evaluator of type *Eval_Type_Sample* (as described above) and pass the evaluator to the SDK. The name of the evaluator is *EvalSim()*.

```
SolverEvaluatorAdd(prob, EvalSim, NULL, Eval_Type_Sample);
```

4. **Write the simulation evaluator.** Inside of the evaluator, *evalSim()*, we first create two convenient pointers, *pVar*, which is a pointer to the uncertain variables, and *pFcn*, which is a pointer to the uncertain functions.

```
double pVar[2];
```

```
double pFcn[2];
```

First, we ask for the current uncertain variable values,

```
SolverVarValue(prob, L"Vars", pVar);
```

store the computed uncertain function values in the *pFcn* array.

```
pFcn[0] = pVar[0] + pVar[1];
```

```
pFcn[1] = pVar[0] * pVar[1];
```

and then pass the function values back to the SDK.

```
SolverFcnValueSet(prob, L"Funcs", pFcn);
```

Finally, we tell the SDK to continue simulating by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the uncertain functions. The SDK will mark the current trial as an error trial and continue with the remaining trials.

```
return Engine_Action_Continue;
```

Our evaluator is complete. We will now finish the code inside of the *simulation()* function.

5. **Solve the simulation model.** Back inside the *simulation()* function, we solve the simulation.

```
SolverSimulate(prob, & status);
```

6. **Display the various simulation statistics.** Recall the *status* variable which now holds the simulation status, the variables of type integer, *numvalues* and *numerrors*, and the arrays created to hold various statistics.

```
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*100+10]);
    printf("90th Percentile = %g\n", percentiles[i*10+90])
}
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*100+10]);
    printf("90th Percentile = %g\n", percentiles[i*100+90]);
}

```

Finally, we release the problem from memory.

```
SolverProbFree(&prob);
```

- 7. Build and run the application.** To compile and link this project, select Build | Build userguideexample. Then use Debug | Start Debugging to run the program. As output window like the one below should appear.

```

c:\Program Files\Frontline Systems\Solver Platform SDK\Examples\C++\Procedural\UserGuideExamp...
Simulation Example
Status = 0
Statistics on Uncertain Variables
Variable 0
Minimum = 0.000490638
Maximum = 1.99899
Mean = 1.00002
StdDev = 0.577655
Variance = 0.333685
Skewness = 7.0457e-006
Kurtosis = -1.20361
10th Percentile = 0.200457
90th Percentile = 1.79859
Variable 1
Minimum = -2.20767
Maximum = 4.16407
Mean = 1.00024
StdDev = 0.99959
Variance = 0.99918
Skewness = 0.00338060
Kurtosis = -0.0432753
10th Percentile = -0.278986
90th Percentile = 2.28117
Statistics on Uncertain Functions
Function 0
Minimum = -2.12806
Maximum = 6.07408
Mean = 2.00025
StdDev = 1.37534
Variance = 1.89157
Skewness = 0.0195104
Kurtosis = -0.295323
10th Percentile = 0.262144
90th Percentile = 3.76834
Function 1
Minimum = -1.95577
Maximum = 8.11787
Mean = 1.27933
StdDev = 1.5093
Variance = 2.27799
Skewness = 1.25973
Kurtosis = 1.47637
10th Percentile = -0.0906861
90th Percentile = 3.45416

```

Complete Code

```

_Engine_Action EvalSim(HProblem prob, void* refuser, _Eval_Type
evaltype )
{
    double pVar[2];
    double pFcn[2];
    int milliseconds, iterations, subproblems, localsols,

```

```

    funcevals, jacevals, hessevals;

    // First, we ask for the current uncertain variable values
    SolverVarValue(prob, L"Vars", pVar);

    SolverEngStat(prob, L"Monte Carlo Simulation",
        & milliseconds, & iterations, & subproblems,
        & localsols, & funcevals, & jacevals, & hessevals);

    printf("Evaluation x1 = %g, x2 = %g, Trial = %d\n", pVar[0],
        pVar[1], funcevals);

    // We compute the cons given the current variable values

    pFcn[0] = pVar[0] + pVar[1];    // X+Y
    pFcn[1] = pVar[0] * pVar[1];    // X*Y

    // We set the values of the function.

    SolverFcnValueSet(prob, L"Funcs", pFcn);

    return Engine_Action_Continue;
}

void example17()
{
    /*
    Example program calling the Monte Carlo Simulation engine.
    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
    */
    _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); //create 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 * 100 +
10]);
    printf("90th Percentile = %g\n", percentiles[i * 100 +
90]);
}

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 * 100 +
10]);
        printf("90th Percentile = %g\n", percentiles[i * 100 +
90]);
    }

    SolverProbFree(&prob);
}

```


Calling the Solver SDK from VB6 using the COM API

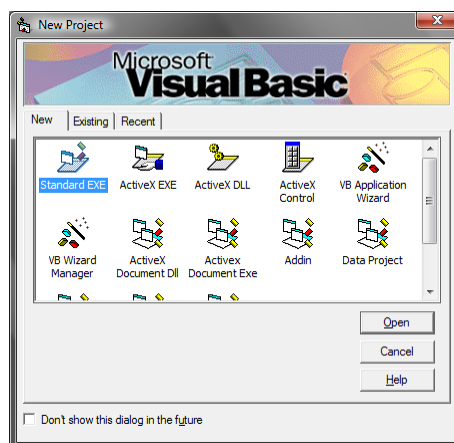
Introduction

This chapter details the steps required to build and run a simple linear, nonlinear, and simulation example using the Solver SDK from COM (Component Object Model). To demonstrate, we will be using Microsoft Visual Basic. A section explaining how to call the Solver SDK from VBA will follow.

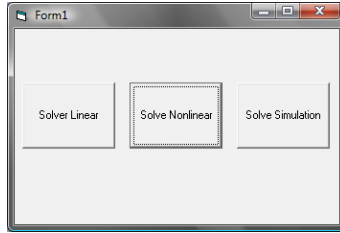
Creating a New VB Project

First, follow the steps in the "Installation" chapter, which will install all required SDK files, including Example files, into the Frontline Systems directory located at C:\Program Files on your hard disk.

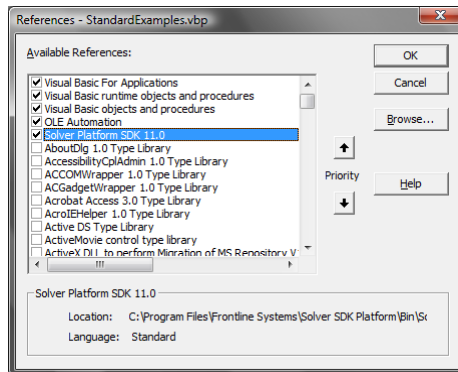
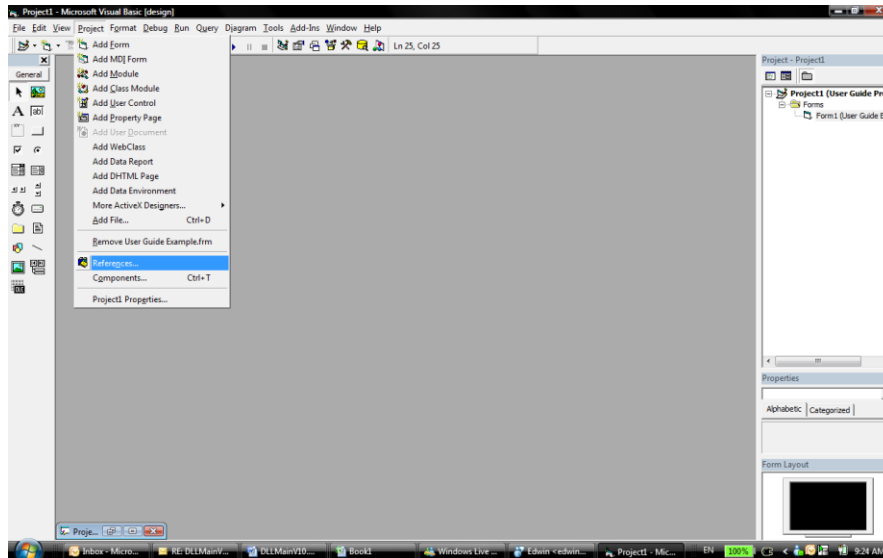
1. **Create a new project.** Start Microsoft Visual Basic 6.0 and select Standard EXE on the dialog shown below. Then click Open.



A new form will open. Construct a form of your choice using the Visual Basic Toolbox. In this example, the simple form shown below will be referenced.

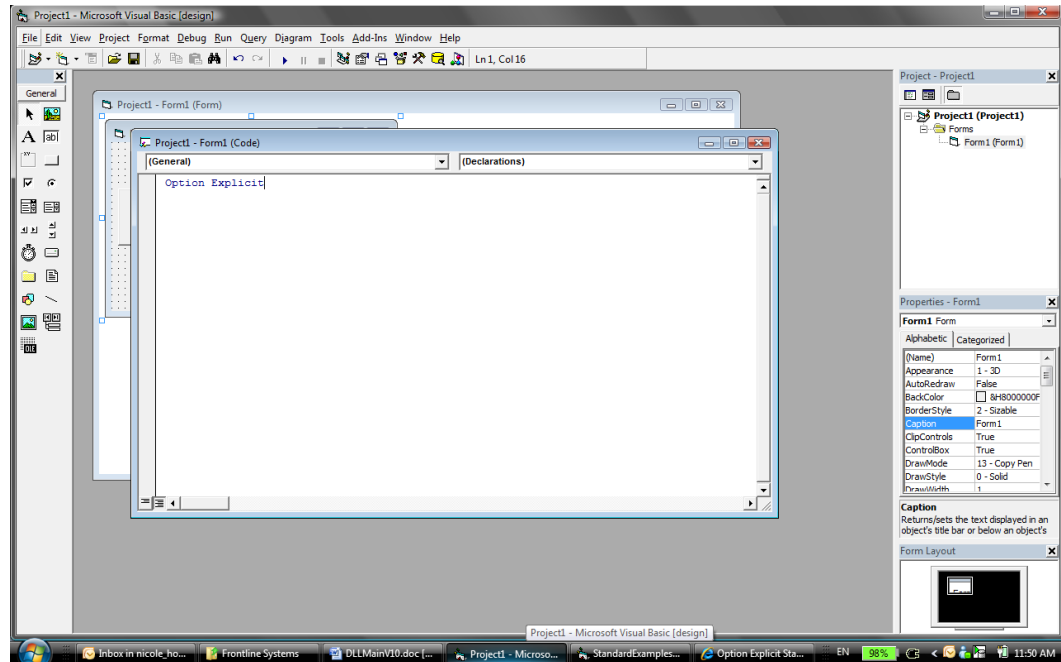


2. **Add a reference to the SDK.** In order to access the methods and properties in the Solver SDK, you must set a reference. To add the reference, click Project on the toolbar and then choose References from the menu. Scroll down to find **Solver Platform SDK V2025 Q4** and place a check in the box next to this item, as shown in the screenshots below.



3. **Add Option Explicit statement.** Right click Form1 in the Project Explorer window, choose **View Code** from the menu and type *Option Explicit* as shown in the screen print below.

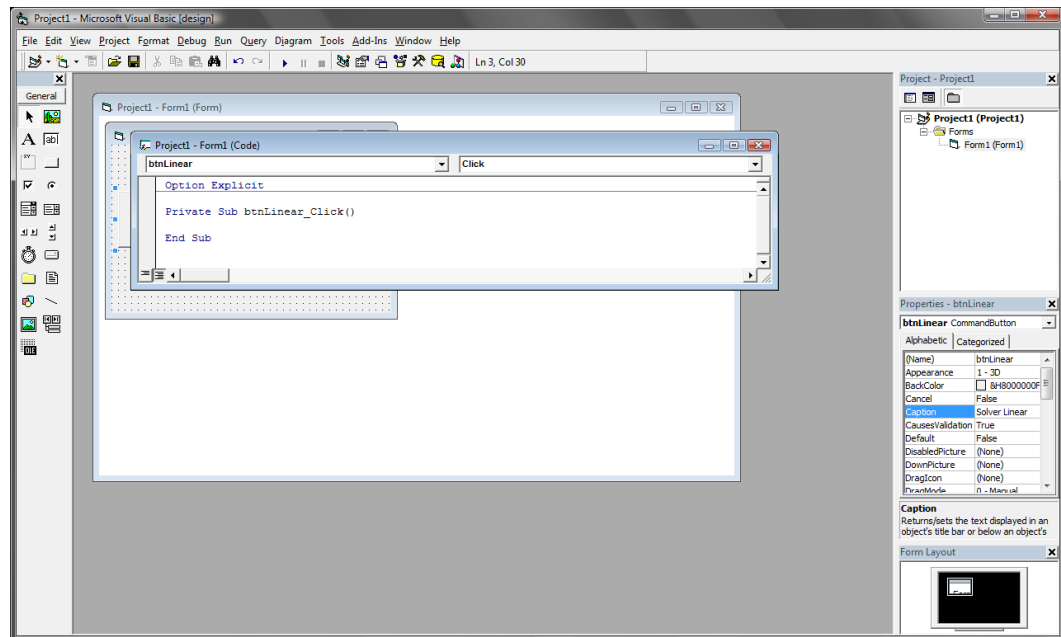
Using **Option Explicit** in our code will prohibit us from using undeclared variables and incorrectly typing the name of an existing variable in our code.



From here you can solve either a linear, nonlinear, or simulation model. For more details on each, please read on.

Creating a Linear Model

1. **First steps.** Follow the steps in the section, *Creating a New Visual Basic Project*, and then double click on the *Solve Linear* command button. This will open the *btnLinear_Click()* event which is where we will write our code to solve the linear optimization problem.



2. **Create the linear model.** In this section, we will solve a linear optimization problem with two variables (x_1 and x_2) and three constraints. Both final variable values must be zero or larger.

Maximize $2 x_1 + 3 x_2$

```

Subject To:
9 x1 + 6 x2 <= 54
6 x1 + 7 x2 <= 42
5 x1 + 10 x2 <= 50
x1, x2 non-negative

```

To begin, the example code first initializes two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model.

```

Dim nvars As Integer, ncons As Integer
nvars = 2
ncons = 3

```

The next few lines of code setup the objective and constraint coefficient matrices using two separate double matrices: *objcoeff* – for the objective coefficients and *concoeff* – for the constraint coefficients. Since there is only one objective, we pass the double matrix, *objcoeff*, of size *nvars* (number of variables) x 1 for the objective coefficients and the second double matrix, *concoeff*, of size *nvars* x *ncons* (number of constraints) for the constraint coefficients. Notice that we are using *NZBgn* and *NZEnd* when passing the constraint matrix (*concoeff*) as explained in the subsection *Passing Sparse Matrices* in the chapter, *Designing Your Application*. As a result, the SDK Solver will not update the *concoeff* DoubleMatrix object until *NZEnd* is executed.

```

Dim objcoeff as New DoubleMatrix
With objcoeff
    .Create Rows:=1, Columns:=2
    .Value(0,0) = 2
    .Value(0,1) = 3
End With
Dim concoeff as New DoubleMatrix
concoeff.NZBgn
With concoeff
    .Create Rows:=3, Columns:=2
    .Value(0,0) = 9
    .Value(0,1) = 6
    .Value(1,0) = 6
    .Value(1,1) = 7
    .Value(2,0) = 5
    .Value(2,1) = 10
End With
Concoeff.NZEnd

```

Next, we create an array to hold the upper bounds of the constraint right hand sides.

```

Dim ub_constraints(2) as Double
ub_constraints(0) = 54
ub_constraints(1) = 42
ub_constraints(2) = 50

```

This next line of code (*Dim prob as New SolverPlatform.Problem*) creates an instance of the Problem class. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem.

```
Dim prob as New SolverPlatform.Problem
```

These two lines of code pass the number of decision variables (*prob.VarDecision.Size = nvars*) and constraints (*prob.FcnConstraint.Size = ncons*) that are included in the model.

```
prob.VarDecision.Size = nvars
prob.FcnConstraint.Size = ncons
```

To tell the SDK to maximize the model, we use the line of code below. Alternatively, *Solver_Type_Minimize* will result in the model being minimized.

```
prob.Solver.SolverType = Solver_Type_Maximize
```

We pass the *ub_constraints* array to the SDK. Recall that this array holds the upper bounds of the constraint right hand sides.

```
prob.FcnConstraint.UpperBound = ub_constraints
```

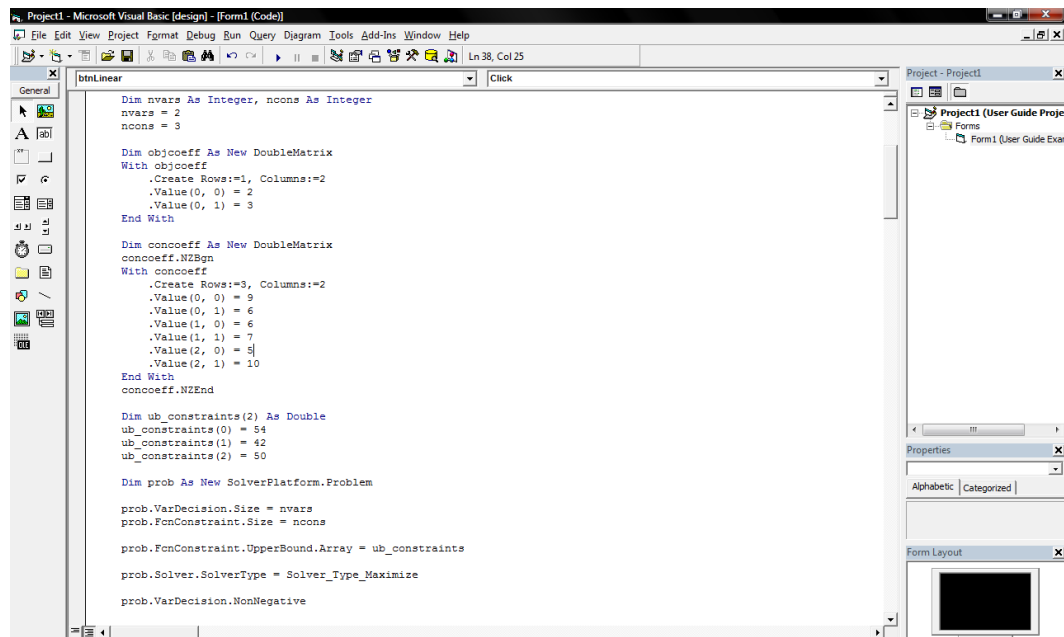
To pass a lower bound of 0 on any variable that does not currently have a lower bound, call:

```
prob.VarDecision.NonNegative
```

In this example, we have not provided lower bounds for either of our variables. This line of code effectively adds a lower bound of 0 for both. See the nonlinear problem below for example code on how to pass upper and lower variable bounds to the SDK.

To tell the SDK to maximize the model, we use the line of code below. Alternatively, *Solver_Type_Minimize* will minimize the problem.

```
prob.Solver.SolverType = Solver_Type_Maximize
```



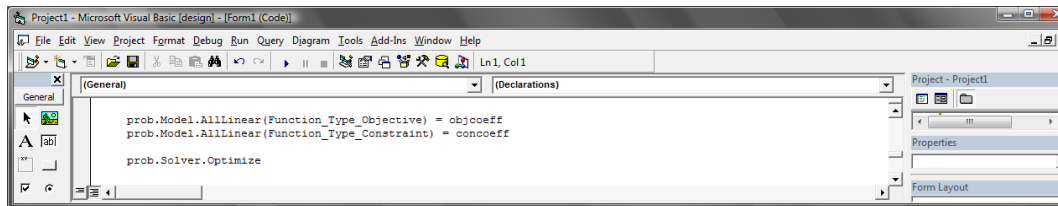
3. **Solve the linear model.** The line of code, *prob.Model.AllLinear(Function_Type_Objective)*, passes the objective coefficient matrix and *prob.Model.AllLinear(Function_Type_Constraint)* passes the constraint coefficient matrix setup above. Calling *prob.Model.AllLinear* sets the

problem type to "linear". Consequently, the SDK will automatically choose the best linear engine available to solve the problem. If no additional Large Scale "plug in" engines have been added, then the LP/Quadratic engine will be selected in the Solver SDK Platform and the LP Simplex engine will be selected in the Solver SDK Pro.

```
prob.Model.AllLinear(Function_Type_Objective) = objcoeff;
prob.Model.AllLinear(Function_Type_Constraint) = concoeff;
```

Solve the optimization model by calling:

```
problem.Solver.Optimize
```



4. Obtain and display the solution to the linear model. We obtain the solution status,

```
Dim status as Optimize_Status
```

```
status = prob.Solver.OptimizeStatus
```

along with the final objective function value,

```
Dim objValue() as Double
```

```
objValue = prob.FcnObjective.FinalValue.Array
```

and final decision variable values.

```
Dim varvalue() as Double
```

```
varvalue = prob.VarDecision.FinalValue.Array
```

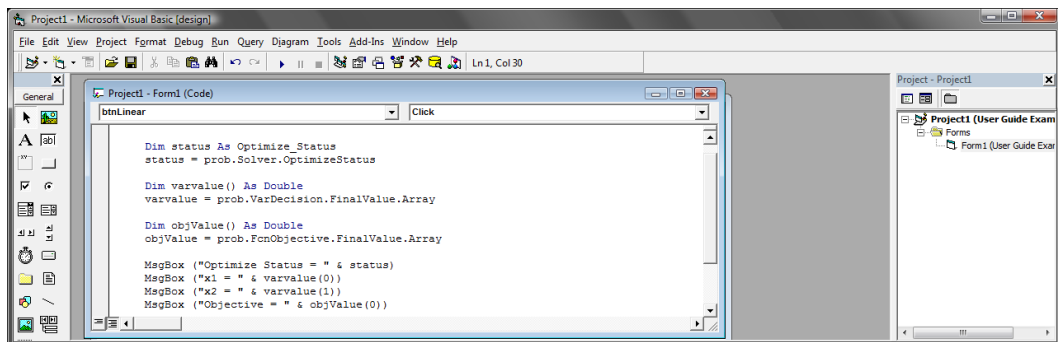
Then we use message boxes to display the answer.

```
MsgBox ("Optimize Status = " & status)
```

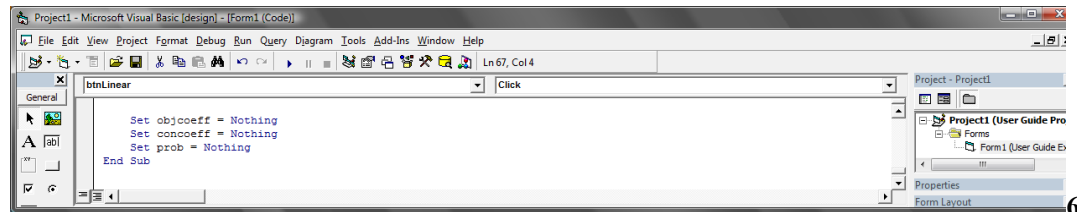
```
MsgBox ("x1 = " & varvalue(0))
```

```
MsgBox ("x2 = " & varvalue(1))
```

```
MsgBox ("Obj = " & objValue(0))
```



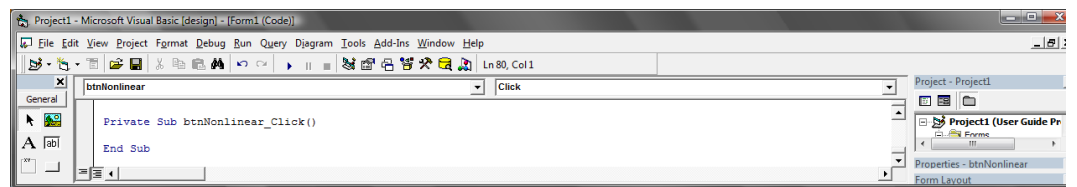
5. Release the problem. Set both double matrices, *objcoeff* and *concoeff*, and the problem, *prob*, to *Nothing* to release the problem from memory.



Running the example. To run the example, go to Run|Start.

Creating a Nonlinear Model

1. **First steps.** Follow the steps above in the section, *Creating a New VB Project*, and then double click on the *Solve Nonlinear* button. Doing so will open the *btnNonlinear_Click* event which is where we will write our code to solve the nonlinear optimization problem.



2. **Create the nonlinear problem.** This section demonstrates how to solve a simple nonlinear model with two variables (x and y) and two constraints. The first constraint ($x + y$) forces the total of the two variables equal to 1 and the second constraint ($x * y$) forces the product of the two variables to be greater than zero. Both final variable values must be zero or greater.

```
Minimize  $x^2 + y^2$ 
Subject to:
 $x + y = 1$ 
 $x * y \geq 0$ 
 $x, y$  are non-negative
```

First, we initialize the number of variables (*nvars*) and the number of constraints (*ncons*).

```
Dim nvars As Integer, ncons As Integer
nvars = 2
ncons = 2
```

Next, we create two arrays of size *ncons* (2 constraints) to hold the upper and lower bounds on the constraint right hand sides. Notice that the upper and lower bound of the first constraint is 1 to denote an equality constraint. 1E+30 sets the upper bound of the second constraint to positive infinity.

```
Dim ub_constraints(1) As Double
ub_constraints(0) = 1#
ub_constraints(1) = 1E+30
Dim lb_constraints As Double
lb_constraints(0) = 1#
ub_constraints(1) = 0
```

The next two lines of code create two arrays, *lb* and *ub*, to hold the lower and upper variable bounds, respectively.

```
Dim lb(1) As Double
lb(0) = 0
```

```

lb(1) = 0
Dim ub(1) as Double
ub(0) = 1
ub(1) = 1

```

This next line of code (*Dim prob as New SolverPlatform*) creates an instance of the Problem class. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem.

```
Dim prob as New SolverPlatform.Problem
```

Here we pass the constraint upper and lower bounds to the SDK,

```

prob.FcnConstraint.UpperBound.Array = ub_constraints
prob.FcnConstraint.LowerBound.Array = lb_constraints

```

following up with the variable upper and lower bounds.

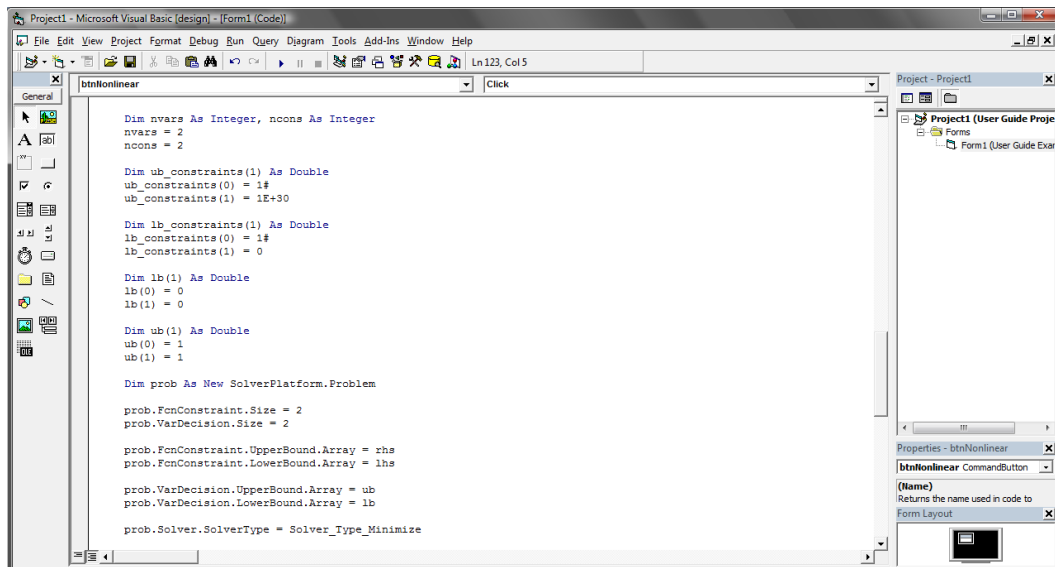
```

prob.VarDecision.UpperBound.Array = ub_constraint
prob.VarDecision.LowerBound.Array = lb_constraint

```

To tell the SDK to maximize the model, we use the line of code below. Alternatively, *Solver_Type_Minimize* will result in the model being minimized.

```
prob.Solver.SolverType = Solver_Type.Minimize
```

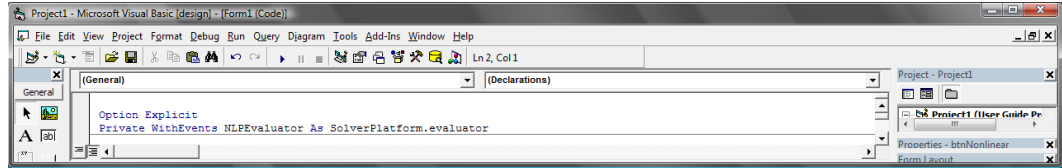


3. **Create the nonlinear evaluator.** In a linear model or quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator of type *Eval_Type.Function* to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied. (It is possible to pass these derivatives directly to the SDK using the evaluator of type *Eval_Type.Gradient*. For more information, see the subheading, *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter *Solver SDK Evaluators*.)

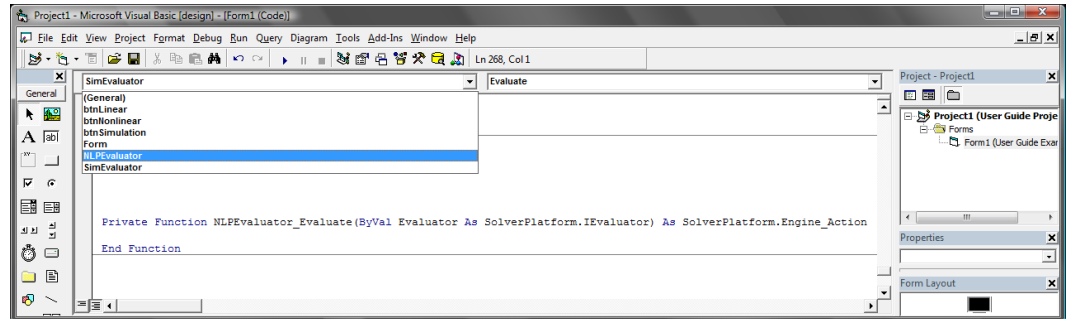
We create and pass the evaluator (as described above) to the SDK.

```
Set NLPEvaluator = New SolverPlatform.evaluator  
prob.Evaluators.Item(Eval_Type.Function) = NLPEvaluator
```

4. **Write the nonlinear evaluator.** Before we write our evaluator, we must declare a custom event handler. We declare this event handler directly underneath *Option Explicit* as shown in the screenprint below.



Once this line of code is in place, we can click the down arrow next to **(General)** and select **NLPEvaluator** from the menu as shown in the screenshot below. Visual Basic will automatically create our nonlinear evaluator, *NLPEvaluator_Evaluate*.



Inside of our *NLPEvaluator_Evaluate* event handler, we first create a convenient pointer, *p*, to the Problem,

```
Dim p As Problem  
p = evaluator.Problem
```

and another pointer, *pVar*, to the variables.

```
Dim pVar(1) as Double  
pVar = p.VarDecision.Value.Array
```

Next, we obtain a pointer to the constraint functions and calculate the constraints.

```
Dim pFcn(1) as Double  
pFcn = p.FcnConstraint.Value.Array
```

Here we pass the first constraint, $x + y$,

```
pFcn(0) = pVar(0) + pVar(1)
```

and here we pass the second, $x * y$.

```
pFcn(1) = pVar(0) * pVar(1)
```

Then we compute the objective function, $x * x + y * y$.

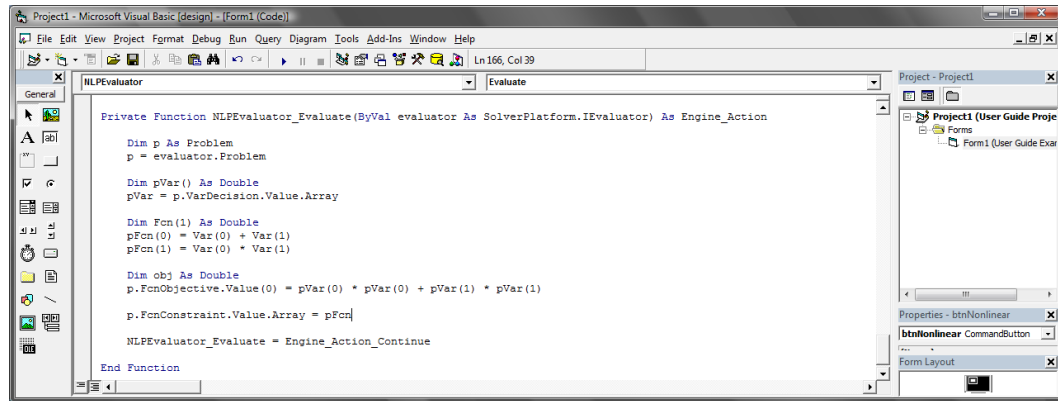
```
p.FcnObjective.Value(0) = pVar(0) * pVar(0) + pVar(1) * pVar(1)
```

In C#, we must pass the constraint values back to the SDK.

```
p.FcnConstraint.Value.Array = pFcn
```

Finally, we tell the SDK to continue optimizing by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the functions (objective or constraints). The Large Scale GRG engine (within the Solver SDK Platform), the GRG Nonlinear engine (within the Solver SDK Pro), and the Evolutionary engine (for solving nonsmooth problems) will disregard the current point and continue solving.

```
NLPEvaluator_Evaluate = Engine_Action_Continue
```



An alternative way to write this evaluator without using the *problem*, *pVar* and *pFcn* pointers is below.

```
Private Function NLPEvaluator_Evaluate(ByVal evaluator As
SolverPlatform.IEvaluator) As Engine_Action
    evaluator.Problem.FcnConstraint.Value(0) =
        evaluator.Problem.VarDecision.Value(0) +
        evaluator.Problem.VarDecision.Value(1)
    evaluator.Problem.FcnConstraint.Value(1) =
        evaluator.Problem.VarDecision.Value(0) *
        evaluator.Problem.VarDecision.Value(1)
    evaluator.Problem.FcnObjective.Value(0) =
        evaluator.Problem.VarDecision.Value(0) *
        evaluator.Problem.VarDecision.Value(0) +
        evaluator.Problem.VarDecision.Value(1) *
        evaluator.Problem.VarDecision.Value(1)
    NLPEvaluator_Evaluate = Engine_Action_Continue
End Function
```

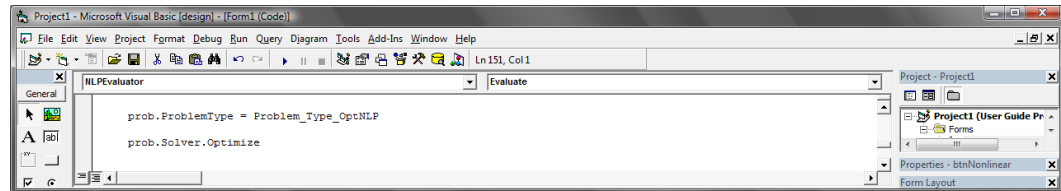
Our evaluator is complete. We will now finish the code inside of the *button_NonlinearClick* event.

5. **Solve the nonlinear problem.** Back inside the *button_NonlinearClick* event, we provide the problem type as nonlinear which directs the SDK to select the GRG engine to solve the model. Alternatively, we also could have set the GRG engine manually (in either the SDK Platform or the SDK Pro) by calling:
prob.Engine = prob.Engines(prob.engine.GRGName).

```
prob.ProblemType = Problem_Type.OptNLP
```

To solve the optimization problem, we call:

```
prob.Solver.Optimize
```

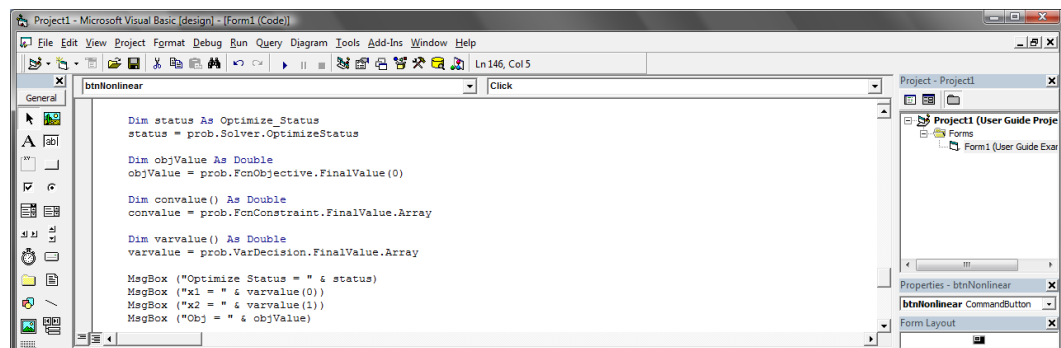


- Obtain the solution status and final answer.** We obtain the solution status, along with the final objective and variable values,

```
Dim convalue() As Double
convalue = prob.FcnConstraint.FinalValue.Array
Dim varvalue() As Double
varvalue = prob.VarDecision.FinalValue.Array
Dim objValue As Double
objValue = prob.FcnObjective.FinalValue(0)
Dim status As Optimize_Status
status = prob.Solver.OptimizeStatus
```

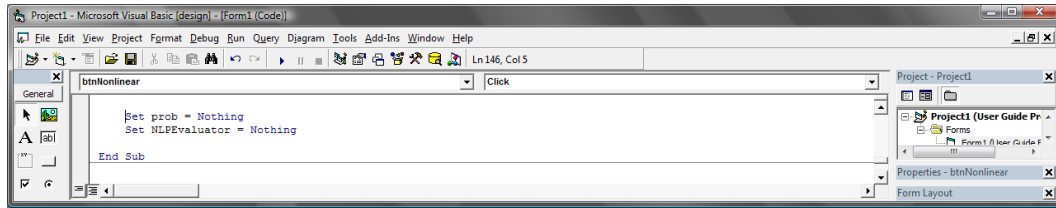
and display the answer with multiple message boxes.

```
MsgBox ("Optimize Status = " & status)
MsgBox ("x1 = " & varvalue(0))
MsgBox ("x2 = " & varvalue(1))
MsgBox ("Obj = " & objValue)
```



- Release the problem.** Set the evaluator, *NLPEvaluator*, and the problem, *prob*, to *Nothing* to release the problem from memory.

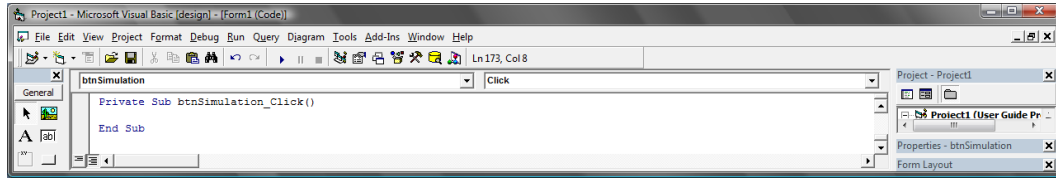
```
Set prob = Nothing
Set NLPEvaluator = Nothing
End Sub
```



8. **Running the example.** To run the example, go to Run|Start .

Creating a Simulation Model

1. **Basic steps.** Follow the steps in the chapter, *Creating a New VB Project*, and then double click on the *Solve Simulation* button. Doing so will open the *btnSimulation_Click* event which is where we will write our code to solve the simulation problem.



2. **Create a simulation problem.** The simulation model that we will be solving in this example contains two uncertain variables, X and Y. The uncertain variable X has a uniform distribution with a lower parameter of 0 and an upper parameter of 2. The uncertain variable Y has a Normal distribution with a mean and standard deviation equal to 1. In addition, there are two uncertain functions. The first uncertain function is the total of the two uncertain variables (X + Y). The second is the product of the two uncertain variables (X * Y).

`X = PsiUniform(0,2)    and    Y = PsiNormal(1,1)`

`Function[1]: X + Y`

`Function[2]: X * Y`

First, we initialize the number of uncertain variables (*nvars*) and functions (*nfcns*).

```
Dim nvars As Integer
    Dim nfcns As Integer
    nvars = 2
    nfcns = 2
```

This example uses a correlation matrix which measures the degree to which the movements of the two uncertain variables are related. This is optional. The correlation matrix is:

1	.5
.5	1

The elements are being passed row-wise.

```
Dim correl(3) As Double
correl(0) = 1#
correl(1) = 0.5
correl(2) = 0.5
correl(3) = 1#
```

This next line of code (*Dim prob as New SolverPlatform*) creates an instance of the Problem class. This constructor will not create any variable or function blocks or pass any additional information, such as whether you are solving a minimization optimization, maximization optimization or simulation problem.

```
Dim prob as New SolverPlatform.Problem
```

Below, we pass the number of uncertain variables (*nvars*) and uncertain functions (*nfcns*) to the SDK.

```
prob.VarUncertain.Size = nvars
```

```
prob.FcnUncertain.Size = nfcns
```

Next, we pass the distributions of the two uncertain variables, Uniform and Normal. Since Visual Basic 6.0 does not support constructors, we must use the *.Init* property to set the properties and methods of the *Distribution* class.

```
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
```

These next few lines of code create and pass the correlation matrix. The first line of code (*Dim matDistCor as New SolverPlatform.DoubleMatrix*) creates a new double matrix, *matDistCor*. The second line of code indicates to the SDK that this matrix will be passed "row wise". The third line of code initializes the Double Matrix by passing the number of variables (*nvars*) for the rows and columns and then passing the correlation coefficients from the *correl* array. The last line of code passes the *matDistCor* Double Matrix to the SDK.

```
Dim matDistCor As New SolverPlatform.DoubleMatrix
```

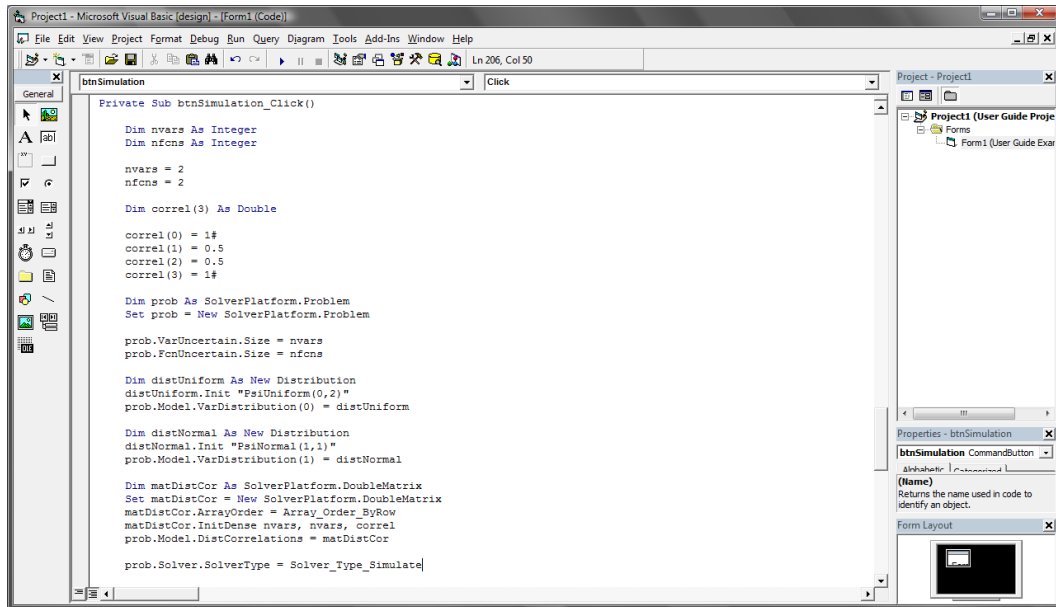
```
matDistCor.ArrayOrder = Array_Order_ByRow
```

```
matDistCor.InitDense nvars, nvars, correl
```

```
prob.Model.DistCorrelations = matDistCor
```

Next, we tell the SDK to solve a simulation problem.

```
prob.Solver.SolverType = Solver_Type_Simulate
```

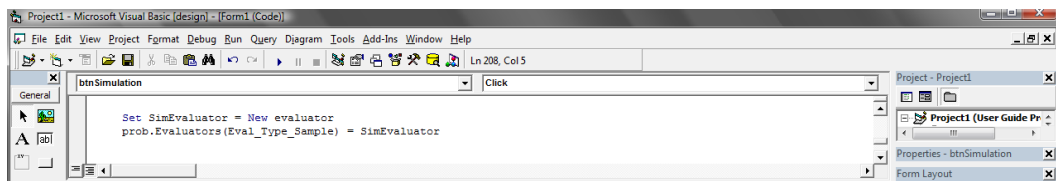


3. **Create the simulation evaluator.** When running a simulation, the evaluator of type *Eval_Type_Sample* is called so the user can calculate the uncertain functions, much like the evaluator of type *Eval_Type_Function* is called to compute the objective and constraints in a nonlinear optimization problem.

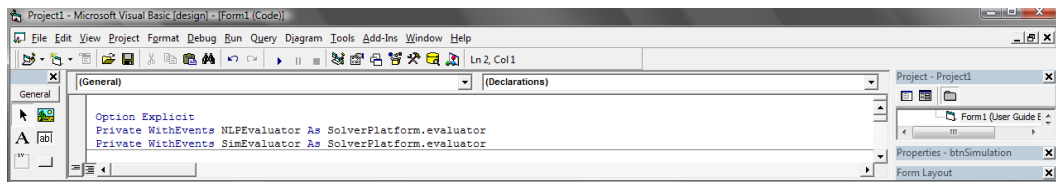
We create the *Eval_Type_Sample* evaluator (as described above) and pass the evaluator to the SDK.

```
Set SimEvaluator = New evaluator
```

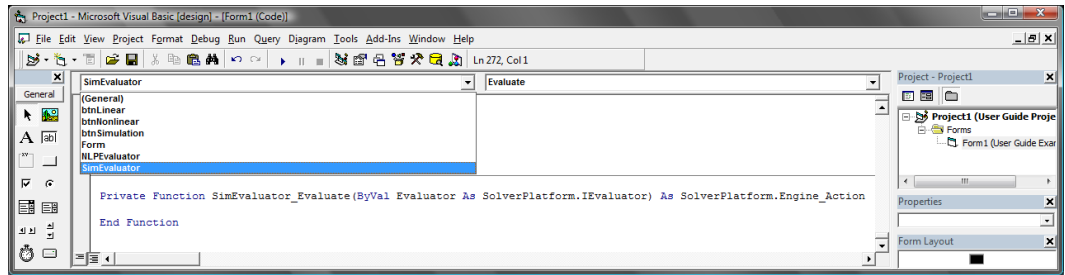
```
prob.Evaluators(Eval_Type_Sample) = SimEvaluator
```



4. **Write the simulation evaluator.** Before we write our evaluator, we must declare a custom event handler. We declare this event handler directly underneath *Option Explicit* as shown in the screenshot below.



Once this line of code is in place, we can click the down arrow next to **(General)** and select **SimEvaluator** from the menu as shown in the screenshot below. Visual Basic will automatically create our simulation evaluator, *SimEvaluator_Evaluate*.



Inside of our *Simulation_OnEvaluate* event, we first create a convenient pointer, *pVar*, to the Decision Variables.

```
Dim Var(1) As Double
```

```
Var = evaluator.Problem.VarUncertain.Value.Array
```

Next, we create an array, *pFcn*, to hold the computed uncertain function values

```
Dim pFcn(1) as Double
```

and then compute the values given the current uncertain variable values.

```
pFcn(0) = pVar(0) + pVar(1);
```

```
pFcn(1) = pVar(0) * pVar(1);
```

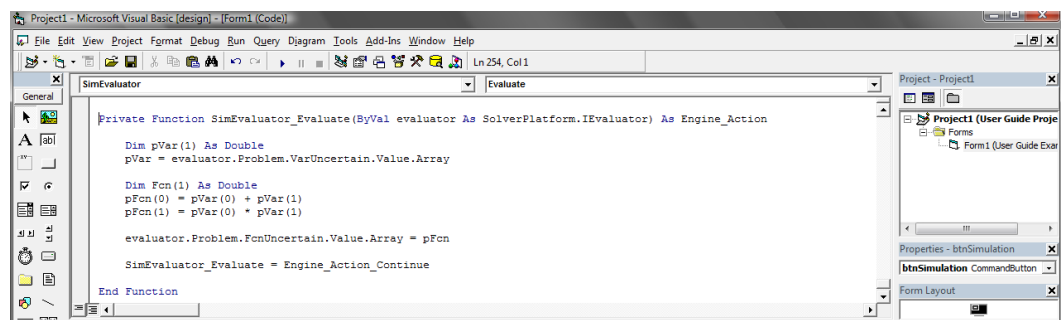
The line of code below passes the constraint values back to the SDK.

```
evaluator.Problem.FcnUncertain.Value.Array = pFcn
```

Finally, we tell the SDK to continue simulating.

```
SimEvaluator_Evaluate = Engine_Action_Continue
```

Finally, we tell the SDK to continue simulating by returning *Engine_Action_Continue*. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the evaluator of type *Eval_Type_Function* a second time which can be useful if you need to inspect the variable values. *Engine_Action_Stop* will indicate to the SDK that an error has occurred during the calculation of the uncertain functions. The SDK will mark the current trial as an error trial and continue with the remaining trials.



An alternative way to write this evaluator without using the *pVar* and *pFcn* pointers is below.

```
Private Function SimEvaluator_Evaluate(ByVal evaluator As SolverPlatform.IEvaluator) As Engine_Action
```

```
evaluator.Problem.FcnUncertain.Value(0) =
```

```
    evaluator.Problem.VarUncertain.Value(0) +
```

```
    evaluator.Problem.VarUncertain.Value(1)
```

```
evaluator.Problem.FcnUncertain.Value(1) =
```

```

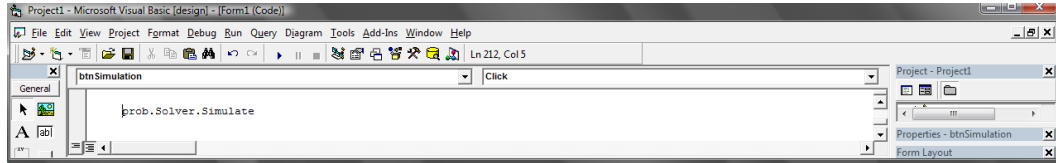
        evaluator.Problem.VarUncertain.Value(0) *
        evaluator.Problem.VarUncertain.Value(1)
    SimEvaluator_Evaluate = Engine_Action_Continue
End Function

```

Our evaluator is complete. We will now finish the code inside of the *button_SimulationClick* event.

5. **Solve the simulation problem.** Back inside the *btn_SimulationClick* event, we call the following line of code to run the simulation.

```
prob.Solver.Simulate();
```



6. **Display various simulation statistics.** We obtain the simulation status and various uncertain variable and uncertain function statistics using message boxes.

```
MsgBox ("Status = " & prob.Solver.SimulateStatus)
```

Display statistics on the two uncertain variables.

```
Dim i As Integer
```

```
For i = 0 To nvars - 1
```

```
    MsgBox ("Variable " & i)
```

```
    MsgBox ("Minimum = " &
    prob.VarUncertain.Statistics.Minimum(i))
```

```
    MsgBox ("Maximum = " &
    prob.VarUncertain.Statistics.Maximum(i))
```

```
    MsgBox ("Mean= " & prob.VarUncertain.Statistics.Mean(i))
```

```
    MsgBox ("StdDev = " & prob.VarUncertain.Statistics.StdDev(i))
```

```
    MsgBox ("Variance = " &
    prob.VarUncertain.Statistics.Variance(i))
```

```
    MsgBox ("Skewness = " &
    prob.VarUncertain.Statistics.Skewness(i))
```

```
    MsgBox ("Kurtosis = " &
    prob.VarUncertain.Statistics.Kurtosis(i))
```

```
    MsgBox ("10th Percentile = " & prob.VarUncertain.Percentiles(i,
    10))
```

```
    MsgBox ("90th Percentile = " & prob.VarUncertain.Percentiles(i,
    90))
```

```
Next
```

Display statistics on the two uncertain functions.

```
MsgBox ("Statistics on Uncertain Functions")
```

```
For i = 0 To nfcns - 1
```

```
    MsgBox ("Function " & i)
```



```

MsgBox ("Minimum = " &
prob.FcnUncertain.Statistics.Minimum(i))

MsgBox ("Maximum = " &
prob.FcnUncertain.Statistics.Maximum(i))

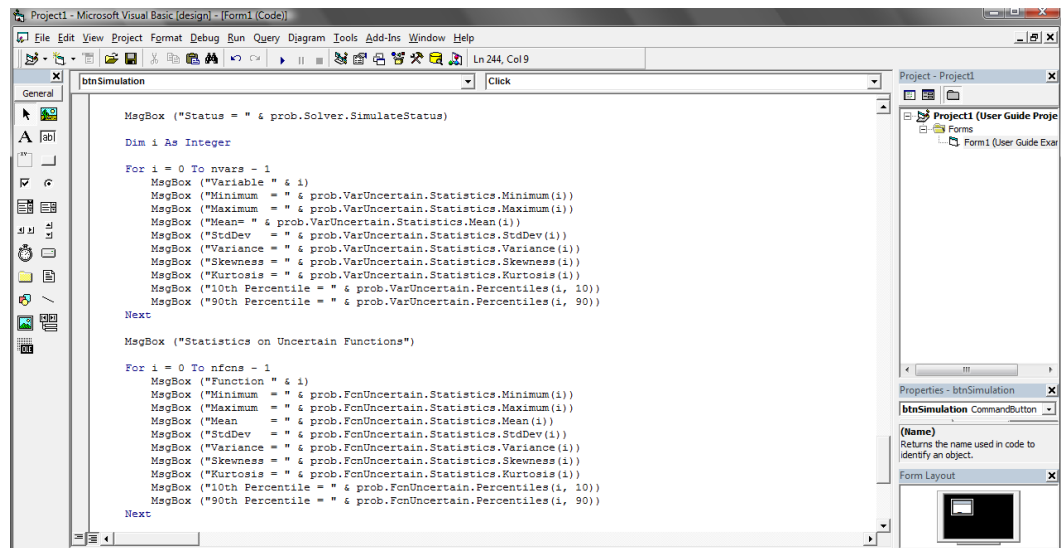
MsgBox ("Mean = " & prob.FcnUncertain.Statistics.Mean(i))
MsgBox ("StdDev = " & prob.FcnUncertain.Statistics.StdDev(i))
MsgBox ("Variance = " &
prob.FcnUncertain.Statistics.Variance(i))
MsgBox ("Skewness = " &
prob.FcnUncertain.Statistics.Skewness(i))
MsgBox ("Kurtosis = " &
prob.FcnUncertain.Statistics.Kurtosis(i))

MsgBox ("10th Percentile = " & prob.FcnUncertain.Percentiles(i,
10))

MsgBox ("90th Percentile = " & prob.FcnUncertain.Percentiles(i,
90))

```

Next



- Release the problem.** Set the double matrix, *matdistcor*, the evaluator, *SimEvaluator*, both distributions, *distuniform* and *distnormal*, and the problem, *prob*, to *Nothing* to release the problem from memory.

```

Set SimEvaluator = Nothing

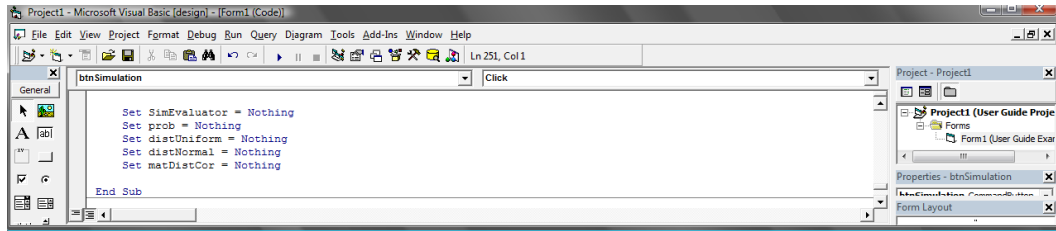
Set prob = Nothing

Set distUniform = Nothing

Set distNormal = Nothing

Set matdistcor = Nothing

```



8. **Running the example.** To run the example, go to Run|Start.

Calling Solver SDK from VBA

It is also possible to call the Solver SDK through Microsoft Excel's Visual Basic for Applications (VBA). The steps for creating and solving a linear, nonlinear, or simulation problem in VBA will be the same as above, with one exception. VBA does not support event handlers inside of "regular" modules. Consequently, you must include all callback functions (evaluators) in a class module. To insert a class module, go to Insert | Class Module in VBA. To set a reference to the SDK in VBA, go to Tools | References.

Visual Basic 6 Source Code

The source code file FormMain.frm (installed into C:\Program Files\Frontline Systems\Solver SDK Platform\SolverSDK\Examples\VB6\Standard Examples) contains over thirty examples of how to solve linear, integer, quadratic, nonlinear, nonsmooth, Monte Carlo simulation and simulation optimization programming problems. The list below gives a brief explanation of each example problem.

You can easily open and run each example by opening Visual Basic V6.0, clicking File | Open Project, browsing to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples\VB6\Standard Examples and opening the file, StandardExamples.vbp on the Existing tab. To run the examples, go to Run | Start in Visual Studio.

1. Linear Programming Example (Dense Matrix)
 - Simple two-variable LP problem which passes a dense constraint matrix "row wise".
2. Integer Linear Programming Example (Dense Matrix)
 - Simple MIP problem, identical to the previous LP problem except that the decision variables are required to have integer values and the coefficients are passed column wise.
3. Markowitz Optimization with QP Solver
 - Quadratic programming problem, using the classic Markowitz method to find an efficient portfolio of five securities.
4. Nonlinear programming with the GRG engine
 - Simple two-variable nonlinear optimization problem, which defines an evaluator of type *Eval_Type_Function* evaluator to compute the problem functions.
5. Non-Linear Programming with Evolutionary engine
 - Finds the global optimum of a classic two-variable problem, the Branin function, which has three local optima.
6. Non-Linear Programming with the GRG engine using Gradients
 - Identical to example 5, except that an evaluator of type *Eval_Type_Gradient* is also defined, to give the SDK faster and more accurate partial derivatives of the problem functions.
7. Markowitz Optimization with SOCP Solver

- Identical to example 3, except that the *AllGradDepend* method is used to pass *objtype* and *matvaltype* arrays to the SOCP engine.
8. Quadratically Constrained Programming with SOCP Solver
 - Similar to examples 3 and 7, except that the objective maximizes the return.
 9. Exception Handling Example
 - Illustrates how to "catch" an exception.
 10. Reading an MPS, OSiL or Excel File
 - Example shows how a problem can be loaded from a text file.
 11. Markowitz Optimization using Hessians
 - Identical to examples 3 and 6, except that the evaluator of type *Eval_Type_Hessian* is defined to give the SDK faster and more accurate second order derivatives of the problem functions.
 12. Second order Cone programming Example
 - Simple example of a problem with two cone constraints.
 13. Integer Programming with All-Different Constraint
 - Similar to example 1 and 2 with an added All-Different Constraint.
 14. Linear Programming Infeasibility Diagnosis Example
 - Attempts to solve a linear problem that is infeasible, then it uses the IIS finder to diagnose the infeasibility.
 15. Illustrating Multiple Solutions
 - Example illustrates how a user can access all incumbent solutions that the SDK finds during the optimization.
 16. Linear Programming with Sparse Matrix
 - Example illustrates how to pass a sparse matrix.
 17. Simulation Example
 - Example program calling the Monte Carlo simulation engine.
 18. Simulation Example with Distribution Fitting
 - Illustrates how to fit a distribution to a dataset.
 19. Creating a Problem Object
 - Example illustrates two different methods of creating the *CProblem* object.
 20. Adding Variable and Function Blocks
 - This example illustrates the use of pointers to create a *CProblem* object.
 21. Creating a vector of Problem Objects
 - Creates a vector of problem objects, each with 2 vars and 4 fens.

Note: This example only will work if multiple licences exist.
 22. Traveling Salesman Problem 1 [All-Different Constraint]
 - Example models the Traveling Salesman Problem using an All-Different Constraint.
 23. Traveling Salesman Problem 2 [Multiple All-Different Constraints]

- This example is identical to Example 22, but this example models four decision variables using four blocks of variables, each of size 1.
24. Simulation Example 2
 - This example illustrates the use of the Truncate function to define the uncertain distributions; it also illustrates the use of a simple binary search as an alternative to the PsiTarget function.
 25. Simulation Example 3
 - This example is similar to example24; the difference lies in how we model the variables and the uncertain functions. In this example, we create a single block of 2 variables, and just 1 uncertain function block with 100 functions.
 26. Simulation Rank Correlations Example
 - Example illustrates how to specify a correlation matrix.
 27. Optimization within Simulation Example
 - In this example we find the optimal order quantity automatically, by embedding an optimization problem inside each trial of a simulation problem.
 28. Simulation within Optimization Example
 - In this example, which is similar to example 27, we find the optimal order quantity by embedding a simulation problem inside of an optimization problem.
 29. Simulation Optimization using SIP/SLURP
 - Example generates a sample of demand and price (the uncertain variables) in the main program, stores these sampled values in appropriate arrays, then use these values to compute the constraints and the objective of the optimization problem in the Evaluator.
 30. Sudoku with All-Different Constraints
 - Constructs 3 arrays of variable objects to create 3 variable objects each representing 9 blocks of variables, where each block will contain 9 variables.
 31. Simulation Statistical Functions Example
 - Illustrates the different statistical functions that are supported for uncertain variables.
 32. PSD Correlation Matrix Example
 - This example demonstrates using a correlation matrix to correlate input uncertain variables.

There are five additional examples installed into the Frontline Systems\Solver SDK Platform\Examples\VB6 directory. You can run each of them by clicking the appropriate solution file inside of each directory described below.

1. CollegeFundModel
 - This example illustrates how to load and solve a model from an Excel workbook that contains a Solver simulation model. The simulation predicts the amount of funds needed to cover college tuition.
2. CutStock
 - A cutting stock problem involves allocating space on large sheets of paper or timbers to be cut into smaller pieces to meet demand while minimizing waste.
3. Portfolio
 - An example that uses the Markowitz method to obtain the best combination of stocks to minimize the variance of the portfolio given a return.
4. ProductMix

- This example demonstrates how to load and solve a model from an Excel workbook that contains a Solver optimization model.
5. TSP
- This example uses the Evolutionary Engine to solve a classic Traveling Salesman Problem.

From Excel to SDK

Introduction

In many instances, users have a working optimization or simulation model in Excel that they want to solve with the SDK. Using just three lines of code, you can easily load an Excel model with the Solver SDK via the *prob.load()* function. This method allows the User to maintain the data and the model in Excel while solving the model with the SDK. (For more information along with an example on loading and solving Excel models with the SDK, please see the section, "Loading and Solving Excel Models" in the chapter, "Designing Your Application".) However, there are times when a user wants to leave Excel completely and create a stand-alone application that creates and solves a model using the SDK in a programming language of their choice. This chapter breaks down the task into manageable steps and answers the question, "Where do I start?"

If during the course of your transition from Excel to the SDK, you find that you have any questions or concerns about your particular model, we invite you to contact our Support desk at support@solver.com.

Optimization Problem Type

This chapter examines two different optimization problem types, a linear example and a nonlinear example. If you are unsure what type of model you are solving and you own a license for Analytic Solver for Excel, you can perform a problem diagnosis. To do so, click **Optimize|Analyze without Solving** on the Analytic Solver ribbon. The Solver will parse your model to determine the structure and will stop with a model type diagnosis of LP (Linear Problem), QP (Quadratic Problem), QCP (Quadratically Constrained Problem), NLP (Nonlinear Problem), or NSP (Nonsmooth Problem). In the Solver SDK Platform, LPs and QPs can be solved with the LP/Quadratic Engine; LPs, QPs and QCPs can be solved with the SOCP engine; NLPs and some NSPs with the GRG Nonlinear engine, and NSPs with the Evolutionary Engine. For more information on how to diagnose your model using the Analytic Solver, please see the Analytic Solver User Guide or contract support at Support@solver.com.

Converting a Linear Example

Extracting the Problem from the Excel Worksheet

For the linear example, we will be using the Product Mix example problem to illustrate how to extract the decision variables, constraints and the objective function so they can be encoded for the SDK using the programming language C++.

This example introduces a company that manufactures TVs, stereos and speakers, using a common parts inventory of power supplies, speaker cones, electronics, etc. Parts are in limited supply. The objective of this example is to determine the most profitable mix of products to build.

Excel Solver Model

In the screenshot below, the variables, the amount of TVs, Stereos and Speakers that are to be made, are located in cells C14:E14 and are surrounded by a blue border. The objective is in cell G24 and is being maximized. The constraints in cells G18:G22 must be less than or equal to ($\leq$) cells H18:H22 and the variables must all be 0 or larger.

	A	B	C	D	E	F	G	H
12			Production Decisions					
13			LCD TV	Stereo	Speakers			
14		Number to Build:	0	0	0			
15								
16			Part Requirements by Product				Part Inventory Levels	
17		Part Name	LCD TV	Stereo	Speakers		Used	Available
18		Chassis	1	1	0		0	450
19		LCD Screen	1	0	0		0	250
20		Speaker	2	2	1		0	800
21		Power Supply	1	1	0		0	450
22		Electronics	2	1	1		0	600
23								
24		Profit per unit	\$75	\$50	\$35		\$0	
25								

Extracting the Variables

As mentioned above, the three variables are included in cells C14:E14. We can denote the variables in cells C14:E14 as var1, var2, and var3, respectively.

Extracting the Constraints

The constraint coefficients are located in cells C18:E22 while the constraint right hand sides are located in cells H18:H22. Cells G18:G22 multiply the variables in cells C14:F14 by the constraint coefficients (separately by constraint). Let's take the first constraint as an example. Cell G18 = $\text{=SUMPRODUCT}(C18:E18, \$C\$14:\$E\$14)$ which is multiplying the coefficients for the first constraint (in cells C18:E18) by the variable values (in cells C14:E14). This constraint ensures that no more than 450 (H18 = 450) chassis are used in the total production of the three products. A TV set requires 1 chassis (C18=1), a stereo requires 1 chassis (D18=1) while speakers do not use a chassis at all (E18 = 0).

The rest of the constraints in the range G18:G22 are as follows:

G19 = $\text{SUMPRODUCT}(C19:E19, \$C\$14:\$E\$14)$

G20 = $\text{SUMPRODUCT}(C20:E20, \$C\$14:\$E\$14)$

G21 = $\text{SUMPRODUCT}(C21:E21, \$C\$14:\$E\$14)$

G22 = $\text{SUMPRODUCT}(C22:E22, \$C\$14:\$E\$14)$

It follows that the first constraint can be rewritten as: $1 * \text{var1} + 1 * \text{var2} + 0 * \text{var3}$. As mentioned above, the right hand sides of the constraints are located in cells H18:H22 and are 450, 250, 800, 450, 600, respectively. This means that the constraint located in cells $G18 \leq H18$, $G19 \leq H19$, and so on.

Chassis Constraint: $1 * \text{var1} + 1 * \text{var2} + 0 * \text{var3} \leq 450$

Picture Tube Constraint: $1 * \text{var1} + 0 * \text{var2} + 0 * \text{var3} \leq 250$

Speaker Cone Constraint: $2 * \text{var1} + 2 * \text{var2} + 1 * \text{var3} \leq 800$

Power Supply Constraint: $1 * \text{var1} + 1 * \text{var2} + 0 * \text{var3} \leq 450$

Electronics Constraint: $2 * \text{var1} + 1 * \text{var2} + 1 * \text{var3} \leq 600$

Finally, we must ensure that the lower bounds on our variables must be 0 since it is impossible to create negative products.

Extracting the Objective Function

The objective function resides in cell G24 = SUMPRODUCT(C24:E24, C14:E14) which is multiplying the objective function coefficients in cells C24:E24 (75, 50, and 35) by the variables in cells C14:E14. In addition, the Solver Task Pane tells us that the objective is being Maximize. As a result, this function can be written as:

Maximize: $75 * \text{var1} + 50 * \text{var2} + 35 * \text{var3}$

Complete Model

At this point, we have all the information we need to create this same linear problem (below) in the SDK.

Maximize: $75 * \text{var1} + 50 * \text{var2} + 35 * \text{var3}$

Chassis Constraint: $1 * \text{var1} + 1 * \text{var2} + 0 * \text{var3} \leq 450$

Picture Tube Constraint: $1 * \text{var1} + 0 * \text{var2} + 0 * \text{var3} \leq 250$

Speaker Cone Constraint: $2 * \text{var1} + 2 * \text{var2} + 1 * \text{var3} \leq 800$

Power Supply Constraint: $1 * \text{var1} + 1 * \text{var2} + 0 * \text{var3} \leq 450$

Electronics Constraint: $2 * \text{var1} + 1 * \text{var2} + 1 * \text{var3} \leq 600$

$\text{var1}, \text{var2}, \text{var3} \geq 0$

Programming the Problem for the SDK

Creating the Problem

To begin, we will first initialize two variables, *nvars* and *ncons*, to pass the number of variables and constraints in the model. Our model contains 3 variables and 5 constraints.

```
int nvars = 3, ncons = 5;
```

Next, we create an array to hold the upper bounds of the constraint right hand sides.

```
double ub_constraints[] = { 450, 250, 800, 450, 600};
```

This next line of code (CProblem prob) creates an instance of the CProblem class. For more information on creating a Problem, please see the chapter for the programming language of choice, **Calling the SDK from C++, C#, the Procedural API or from COM** using Visual Basic 6.

```
CProblem prob;
```

The next line of code creates one variable block of size *nvars* (three).

```
prob.VarDecision.Size = nvars;
```

while the next line of code creates one function block of size *ncons* (five).

```
prob.FncConstreaint.Size = ncons;
```

To tell the SDK to maximize the model, we use the line of code below.

```
prob.Solver.SolverType = Solver_Type_Maximize;
```

We pass the *ub_constraints* array which holds the upper bounds of the constraint right hand sides to the SDK.

```
prob.FcnConstraint.UpperBound = ub_constraints;
```

To pass a lower bound of 0 on all three variables, call:

```
prob.VarDecision.NonNegative();
```

Passing the Objective and Constraints

The next two lines of code create two different *DoubleMatrix* objects to hold the objective and constraint coefficients: *objcoeff*- for the objective coefficients, and *concoeff*- for the constraint coefficients. Notice that we are using *NZBgn()* and *NZEnd()* when passing the constraint matrix (*concoeff*) as explained in the subsection *Passing Sparse Matrices* in the chapter *Designing Your Application*. As a result, the Solver SDK will not update the *concoeff* *DoubleMatrix* object until *NZEnd()* is executed. Also note that there is no need to pass zeros for the coefficients located in Excel cells F6, E7, F7, and F9.

```
CDoubleMatrix objcoeff(1, nvars);
    objcoeff.Value[0][0] = 75;
    objcoeff.Value[0][1] = 50;
    objcoeff.Value[0][2] = 35;

CDoubleMatrix concoeff (ncons, nvars);
    concoeff.NZBgn();
    concoeff.Value[0][0] = 1;
    concoeff.Value[0][1] = 1;
    concoeff.Value[1][0] = 1;
    concoeff.Value[2][0] = 2;
    concoeff.Value[2][1] = 2;
    concoeff.Value[2][2] = 1;
    concoeff.Value[3][0] = 1;
    concoeff.Value[3][1] = 1;
    concoeff.Value[4][0] = 2;
    concoeff.Value[4][1] = 1;
    concoeff.Value[4][1] = 1;
    concoeff.NZEnd();
```

The example code below calls *prob.Model.AllLinear[Function_Type_Objective]* to pass the objective coefficient matrix, *objcoeff*, and *prob.Model.AllLinear[Function_Type_Constraint]* to pass the constraint coefficient matrix, *concoeff*. Calling *prob.Model.AllLinear* sets the problem type to "linear". Consequently, the SDK will automatically choose the best linear engine available to solve the problem. Within the SDK engines, the LP/Quadratic engine will be selected.

```
prob.Model.AllLinear[Function_Type_Objective] = objcoeff;
prob.Model.AllLinear(Function_Type_Constraint) = concoeff;
```

Solve the optimization problem by calling:

```
prob.Solver.Optimize();
```

From here you can obtain and display the solution as illustrated in the previous chapters, please select the appropriate link: [C++](#), [C#](#), [Procedural API](#), or [COM](#).

Complete SDK Program

Please see below for a complete snapshot of the code required for our linear example.

```
int nvars = 3, ncons = 5;
double ub_constraints[] = { 450, 250, 800, 450, 600};
CProblem prob;
```

```

prob.VarDecision.Size = nvars;
prob.FncConstreaint.Size = ncons;
prob.Solver.SolverType = Solver_Type_Maximize;
prob.FcnConstraint.UpperBound = ub_constraints;
prob.VarDecision.NonNegative();
CDoubleMatrix objcoeff(1, nvars);
    objcoeff.Value[0][0] = 75;
    objcoeff.Value[0][1] = 50;
    objcoeff.Value[0][2] = 35;
CDoubleMatrix concoeff (ncons, nvars);
    concoeff.NZBgn();
    concoeff.Value[0][0] = 1;
    concoeff.Value[0][1] = 1;
    concoeff.Value[1][0] = 1;
    concoeff.Value[2][0] = 2;
    concoeff.Value[2][1] = 2;
    concoeff.Value[2][2] = 1;
    concoeff.Value[3][0] = 1;
    concoeff.Value[3][1] = 1;
    concoeff.Value[4][0] = 2;
    concoeff.Value[4][1] = 1;
    concoeff.Value[4][1] = 1;
    concoeff.NZEnd();
prob.Model.AllLinear[Function_Type_Objective] = objcoeff;
prob.Model.AllLinear[Function_Type_Constraint] = concoeff;
prob.Solver.Optimize();

```

For example code on solving linear problems in additional languages, select the appropriate link: [C++](#), [C#](#), [Procedural API](#), or [COM](#).

Converting a Nonlinear Example

Extracting the Problem from the Excel Worksheet

Our next example is a nonlinear problem which finds the best ordering policy for a warehouse to minimize costs while meeting demands. The warehouse sells four different products. Each product has its own demand, holding cost and requires a certain amount of space.

Excel Solver Model

The Model Taskpane shows the Solver Model in the screenshot below. The objective is located in cell G19 and is being minimized. The variables are contained in cells C17:F17 which are surrounded by a blue border. These variables must be integer values only. The two constraints are G20 <= H20 and G21 <= H21.

Extracting the Variables

As mentioned above, the variables are included in cells C17:F17. We can denote the variables as var1, var2, var3, and var4, respectively.

Extracting the Constraints

The first constraint ($G21 \leq H21$) ensures that the total cost of the products (cell G21) ordered must be within the budget of \$30,000 (cell H21).

$$G21 = \text{SUM}(C21:F21) \text{ where } C21 = C16 * C17, D21 = D16 * D17, E21 = E16 * E17, F21 = F16 * F17$$

If we denote the variables in cells C17:F17 as var1, var2, var3 and var4, respectively. This constraint can be rewritten as:

$$\text{Constraint1: } 200 * \text{var1} + 300 * \text{var2} + 275 * \text{var3} + 400 * \text{var4} \leq 30000$$

The second constraint ($G20 \leq H20$) requires that the total space used by each product (cell G20) is less than the amount of warehouse space available (cell H20).

$$G20 = \text{SUM}(C20:F20)$$

where..

$$C20 = C17 * C13/2$$

$$D20 = D17 * D13/2$$

$$E20 = E17 * E13/2$$

$$F20 = F17 * F13/2$$

This constraint can be rewritten as:

$$\text{Constraint2: } 440/2 * \text{var1} + 850/2 * \text{var2} + 1260/2 * \text{var3} + 950/2 * \text{var4} \leq 50000$$

Simplifying the constraint further yields:

$$\text{Constraint2: } 220 * \text{var1} + 425 * \text{var2} + 630 * \text{var3} + 475 * \text{var4} \leq 50000$$

Extracting the Objective Function

The objective function in cell G19 is minimizing the ordering cost as well as the holding cost.

$$G19 = \text{SUM}(C19:F19) \text{ where...}$$

$$C19 = C12 * C17/2 + C15 * C14/C17$$

$$D19 = D12 * D17/2 + D15 * D14/D17$$

$$E19 = D12 * E17/2 + E15 * E14/D17$$

$$F19 = F12 * F17/2 + F15 * F14/F17$$

The objective can be rewritten as:

$$\text{Minimize: } (25 * \text{var1}/2 + 50 * 200/\text{var1}) + (20 * \text{var2} + 50 * 325/\text{var2})$$

$$+ (30 * \text{var3} + 50 * 400/\text{var3}) + (15 * \text{var4}/2 + 50 * 150/\text{var4})$$

Complete Model

At this point, we have all the information we need to create this same nonlinear problem (below) in the SDK.

$$\text{Minimize: } (25 * \text{var1}/2 + 50 * 200/\text{var1}) + (20 * \text{var2} + 50 * 325/\text{var2})$$

$$+ (30 * \text{var3} + 50 * 400/\text{var3}) + (15 * \text{var4}/2 + 50 * 150/\text{var4})$$

$$\text{Constraint1: } 200 * \text{var1} + 300 * \text{var2} + 275 * \text{var3} + 400 * \text{var4} \leq 30000$$

$$\text{Constraint2: } 220 * \text{var1} + 425 * \text{var2} + 630 * \text{var3} + 475 * \text{var4} \leq 50000$$

Programming the Same Problem for the SDK

Creating the Problem

To begin, we initialize the number of variables (*nvars*) and constraints (*ncons*). In our example, we have 4 variables and 2 constraints.

```
int nvars = 4, ncons = 2;
```

Next, we create one array of size *ncons* (2 constraints) to hold the upper bounds of the constraint right hand sides.

```
double ub_constraints[2] = { 30000, 50000 }
```

This next line of code (*CProblem prob (Solver_Type_Minimize, nvars, ncons)*) creates an instance of the *CProblem* class. This constructor not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type_Maximize*, *Solver_Type_Minimize*, or *Solver_Type_Simulate*) the number of variables (*nvars*) and the number of constraints (*ncons*). For more information on creating a Problem, please see the chapter for the programming language of choice, **Calling the SDK from C++, C#, the Procedural API** or from **COM** using Visual Basic 6.

```
CProblem prob(Solver_Type_Minimize, nvars, ncons);
```

Here we pass the constraint upper (*ub_constraints*) bounds to the SDK,

```
prob.FcnConstraint.UpperBound = ub_constraints;
```

Create the Evaluator to Compute the Objective and Constraints

In a linear model or quadratic model, you typically supply the Solver SDK with the objective and constraint coefficient matrices. The SDK can determine the values of the objective and constraints by computing the sums of products of the variables with the supplied coefficients. However, in a nonlinear model, the coefficients represent first partial derivatives of the objective and constraints with respect to the variables, and these derivatives change as the values of the variables change. Therefore, we must supply an evaluator to compute the function (objective and constraint) values. This evaluator will be called at every iteration, to compute the values of the objective and constraints, as well as to compute numerical derivatives, if no Jacobian evaluator is supplied. (It is possible to pass these derivatives directly to the SDK using an evaluator of type *Eval_Type_Jacobian*. For more information, see the subheading, *Using a Gradient Evaluator to Provide First Order Partial Derivatives* in the chapter *Solver SDK Evaluators*.)

We create and pass the evaluator of type *Eval_Type_Function* (as described above) to the SDK. The name of this evaluator is *evalNLP*.

```
CNLP evaluator evalNLP;
```

```
prob.Evaluators[Eval_Type_Function].Evaluator = &evalNLP;
```

Write the Evaluator to Compute the Objective and Constraints

Inside of the evaluator *evalNLP*, we first create a convenient pointer, *p*, to the Problem,

```
CProblem& p = evaluator.Problem;
```

and another pointer, *pVar*, to the variables.

```
const double* pVar=p.VarDecision.Value;
```

Next, we obtain a pointer to the constraint functions and calculate the constraints.

```
double* pFcn=p.FcnConstraint.Value;
```

Here, we pass the left hand side of the first constraint:

```
200 * var1 + 300 * var2 + 275 * var3 + 400 * var4.
```

```
pFcn[0] = 200 * pVar[0] + 300 * pVar[1] + 275 * pVar[2] + 400 *
        pVar[3];
```

and now the second constraint: $220 * \text{var1} + 425 * \text{var2} + 630 * \text{var3} + 475 * \text{var4}$.

```
pFcn[1] = 220 * pVar[0] + 425 * pVar[1] + 630 * pVar[2] + 475 *
        var[3];
```

Then we compute the objective function:

```
(25 * var1/2 + 50 * 200/var1) + (20 * var2 + 50 * 325/var2)
+ (30 * var3 + 50 * 400/var3) + (15 * var4/2 + 50 * 150/var4)
p.FcnObjective.Value[p.ObjectiveIndex] = ( 25 * pVar[0] / 2 + 50 *
200 / pVar[0] ) + ( 20 * pVar[1] / 2 + 50 * 325 / pVar[1] ) + ( 30
* pVar[2] / 2 + 50 * 400 / pVar[2] ) + ( 15 * pVar[3] / 2 + 50 * 150
/ pVar[3] ) ;
```

Finally, we tell the SDK to continue solving. There are two additional *Engine_Action* functions, *Engine_Action_Evaluate* and *Engine_Action_Stop*. *Engine_Action_Evaluate* forces the SDK to call the *Eval_Type_Function* evaluator a second time which can sometimes be useful. *Engine_Action_Stop* will tell the SDK to abort the optimization (or simulation).

```
return Engine_Action_Continue;
```

Our evaluator is complete. We will now finish the code inside of the *nonlinear()* function.

Solve the Nonlinear Problem

Back inside the *nonlinear()* function, we provide the problem type as nonlinear which directs the SDK to select the GRG engine to solve the model. Alternatively, we also could have set the GRG engine manually (in either the SDK Platform or the SDK Pro) by calling: `problem.Engine = problem.Engines[CEngine::GRGName]`.

```
prob.ProblemType = Problem_Type_OptNLP;
```

To solve the nonlinear optimization problem, we call:

```
prob.Solver.Optimize();
```

From here you can obtain and display the solution as illustrated in the previous chapters, please select the appropriate link: [C++](#), [C#](#), [Procedural API](#), or [COM](#).

Complete Program

Please see below for a complete snapshot of the code required for our nonlinear example.

Evaluator

```
CProblem& p = evaluator.Problem;
const double* pVar=p.VarDecision.Value;
double* pFcn=p.FcnConstraint.Value;
200 * var1 + 300 * var2 + 275 * var3 + 400 * var4
pFcn[0] = 200 * pVar[0] + 300 * pVar[1] + 275 * pVar[2] + 400 *
        pVar[3];
220 * var1 + 425 * var2 + 630 * var3 + 475 * var4.
pFcn[1] = 220 * pVar[0] + 425 * pVar[1] + 630 * pVar[2] + 475 *
        var[3];
(25 * var1/2 + 50 * 200/var1) + (20 * var2 + 50 * 325/var2)
+ (30 * var3 + 50 * 400/var3) + (15 * var4/2 + 50 * 150/var4)
```

```
p.FcnObjective.Value[p.ObjectiveIndex] = ( 25 * pVar[0] / 2 + 50 *
200 / pVar[0] ) + ( 20 * pVar[1] / 2 + 50 * 325 / pVar[1] ) + ( 30
* pVar[2] / 2 + 50 * 400 / pVar[2] ) + ( 15 * pVar[3] / 2 + 50 * 150
/ pVar[3] ) ;
```

```
return Engine_Action_Continue;
```

Main Body of Code

```
int nvars = 4, ncons = 2;
double ub_constraints[2] = { 30000, 50000}
CProblem prob(Solver_Type_Minimize, nvars, ncons);
prob.FcnConstraint.UpperBound = ub_constraints;
CNLPEvaluator evalNLP;
prob.Evaluators[Eval_Type_Function].Evaluator = &evalNLP;
prob.ProblemType = Problem_Type_OptNLP;
prob.Solver.Optimize();
```

For example code on solving nonlinear problems in additional languages, select the appropriate link: [C++](#), [C#](#), [Procedural API](#), or [COM](#).

Converting a Simulation Example

Extracting the Problem from the Excel Worksheet

Our final example is a simulation example. In this example, a software development project consists of 7 main components, which have been assigned to 7 different teams of software engineers. The time required to complete each of the 7 components is uncertain, but the average time required by each of the 7 teams is 2 weeks. However, for each of the 7 teams, there is a 10% chance that the time required is going to be more than 3 weeks and there is a 10% chance that the time required is going to be less than 1 week. This example estimates the total time required to complete the entire project.

Excel Solver Model

In the screenshot below, you can see that the uncertain variables are located in cells C10:C16. There is one uncertain function located in cell B20 and two statistics functions located in cells C20 and D20.

Time Required (in weeks)	
Lower Estimate	1
Average	2
Higher Estimate	3
Lower Estimate Probability	10%
Higher Estimate Probability	90%

	Time Required
Team1	1.94
Team2	0.96
Team3	2.85
Team4	2.78
Team5	1.67
Team6	3.40
Team7	0.48

Project Duration	Expected Project Duration	Chances of duration less than 14 weeks
14.10	14.00003	0.51

Extracting the Uncertain Variables

The uncertain variables are contained in cells C10:C16 which are bounded in green. This example models the time required by each of the seven teams using the PsiTriangGen distribution. This distribution requires five parameters: bottom, likely, top, bottom percentage and top percentage. (For a complete description of this distribution see PsiTriangGen in the Psi Function Reference chapter.) Each of these parameters are included in the spreadsheet below in cells C3:C7, Bottom = 1 (cell C3), Likely = 2 (C4), Top = 3 (C5), Bottom Percentage = 10% (C6) and Top Percentage = 90% (C7).

C10 = PsiTriangGen(\$C\$3,\$C\$4, \$C\$5, \$C\$6, \$C\$7)

C11 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

C12 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

C13 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

C14 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

C15 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

C16 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

Extracting the Uncertain Function

The uncertain function appears in cell B20 = SUM(C10:C16).

Extracting the Uncertain Function Statistics

Two statistic functions are held in cells C20 and D20. The statistic in cell C20 is the PsiMean() statistic and gives the expected duration of the project. The statistic in cell D20 is the PsiTarget() function which gives the likelihood of the project duration lasting less than 14 weeks.

C20 = PsiMean(B20)

D20 =PsiTarget(B20,14)

Complete Model

At this point, we have all that we need to move this model to the SDK.

Uncertain Variable1 = PsiTriangGen(\$C\$3,\$C\$4, \$C\$5, \$C\$6, \$C\$7)

Uncertain Variable2 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

Uncertain Variable3 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

Uncertain Variable4 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

Uncertain Variable5 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

Uncertain Variable6 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

Uncertain Variable7 = PsiTriangGen(\$C\$3, \$C\$4, \$C\$5, \$C\$6, \$C\$7)

Uncertain Function = Uncertain Variable1 + Uncertain Variable2 + Uncertain Variable3 + Uncertain Variable4 + Uncertain Variable5 + Uncertain Variable6 + Uncertain Variable7

Statistics of the Functions: Mean and Target(14%)

Programming the Same Problem for the SDK

Creating the Problem

To begin, we initialize the number of uncertain variables (*nvars*) and functions (*ncons*).

```
int nvars = 7, nfcns = 1;
```

This next line of code (*CProblem prob (Solver_Type_Simulate, nvars, ncons)*) creates an instance of the *CProblem* class. This constructor not only creates the Problem, but also passes the *Solver_Type*, (*Solver_Type_Minimize*, *Solver_Type_Maximize*, or *Solver_Type_Simulate*) the number of variables (*nvars*) and the number of constraints (*ncons*). For more information on creating a Problem, please see the chapter for the programming language of choice, **Calling the SDK from C++, C#, the Procedural API** or from **COM** using Visual Basic 6.

```
CProblem prob(Solver_Type_Simulate, nvars, ncons);
```

Next, we pass the distributions of the seven uncertain variables.

```
prob.Model.VarDistribution[0] =
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");

prob.Model.VarDistribution[1] =
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");

prob.Model.VarDistribution[2] =
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");

prob.Model.VarDistribution[3] =
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");

prob.Model.VarDistribution[4] =
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");

prob.Model.VarDistribution[5] =
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");

prob.Model.VarDistribution[6] =
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");
```

Create the Evaluator to Compute the Uncertain Functions

When running a simulation, the evaluator of type *Eval_Type_Sample* is called so the user can calculate the uncertain functions, much like the evaluator of type *Eval_Type_Function* is called to compute the objective and constraints in a nonlinear optimization problem.

We create the evaluator of type *Eval_Type_Sample* and pass the evaluator to the SDK. The name of the evaluator is *evalSim()*.

```
CSimEvaluator evalSim;

problem.Evaluators[Eval_Type_Sample].Evaluator = &evalSim;
```

Write the Evaluator to Compute the Uncertain Functions

Inside of the evaluator *evalSim*, we first create a convenient pointer, *p*, to the Problem,

```
CProblem& p = evaluator.Problem;
```

and a second pointer, *pVar*, to the uncertain variable values.

```
const double* pVar = p.VarUncertain.Value;
```

Next, we create a pointer, *pFcn*, to the uncertain functions.

```
double* pFcn=p.FcnUncertain.Value;
```

We compute the uncertain function values given the current variable values.

```
pFcn[0] = pVar[0] + pVar[1] + pVar[2] + pVar[3] + pVar[4] +
          pVar[5] + pVar[6];
```

Finally, we tell the SDK to continue simulating.

```
return Engine_Action_Continue;
```

Our evaluator is complete. We will now finish the code inside of the main problem function.

Solve the simulation model

Back inside the main program, we call the following line of code to run the simulation.

```
prob.Solver.Simulate();
```

Obtain and display various simulation statistics.

We obtain the simulation status.

```
_Simulate_Status status = prob.Solver.SimulateStatus;  
cout << "Status = " << status << endl;
```

The next line of code creates a convenient pointer, *FcnStat* for the uncertain function statistics.

```
CStatistics FcnStat = prob.FcnUncertain.Statistics;
```

While the next two lines of code display the two statistics for the uncertain function.

```
cout << " Expected Project Duration      = " << VarStat.Mean[0] <<  
endl;  
  
cout << " Chances of Duration <= 14 Weeks    = " <<  
VarStat.Target(14)[0] << endl;
```

This completes our simulation example transformation.

Complete Program

Please see below for a complete snapshot of the code required for our nonlinear example.

Evaluator

```
CProblem& p = evaluator.Problem;  
const double* pVar = p.VarUncertain.Value;  
double* pFcn=p.FcnUncertain.Value;  
pFcn[0] = pVar[0] + pVar[1] + pVar[2] + pVar[3] + pVar[4] +  
          pVar[5] + pVar[6];  
return Engine_Action_Continue;
```

Main Body of Code

```
int nvars = 7, nfcns = 1;  
CProblem prob(Solver_Type_Simulate, nvars, ncons);  
prob.Model.VarDistribution[0] =  
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");  
prob.Model.VarDistribution[1] =  
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");  
prob.Model.VarDistribution[2] =  
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");  
prob.Model.VarDistribution[3] =  
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");  
prob.Model.VarDistribution[4] =  
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");  
prob.Model.VarDistribution[5] =  
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");  
prob.Model.VarDistribution[6] =  
CDistribution(L"PsiTriangGen(1,2,3,.1,.9)");
```

```

CSimEvaluator evalSim;
problem.Evaluators[Eval_Type_Sample].Evaluator = &evalSim;
prob.Solver.Simulate();
_Simulate_Status status = prob.Solver.SimulateStatus;
cout << "Status = " << status << endl;
CStatistics FcnStat = prob.FcnUncertain.Statistics;
cout << " Expected Project Duration      = " << VarStat.Mean[0] <<
endl;
cout << " Chances of Duration <= 14 Weeks    = " <<
VarStat.Target(14)[0] << endl;

```

For example code on solving simulation problems in additional languages, select the appropriate link:
[C++](#), [C#](#), [Procedural API](#), [COM](#).

PSI Function Reference

Using PSI Functions

This chapter provides summary information on the Solver SDK Risk Solver Engine's PSI functions that you can use to create Monte Carlo simulation models. The Risk Solver is one of the five bundled engines in the Solver SDK Platform and one of the four bundled engines in the Solver SDK Pro.

You use **Distribution** functions to define the properties of uncertain variables, either via an analytic probability distribution and its parameters, or via a pre-generated Stochastic Information Packet (SIP).

You use **Correlation** matrices to specify relationships between uncertain variables and functions.

You can create a histogram of the sampled values for an uncertain variable or function using the **GetFrequency** method.

You use **Statistics** functions to obtain results of the simulation, such as statistics, frequency data, or raw trial data, for uncertain functions.

You use **Control Parameters** functions to modify the properties of analytic probability distributions, by shifting or truncating their domains, or by correlating one distribution with another.

You can find a distribution that best fits your data using the **Distribution Fitting** tools.

Using PSI Distribution Functions

The PSI Distribution functions are used to define the 'nature of the uncertainty' assumed by uncertain variables. They can be broadly classified into four groups:

- *Continuous analytic* distributions such as `PsiUniform()` and `PsiNormal()`
- *Discrete analytic* distributions such as `PsiBinomial()` and `PsiGeometric()`
- *Custom* distributions such as `PsiCumul()` and `PsiGeneral()`
- *Stochastic Information Packets (SIP)*

On each trial of a simulation, the Solver SDK draws a *random sample* value from each PSI Distribution function you use. A SIP operates differently: On each trial, the SDK draws the *next sequential* value listed in the SIP for that uncertain variable. Then Solver SDK uses these sample values to calculate your model and its uncertain functions.

The sample values drawn for PSI Distribution functions depend on the *type* of distribution function, the *parameters* of the distribution (for example, mean and variance for the `PsiNormal` distribution), and the *property* functions that you pass as additional arguments to the distribution function call, which can shift, truncate, or lock the distribution, or correlate its sample values with samples drawn for other uncertain variables.

To learn more about the analytic probability distributions below, you can consult standard reference texts on probability, statistics, and Monte Carlo simulation, such as *Simulation Modeling and Analysis*, 4th Ed. by Averill Law, *Statistical Distributions*, 3rd Ed. by Merran Evans, Nicholas Hastings and Brian Peacock, *Univariate Discrete Distributions*, 3rd Ed. by Norman Johnson, Adrienne Kemp and Samuel Kotz, or *Continuous Univariate Distributions, Vol. 1 & 2*, 2nd Ed. by Norman Johnson, Samuel Kotz and N. Balakrishnan.

Continuous Analytic Distributions

This section describes each Psi Distribution Function present in the Solver SDK. Please see the lines of example code below which illustrate how to pass a distribution function to the SDK. The lines of code are each passing the PsiNormal distribution with parameters mean (0) and standard deviation (2).

```
C++: prob.Model.VarDistribution[0] = Distribution(L"PsiNormal(0,2)");
C#:  prob.Model.VarDistribution[0] = Distribution("PsiNormal(0,2)");
Procedural: SolverModDistInit(prob, Dist_Var, 0, L"PsiNormal(0,2)");
COM:  Dim distNormal As New Distribution
      distNormal.Init "PsiNormal(1,1)"
      prob.Model.VarDistribution(1) = distNormal
```

PsiBeta

PsiBeta ($\alpha_1, \alpha_2, \dots$)

PsiBeta (α_1, α_2) is a flexible distribution for modeling probabilities based on Bayesian statistics. The Beta distribution can be used as an approximation in the absence of specific distribution information. Typical uses include modeling time to complete a task in project networks and Bayesian Statistics.

The Beta distribution can take on a variety of shapes depending on the values of the two parameters α_1 and α_2 . The Beta distribution with $\alpha_1 = \alpha_2 = 1$ is the Uniform (0,1) distribution. The Beta distribution with $\alpha_1 = 1, \alpha_2 = 2$ is the Left Triangular distribution. The beta distribution with $\alpha_1 = 2, \alpha_2 = 1$ is the Right Triangular distribution. A random variable X is defined by PsiBeta (α_1, α_2) if and only if $1 - X$ is defined by Beta (α_2, α_1).

Parameters

$$\alpha_1, \alpha_2 > 0$$

Range of Function Values

$$[0,1]$$

Probability Density Function

$$f(x) = \begin{cases} \frac{x^{\alpha_1-1} (1-x)^{\alpha_2-1}}{B(\alpha_1, \alpha_2)} & \text{if } x \in (0,1) \\ 0 & \text{otherwise} \end{cases}$$

$B(\alpha_1, \alpha_2)$ is the Beta function

$$B(\alpha_1, \alpha_2) = \int_0^1 t^{\alpha_1-1} (1-t)^{\alpha_2-1} dt$$

Cumulative Distribution Function

$$F(x) = \frac{B_x(\alpha_1, \alpha_2)}{B(\alpha_1, \alpha_2)}$$

$B_x(\alpha_1, \alpha_2)$ is the Incomplete Beta Function

$B(\alpha_1, \alpha_2)$ is the Beta Function

Mean

$$\frac{\alpha_1}{\alpha_1 + \alpha_2}$$

Variance

$$\frac{\alpha_1 \alpha_2}{(\alpha_1 + \alpha_2)^2 (\alpha_1 + \alpha_2 + 1)}$$

Skewness

$$\frac{2(\alpha_2 - \alpha_1)}{\alpha_1 + \alpha_2 + 2} \sqrt{\frac{\alpha_1 + \alpha_2 + 1}{\alpha_1 \alpha_2}}$$

Kurtosis

$$\frac{3(\alpha_1 + \alpha_2 + 1) \left[2(\alpha_1 + \alpha_2)^2 + \alpha_1 \alpha_2 (\alpha_1 + \alpha_2 - 6) \right]}{\alpha_1 \alpha_2 (\alpha_1 + \alpha_2 + 2)(\alpha_1 + \alpha_2 + 3)}$$

Median

Not applicable

Mode

$\frac{\alpha_1 - 1}{\alpha_1 + \alpha_2 - 2}$ if $\alpha_1 > 1, \alpha_2 > 1$
 0 and 1 if $\alpha_1 < 1, \alpha_2 < 1$
 0 if $(\alpha_1 < 1, \alpha_2 \geq 1)$ or if $(\alpha_1 = 1, \alpha_2 > 1)$
 1 if $(\alpha_1 \geq 1, \alpha_2 < 1)$ or if $(\alpha_1 > 1, \alpha_2 = 1)$
 does not uniquely exist if $\alpha_1 = \alpha_2 = 1$

PsiBetaGen

`PsiBetaGen ( $\alpha_1, \alpha_2, a, b, \dots$ )`

`PsiBetaGen ( $\alpha_1, \alpha_2, a, b$ )` is a *rescaled and relocated* Beta distribution, with lower and upper bounds given respectively by a and b . The shape parameters α_1, α_2 play the same role as in the `PsiBeta` function. If X is a Beta random variable with support in $[0, 1]$, then $a + (b - a)X$ is a Beta random variable with support in $[a, b]$.

Parameters

$\alpha_1, \alpha_2 > 0$
 $a < b$

Range of Function Values

$[a, b]$

Probability Density Function

$$f(x) = \frac{(x-a)^{\alpha_1-1} (b-x)^{\alpha_2-1}}{B(\alpha_1, \alpha_2)(b-a)^{\alpha_1+\alpha_2-1}}$$

$B(\alpha_1, \alpha_2)$ is the Beta Function

Cumulative Distribution Function

$$F(x) = \frac{B_z(\alpha_1, \alpha_2)}{B(\alpha_1, \alpha_2)}, z = \frac{x-a}{b-a}$$

$B_x(\alpha_1, \alpha_2)$ is the Incomplete Beta Function

$B(\alpha_1, \alpha_2)$ is the Beta Function

Mean

$$a + \frac{\alpha_1}{\alpha_1 + \alpha_2}(b-a)$$

Variance

$$\frac{\alpha_1 \alpha_2}{(\alpha_1 + \alpha_2)^2 (\alpha_1 + \alpha_2 + 1)} (b-a)^2$$

Skewness

$$\frac{2(\alpha_2 - \alpha_1)}{\alpha_1 + \alpha_2 + 2} \sqrt{\frac{\alpha_1 + \alpha_2 + 1}{\alpha_1 \alpha_2}}$$

Kurtosis

$$\frac{3(\alpha_1 + \alpha_2 + 1) \left[2(\alpha_1 + \alpha_2)^2 + \alpha_1 \alpha_2 (\alpha_1 + \alpha_2 - 6) \right]}{\alpha_1 \alpha_2 (\alpha_1 + \alpha_2 + 2)(\alpha_1 + \alpha_2 + 3)}$$

Median

Not applicable

Mode

$$a + \frac{\alpha_1 - 1}{\alpha_1 + \alpha_2 - 2} (b - a) \text{ if } \alpha_1 > 1, \alpha_2 > 1$$

a and b if $\alpha_1 < 1, \alpha_2 < 1$

a if $(\alpha_1 < 1, \alpha_2 \geq 1)$ or if $(\alpha_1 = 1, \alpha_2 > 1)$

b if $(\alpha_1 \geq 1, \alpha_2 < 1)$ or if $(\alpha_1 > 1, \alpha_2 = 1)$

does not uniquely exist if $\alpha_1 = \alpha_2 = 1$

PsiBetaSubj

PsiBetaSubj ($a, c, \mu, b, \dots$)

PsiBetaSubj is a flexible distribution like PsiBetaGen, but with parameters you choose for the minimum (a), most likely (c), mean (μ) and maximum (b) values. These parameters are used to compute the shape parameters α_1, α_2 used in the PsiBeta function.

Parameters

$$a < \mu < b$$

$$a < c < b$$

$$\mu > \frac{a+b}{2} \text{ if } c > \mu$$

$$\mu < \frac{a+b}{2} \text{ if } c < \mu$$

$$\mu = \frac{a+b}{2} \text{ if } c = \mu$$

The shape parameters α_1, α_2 can be determined using

$$\alpha_2 = \alpha_1 \frac{b - \mu}{\mu - a}$$

$$\alpha_1 = \frac{2(\mu - a) \left(\frac{a+b}{2} - c \right)}{(\mu - c)(b - a)}$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \frac{(x-a)^{\alpha_1-1} (b-x)^{\alpha_2-1}}{B(\alpha_1, \alpha_2) (b-a)^{\alpha_1+\alpha_2-1}}$$

$B(\alpha_1, \alpha_2)$ is the Beta Function

Cumulative Distribution Function

$$F(x) = \frac{B_z(\alpha_1, \alpha_2)}{B(\alpha_1, \alpha_2)}, z = \frac{x-a}{b-a}$$

$B_x(\alpha_1, \alpha_2)$ is the Incomplete Beta Function

$B(\alpha_1, \alpha_2)$ is the Beta Function

Mean

$$\mu$$

Variance

$$\frac{(\mu-a)(b-\mu)(\mu-c)}{a+b+\mu-3c}$$

Skewness

$$\frac{(a+b-2\mu)}{\left| \mu + \frac{a+b}{2} - 2c \right|} \sqrt{\frac{(\mu-c)(a+b+\mu-3c)}{(\mu-a)(b-\mu)}}$$

Kurtosis

$$\frac{3(\alpha_1 + \alpha_2 + 1) \left[2(\alpha_1 + \alpha_2)^2 + \alpha_1 \alpha_2 (\alpha_1 + \alpha_2 - 6) \right]}{\alpha_1 \alpha_2 (\alpha_1 + \alpha_2 + 2)(\alpha_1 + \alpha_2 + 3)}$$

Median

Not applicable

Mode

$$c$$

PsiBurr12

`PsiBurr12 (loc, scale, shape1, shape2)`

PsiBurr12 is a continuous probability distribution for a non-negative random variable. This distribution uses four parameters: a location parameter, *loc*, scale parameter, *scale*, and shape parameters *shape1* and *shape2*. This distribution is typically used to model household income.

Parameters

Loc (γ) > 0, scale(β) > 0, shape1(α_1) > 0, shape2 (α_2) > 0

Function Values

$[0, \infty)$

Mean

$\lambda + \beta q_1$ for $\alpha_1 \alpha_2 > 1$

Variance

$B^2[q_2 - q_1^2]$ for $\alpha_1 \alpha_2 > 2$

Skewness

$\frac{q_3 - 3q_1 q_2 + 2q_1^3}{(q_2 - q_1^2)^{\frac{3}{2}}}$ for $\alpha_1 \alpha_2 > 3$

Kurtosis

$\frac{q_4 - 4q_1 q_3 + 6q_1^2 q_2 - 3q_1^4}{(q_2 - q_1^2)^2}$ for $\alpha_1 \alpha_2 > 4$

Mode

$\gamma + \beta \left(\frac{\alpha_1 - 1}{\alpha_1 \alpha_2 + 1} \right)^{\frac{1}{\alpha_2}}$ $\alpha_2 > 1$

γ $\alpha_2 \leq 1$

PsiCauchy

PsiCauchy ($\lambda, \dots$)

PsiCauchy (λ) is a distribution with a central peak, with very heavy tails and no finite moments; it has no moments such as mean, variance, etc. defined, but its mode and median are both equal to zero. The ratio of two independent standard Normal random variables is a Cauchy distribution with parameter $\lambda = 1$.

Parameters

$\lambda > 0$

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{1}{\pi \lambda \left[1 + \left(\frac{x}{\lambda} \right)^2 \right]}$$

Cumulative Distribution Function

$$F(x) = \frac{1}{\pi} \arctan\left(\frac{x}{\lambda}\right) + \frac{1}{2}$$

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

0

Mode

0

PsiChiSquare

PsiChiSquare (df, ...)

PsiChiSquare (df) is a distribution with a finite lower bound of zero, and an infinite upper bound. It is usually used in statistical significance tests.

The Chi Square distribution is a special case of the Gamma distribution. A Chi Square random variable with parameter $df = 2$ is the same as an Exponential random variable with mean 0.5. As the parameter df approaches infinity, the Chi Square distribution tends to a Normal distribution.

Parameters

$df > 0$, integer

Range of Function Values

$[0, \infty)$

Probability Density Function

$$f(x) = \frac{1}{2^{df/2} \Gamma(df/2)} x^{(df/2)-1} e^{-x/2}$$

$\Gamma(a)$ is the Gamma Function,

$$\Gamma(a) = \int_0^{\infty} t^{a-1} e^{-t} dt$$

Cumulative Distribution Function

$$F(x) = \frac{\gamma(df/2, x/2)}{\Gamma(df/2)}$$

$\Gamma(a)$ is the Gamma Function,

$\gamma(a, b)$ is the Incomplete Gamma Function

Mean

$$df$$

Variance

$$2df$$

Skewness

$$\sqrt{8/df}$$

Kurtosis

$$\frac{12}{df} + 3$$

Median

$$df - \frac{2}{3}$$

Mode

$$\begin{cases} (df - 2) & \text{if } df \geq 2 \\ 0 & \text{if } df = 1 \end{cases}$$

PsiDagum

PsiDagum (loc, scale, shape1, shape2)

PsiDagum is a continuous probability distribution for a non-negative random variable. This distribution uses four parameters: a location parameter, *loc* (x), scale parameter, *scale* (b), and shape parameters *scale1* (p) and *scale2* (a). This distribution is mostly associated with the study of income distribution and actuarial statistics.

Parameters

loc (γ), scale (β) > 0, shape1 (α_1) > 0, shape2 (α_2) > 0

Range of Function Values

(0, ∞)

Probability Distribution Function

$$f(x) = \frac{\alpha_1 \alpha_2}{\beta} \frac{z^{\alpha_1 \alpha_2 - 1}}{(1 + z^{\alpha_1})^{\alpha_2 + 1}} \quad \text{where } z \equiv \frac{x - \gamma}{\beta}$$

Cumulative Distribution Function

$$F(x) = (1 + z^{-\alpha_1})^{-\alpha_2} \quad \text{where } z \equiv \frac{x - \gamma}{\beta}$$

Mean

$$\gamma + \beta q_1 \quad \text{for } \alpha_1 > 1$$

Variance

$$B^2 [q_2 - q_1^2] \quad \text{for } \alpha_1 > 2$$

Skewness

$$\frac{q_3 - 3q_1q_2 + 2q_1^3}{(q_2 - q_1^2)^{\frac{3}{2}}} \text{ for } \alpha_2 > 3$$

Kurtosis

$$\frac{q_4 - 4q_1q_3 + 6q_1^2q_2 - 3q_1^4}{(q_2 - q_1^2)^2} \text{ for } \alpha_1 > 4$$

Mode

$$\gamma + \beta \left(\frac{\alpha_1 \alpha_2 - 1}{\alpha_1 + 1} \right)^{\frac{1}{\alpha_1}} \quad \alpha_1 \alpha_2 > 1$$

$$\gamma \quad \alpha_1 \alpha_2 \leq 1$$

PsiDbfTriang

PsiDbfTriang (min, likely, max, p)

PsiDbfTriang is a continuous probability distribution with minimum, likely, and maximum values, along with the probability p of a value falling between min and likely. PsiDbfTriang allows additional probability information as compared with the standard triangular distribution, PsiTriangular.

PsiDbfTriang joins two PsiTriangular distributions, the first from min to likely and the second from likely to max. Probability p is used to guarantee that the two parts of the distribution are normalized and the total area under the density curve is equal to 1.

Parameters

$\min < \text{likely} < \max$, $0 < p < 1$

Range of Function Values

(min, max)

Probability Distribution Function

$$f(x) = \frac{2p(x - \min)}{(\text{likely} - \min)^2} \text{ where } \min \leq x \leq \text{likely}$$

$$f(x) = \frac{2(1-p)(\max - x)}{(\max - \text{likely})^2} \text{ where } \text{likely} \leq x \leq \max$$

Cumulative Distribution Function

$$F(x) = \frac{p(x - \min)^2}{(\text{likely} - \min)^2} \text{ where } \min \leq x \leq \text{likely}$$

$$f(x) = 1 - \frac{(1-p)(\max - x)^2}{(\max - \text{likely})^2} \text{ where } \text{likely} \leq x \leq \max$$

Mean

$$\frac{(p)(\min) + 2(\text{likely}) + (1-p)(\max)}{3}$$

Variance

N/A

Skewness

N/A

Kurtosis

N/A

Mode

likely

PsiErf

PsiErf (h, ...)

PsiErf is a distribution based on the "error function" ERF(x). Its shape is closely related to the Normal distribution.

Parameters

$$h > 0$$

Range of Function Values

$$(-\infty, \infty)$$

Probability Density Function

$$f(x) = \frac{h}{\sqrt{\pi}} e^{-(hx)^2}$$

Cumulative Distribution Function

$$F(x) = \Phi(\sqrt{2}hx)$$

$\Phi(v)$ is the Error Function

$$\Phi(v) = \frac{2}{\sqrt{\pi}} \int_0^v e^{-t^2} dt$$

Mean

$$0$$

Variance

$$\frac{1}{2h^2}$$

Skewness

$$0$$

Kurtosis

$$3$$

Median

$$0$$

Mode

$$0$$

PsiErlang

PsiErlang ($k, \beta, \dots$)

PsiErlang (k, β) is a distribution with a finite lower bound, closely related to the Gamma and Exponential distributions. It has applications in reliability and queuing models. When the parameter $k = 1$, the Erlang distribution is the same as an Exponential distribution.

Parameters

$k, \beta > 0$

k integer

Range of Function Values

$[0, \infty)$

Probability Density Function

$$f(x) = \frac{x^{k-1} e^{-(x/\beta)}}{\beta^k (k-1)!}$$

Cumulative Distribution Function

$$F(x) = \frac{\gamma\left(k, \frac{x}{\beta}\right)}{(k-1)!}$$

$\gamma(x, y)$ is the Incomplete Gamma Function

Mean

$k\beta$

Variance

$k\beta^2$

Skewness

$\frac{2}{\sqrt{k}}$

Kurtosis

$3 + \frac{6}{k}$

Median

Not defined

Mode

$\beta(k-1)$

PsiExponential

`PsiExponential (β, ...)`

PsiExponential (β) is a distribution with a finite lower bound and rapidly decreasing values. It can be used to represent time between random occurrences in queuing and reliability engineering applications.

The minimum of a set of independent exponential random variables is also an exponentially distributed random variable.

Parameters

$$\beta > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \frac{1}{\beta} e^{-x/\beta}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 1 - e^{-x/\beta} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Mean

$$\beta$$

Variance

$$\beta^2$$

Skewness

$$2$$

Kurtosis

$$9$$

Median

$$\beta \ln(2)$$

Mode

$$0$$

PsiFatigueLife

`PsiFatigueLife (loc, scale, shape)`

Also known as the Birnbaum-Saunders distribution, the PsiFatigueLife distribution is used to model failure times, i.e. model crack propagation and the failure of materials over time.

Parameters

loc, scale (β) > 0, shape (α) > 0

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{1}{2\sqrt{2\pi}\beta\alpha} (z^{-1/2} + z^{-3/2}) \exp\left[-\frac{1}{2}\left(\frac{z^{1/2} - z^{-1/2}}{\alpha}\right)^2\right]$$

where $z \equiv \frac{x-\gamma}{\beta}$ and Φ is the cumulative distribution function of the standard normal distribution.

Cumulative Distribution Function

$$F(x) = \Phi\left(\frac{z^{1/2} - z^{-1/2}}{\alpha}\right)$$

where $z \equiv \frac{x-\gamma}{\beta}$ and Φ is the cumulative distribution function of the standard normal distribution.

Mean

$$\gamma + \beta \left(1 + \frac{\alpha^2}{2}\right)$$

Variance

$$(\alpha\beta)^2 \left(1 + \frac{5\alpha^2}{4}\right)$$

Skewness

$$\frac{44\alpha^3 + 24\alpha}{(5\alpha^2 + 4)^{3/2}}$$

Kurtosis

$$3 + \frac{6\alpha^2(93\alpha^2 + 41)}{(5\alpha^2 + 4)^2}$$

Mode

No Analytic Formula

PsiFDist

PsiFDist (df1, df2)

The PsiFDist distribution is an F distribution with two degrees of freedom, df1 and df2. Typically, this distribution is associated with statistical hypothesis testing, specifically when determining if two population variances are equal.

Parameters

df1 > 0 and integer, df2 > 0 and integer

Range of Function Values

$(0, \infty)$

Probability Density Function

$$f(x) = \frac{\left(\frac{v_1}{v_2}\right)^{v_1/2} x^{\frac{v_1-2}{2}}}{B\left(\frac{v_1}{2}, \frac{v_2}{2}\right) \left[1 + \frac{v_1}{v_2} x\right]^{\frac{v_1+v_2}{2}}}$$

Cumulative Distribution Function

$$F(x) = I_{1+\frac{v_2}{v_1 x + v_2}}\left(\frac{v_1}{2}, \frac{v_2}{2}\right)$$

where β is the Beta function and I is the Regularized Incomplete Beta Function.

Mean

$$\frac{v_2}{v_2-2} \text{ for } v_2 > 2$$

Variance

$$\frac{2v_2^2(v_1+v_2-2)}{v_1(v_2-2)^2(v_2-4)} \text{ for } v_2 > 4$$

Skewness

$$\frac{(2v_1+v_2+2)}{(v_2-6)} \sqrt{\frac{8(v_2-4)}{v_1(v_1+v_2-2)}} \text{ for } v_2 > 6$$

Kurtosis

$$3 + 12 \left[\frac{(v_2-2)^2(v_2-4) + v_1(v_1+v_2-2)(5v_2-22)}{v_1(v_2-6)(v_2-8)(v_1+v_2-2)} \right] \text{ for } v_2 > 8$$

Mode

$$\frac{v_2(v_1-2)}{v_1(v_2+2)} \text{ for } v_1 > 2$$
$$0 \text{ for } v_1 \leq 2$$

PsiFrechet

`PsiFrechet (loc, scale, shape)`

The PsiFrechet distribution is used to model extreme events. This distribution is used commonly in hydrology to model peak annual rainfall and damn overflow.

Parameters

loc (λ) , scale (β) > 0, shape (α) > 0

Range of Function Values

(0, ∞)

Probability Density Function

$$f(x) = \frac{\alpha}{\beta} z^{-(1+\alpha)} \exp[-z^{-\alpha}] \text{ where } z \equiv \frac{x-y}{\beta}$$

Cumulative Distribution Function

$$F(x) = \exp[-z^{-\alpha}] \text{ where } z \equiv \frac{x-y}{\beta}$$

Mean

$$\gamma + \beta q_1 \text{ for } \alpha_1 > 1$$

Variance

$$\beta[q_2 - q_1^2] \text{ for } \alpha_2 > 2$$

Skewness

$$\frac{(q_3 + 3q_1q_2 + 2q_1^3)}{(q_2 - q_1^2)^{\frac{3}{2}}} \text{ for } \alpha_3 > 3$$

Kurtosis

$$\frac{q_4 - 4q_1q_3 + 6q_1^2q_2 - 3q_1^4}{(q_2 - q_1^2)^2} \text{ for } \alpha_4 > 4$$

Mode

$$\gamma + \beta \left(\frac{\alpha}{1+\alpha} \right)^{\frac{1}{\alpha_1}}$$

PsiGamma

PsiGamma ($\alpha, \beta, \dots$)

PsiGamma (α, β) is a flexible distribution with a finite lower bound and decreasing values.

PsiExponential, PsiErlang, and PsiChiSquare are special cases of PsiGamma, as explained below. The Gamma distribution is often used to model the time between events that occur with a constant average rate.

When $\alpha = 1$, the Gamma distribution is the same as an Exponential distribution. If the parameter α is integer, then the Gamma distribution is the same as the Erlang distribution. The Gamma distribution with $\alpha = a/2$, $\beta = 2$ is the same as a Chi Square distribution with parameter a (a degrees of freedom).

If $X_1, X_2, \dots, X_m$ are independent random variables with $X_i \sim \text{PsiGamma}(\alpha_i, \beta)$, then their sum also has a Gamma distribution with parameters $(\alpha_1 + \alpha_2 + \dots + \alpha_m, \beta)$. Additionally, the Gamma distribution approaches a normal distribution with the same mean and standard deviation as the parameter α approaches infinity.

Parameters

$$\alpha, \beta > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \begin{cases} \frac{\beta^{-\alpha} x^{\alpha-1} e^{-x/\beta}}{\Gamma(\alpha)} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$\Gamma(\alpha)$ is the Gamma Function

Cumulative Distribution Function

$$F(x) = \frac{\gamma\left(\alpha, x/\beta\right)}{\Gamma(\alpha)}$$

$\gamma(a, b)$ is the Incomplete Gamma Function

Mean

$$\beta\alpha$$

Variance

$$\beta^2\alpha$$

Skewness

$$\frac{2}{\sqrt{\alpha}}$$

Kurtosis

$$\frac{6}{\alpha} + 3$$

Median

Not defined

Mode

$$\begin{cases} \beta(1-\alpha) & \text{if } \alpha \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

PsiHypSecant

`PsiHypSecant (loc, scale)`

The PsiHypSecant distribution is similar to the normal distribution, but has a larger kurtosis resulting in a sharper peak.

Alternate Formulation: `PsiHypSecantAlt`

`PsiHypSecantAlt` is the `PsiHypSecant` distribution defined through alternative arguments. Two parameters are required and both must be used in the following combinations: two different percentiles, percentile and loc or percentile and scale.

Parameters

loc (λ) , scale (β) > 0

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{1}{2\beta} \operatorname{sech} \left[\frac{\pi}{2} z \right] \text{ where } z \equiv \frac{x-y}{\beta}$$

Cumulative Distribution Function

$$F(x) = \frac{2}{\pi} \tan^{-1} \left[e^{\frac{\pi z}{2}} \right] \text{ where } z \equiv \frac{x-y}{\beta}$$

Mean

$$\gamma$$

Variance

$$\beta^2$$

Skewness

$$0$$

Kurtosis

$$5$$

Mode

$$\gamma$$

PsiInvNormal

`PsiInvNormal` ($\mu, \lambda, \dots$)

`PsiInvNormal` (μ, λ) is a distribution with a finite lower bound, where it is zero. The Inverse Normal distribution is used to model Brownian motion and other diffusion processes. As the parameter λ tends to infinity, the Inverse Normal distribution approaches a Normal distribution.

Parameters

$$\mu, \lambda > 0$$

Range of Function Values

$$(0, \infty)$$

Probability Density Function

$$f(x) = \sqrt{\frac{\lambda}{2\pi x^3}} \left[e^{\left(\frac{-\lambda(x-\mu)^2}{2x\mu^2} \right)} \right]$$

Cumulative Distribution Function

$$F(x) = \Phi \left(\sqrt{\frac{\lambda}{x}} \left(\frac{x}{\mu} - 1 \right) \right) + e^{\left(\frac{2\lambda}{\mu} \right)} \Phi \left(-\sqrt{\frac{\lambda}{x}} \left(\frac{x}{\mu} + 1 \right) \right)$$

$\Phi(z)$ is the Error Function

Mean

$$\mu$$

Variance

$$\frac{\mu^3}{\lambda}$$

Skewness

$$3\sqrt{\frac{\mu}{\lambda}}$$

Kurtosis

$$15\frac{\mu}{\lambda} + 3$$

Median

Not defined

Mode

$$\mu \left\{ \left(1 + \frac{9\mu^2}{4\lambda^2} \right)^{\frac{1}{2}} - \frac{3\mu}{2\lambda} \right\}$$

PsiJohnsonSB

PsiJohnsonSB (shape1, shape2, min, max)

The PsiJohnsonSB distribution is a bounded distribution with shape parameters shape1 and shape2 and boundary parameters min and max.

Parameters

shape1(α_1), shape2 (α_2) > 0 , max(b) > min (a)

Range of Function Values

[a, b]

Probability Density Function

$$f(x) = \frac{\alpha_2(b-a)}{\sqrt{2\pi}(x-a)(b-x)} \times e^{\frac{-1}{2}[\alpha_1 + \alpha_2 \ln(\frac{x-a}{b-x})]}$$

Cumulative Distribution Function

$$F(x) = \Phi \left[\alpha_1 + \alpha_2 \ln \left(\frac{x-a}{b-x} \right) \right] \text{ where } \Phi \text{ is the cumulative distribution function of the standard}$$

Normal distribution (PsiNormal(0,1)).

Mean

N/A

Variance

N/A

Skewness

N/A

Kurtosis

N/A

Mode

No Closed Form

PsiJohnsonSU

PsiJohnsonSU (shape1, shape2, loc, scale)

The PsiJohnsonSU distribution is an unbounded distribution with shape parameters shape1 and shape2, location parameter loc and scale parameter, scale.

Parameters

shape1(α_1), shape2 (α_2) > 0 , loc, scale (β) > 0

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{\alpha_2}{\beta\sqrt{2\pi}(1+z^2)} \times e^{\frac{-1}{2}[\alpha_1+\alpha_2\sinh^{-2}(z)]^2}$$

Cumulative Distribution Function

$F(x) = \Phi[\alpha_1 + \alpha_2\sinh^{-1}(z)]$ where

Φ is the cumulative distribution function of the standard

Normal distribution (PsiNormal(0,1)).

Mean

$$\gamma - \beta\sqrt{\theta} \sinh(r)$$

Variance

$$\frac{\beta^2}{2} (\theta - 1)(\theta \cosh(2r) + 1)$$

Skewness

$$\frac{\frac{-1}{4}\sqrt{\theta}(0-1)^2[\theta(\theta+2)\sinh(3r)+3\sinh(r)]}{\left[\frac{1}{2}(\theta-1)(\theta\cosh(2r)+1)\right]^{\frac{3}{2}}}$$

Kurtosis

$$\frac{\frac{1}{8}(\theta-1)^2[\theta^2(\theta^4+2\theta^3+3\theta^2-3)\cosh(4r)+4\theta^2(\theta+2)\cosh(2r)+3(2\theta+1)]}{\left[\frac{1}{2}(\theta-1)(\theta\cosh(2r)+1)\right]^2}$$

Mode

No Closed Form

PsiKumaraswamy

PsiKumaraswamy (shape1, shape2, min, max)

PsiKumaraswamy is a bounded distribution using the defined min and max parameters. This distribution may be used as a simpler mathematical replacement for thePsiBetaGen distribution.

Parameters

shape1(α_1) > 0, shape2(α_2) > 0, min < max

Range of Function Values

(min, max)

Probability Density Function

$$f(x) = \frac{\alpha_1 \alpha_2}{(\max - \min)} z^{\alpha_1 - 1} (1 - z^{\alpha_1})^{\alpha_2 - 1} \text{ where } z \equiv \frac{x - \min}{\max - \min}$$

Cumulative Distribution Function

$$F(x) = 1 - (1 - F(x)) = 1 - (1 - z^{\alpha_1})^{\alpha_2} \text{ where } z \equiv \frac{x - \min}{\max - \min}$$

Mean

$$\min + (\max - \min) q_1$$

Variance

$$\beta^2 [q_2 - q_1^2]$$

Skewness

$$\frac{q_3 - 3q_1 q_2 + 2q_1^3}{(q_2 - q_1^2)^{\frac{3}{2}}}$$

Kurtosis

$$\frac{q_4 - 4q_1 q_3 + 6q_1^2 q_2 - 3q_1^4}{(q_2 - q_1^2)^2}$$

Mode

$$\min + (\max - \min) \left[\frac{\alpha_1 - 1}{\alpha_1 \alpha_2 - 1} \right]^{\frac{1}{\alpha_1}} \text{ for } \alpha_1 > 1, \alpha_2 > 1$$

$$\min \quad \text{for } \alpha_1 < 1, \alpha_2 \geq 1 \text{ or } \alpha_1 = 1, \alpha_2 > 1$$

$$\max \quad \text{for } \alpha_1 > 1, \alpha_2 \leq 1 \text{ or } \alpha_1 = 1, \alpha_2 < 1$$

$$\text{not defined} \quad \text{for } \alpha_1 < 1, \alpha_2 < 1 \text{ or } \alpha_1 = 1, \alpha_2 = 1$$

PsiLaplace

PsiLaplace (loc, β , ...)

PsiLaplace (β) is an unbounded, fat-tailed distribution that describes the difference between two independent exponentials. If a random variable X has a Laplace distribution, then |X| has an Exponential distribution.

Enter a scale value greater than 0 for the first argument, loc. This value determines the scale or range of the distribution.

Alternate Formulation: PsiLaplaceAlt

PsiLaplaceAlt is the PsiLaplace distribution defined through alternative arguments. Two parameters are required and both must be chosen from the following list: two different percentiles or the standard deviation.

Parameters

$$\beta > 0, \text{loc} > 0$$

Range of Function Values

$$(-\infty, \infty)$$

Probability Density Function

$$f(x) = \frac{e^{-\left(\frac{|x|}{\beta}\right)}}{2\beta}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 1 - \frac{1}{2} e^{\frac{-x}{\beta}} & \text{if } x \geq 0 \\ \frac{1}{2} e^{\frac{x}{\beta}} & \text{otherwise} \end{cases}$$

Mean

0

Variance

$$2\beta^2$$

Skewness

0

Kurtosis

3

Median

0

Mode

0

PsiLevy

`PsiLevy (loc, scale)`

PsiLevy is a continuous distribution with a location, loc, and a scale parameter, scale.

Alternate Formulation: PsiLevyAlt

PsiLevyAlt is the PsiLevy distribution defined through alternative arguments. Two parameters are required. The following combinations may be used: two different percentiles, percentile and loc or percentile and scale.

Parameters

loc (a), scale (c) > 0

Range of Function Values

(loc, ∞)

Probability Density Function

$$f(x) = \sqrt{\frac{c}{2\pi}} \frac{e^{\frac{c}{2(x-\mu)^2}}}{(x-\mu)^{\frac{3}{2}}} \text{ where } erf \text{ is the Error Function.}$$

Cumulative Distribution Function

$$F(x) = 1 - erf\left(\sqrt{\frac{c}{2(x-\mu)^2}}\right) \text{ where } erf \text{ is the Error Function}$$

Mean

Does not exist.

Variance

Does not exist.

Skewness

Does not exist.

Kurtosis

Does not exist.

Mode

$$\mu + \frac{c}{3}$$

PsiLaplace

PsiLaplace ($\beta, \dots$)

PsiLaplace (β) is an unbounded, fat-tailed distribution that describes the difference between two independent exponentials. If a random variable X has a Laplace distribution, then $|X|$ has an Exponential distribution.

Parameters

$$\beta > 0$$

Range of Function Values

$$(-\infty, \infty)$$

Probability Density Function

$$f(x) = \frac{e^{-\left(\frac{|x|}{\beta}\right)}}{2\beta}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 1 - \frac{1}{2}e^{-x/\beta} & \text{if } x \geq 0 \\ \frac{1}{2}e^{x/\beta} & \text{otherwise} \end{cases}$$

Mean

$$0$$

Variance

$$2\beta^2$$

Skewness

$$0$$

Kurtosis

3

Median

0

Mode

0

PsiLogistic

PsiLogistic ($\mu, s, \dots$)

PsiLogistic (μ, s) is an unbounded distribution, symmetric around its mean, with broader tails than the Normal distribution. The Logistic distribution is often used to model growth processes.

Parameters

μ

$s > 0$

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{e^{-\frac{(x-\mu)}{s}}}{s \left(1 + e^{-\frac{(x-\mu)}{s}} \right)^2}$$

Cumulative Distribution Function

$$F(x) = \frac{1}{1 + e^{-\frac{(x-\mu)}{s}}}$$

Mean

μ

Variance

$$\frac{\pi^2 s^2}{3}$$

Skewness

0

Kurtosis

6/5

Median

μ

Mode

μ

PsiLogLogistic

`PsiLogLogistic` ($\gamma, \beta, \alpha, \dots$)

`PsiLogLogistic` (γ, β, α) is a distribution with a finite lower bound. The natural log of `PsiLogLogistic` is a Logistic random variable. The Log-Logistic distribution can be used to model the time to perform a job or task.

Parameters

$\beta, \alpha > 0$

γ

Range of Function Values

$[\gamma, \infty)$

Probability Density Function

$$f(x) = \frac{\alpha \left(\frac{x - \gamma}{\beta} \right)^{\alpha - 1}}{\beta \left[1 + \left(\frac{x - \gamma}{\beta} \right)^\alpha \right]^2}$$

Cumulative Distribution Function

$$F(x) = \frac{1}{1 + \left(\frac{\beta}{x - \gamma} \right)^\alpha}$$

Mean

$$\frac{\beta \pi \operatorname{cosec} \left(\frac{\pi}{\alpha} \right)}{\alpha} + \gamma \text{ for } \alpha > 1$$

Variance

$$\frac{\beta^2 \pi \left[2 \operatorname{cosec} \left(\frac{2\pi}{\alpha} \right) - \frac{\pi}{\alpha} \operatorname{cosec}^2 \left(\frac{\pi}{\alpha} \right) \right]}{\alpha} \text{ for } \alpha > 2$$

Skewness

$$\frac{\left[3 \operatorname{cosec}\left(\frac{3\pi}{\alpha}\right) - \frac{6\pi}{\alpha} \operatorname{cosec}\left(\frac{2\pi}{\alpha}\right) \operatorname{cosec}\left(\frac{\pi}{\alpha}\right) + \frac{2\pi^2}{\alpha^2} \operatorname{cosec}^3\left(\frac{\pi}{\alpha}\right) \right]}{\left(\sqrt{\frac{\pi}{\alpha}} \left[2 \operatorname{cosec}\left(\frac{2\pi}{\alpha}\right) - \frac{\pi}{\alpha} \operatorname{cosec}^2\left(\frac{\pi}{\alpha}\right) \right] \right)^{3/2}} \text{ for } \alpha > 3$$

Kurtosis

Not defined

Median

Not defined

Mode

$$\begin{cases} \gamma + \beta \left[\frac{\alpha - 1}{\alpha + 1} \right]^{1/\alpha} & \text{for } \alpha > 1 \\ 0 & \text{otherwise} \end{cases}$$

PsiLogNormal

PsiLogNormal ($\mu, \sigma, \dots$)

PsiLogNormal (μ, σ) is a distribution with a finite lower bound and has mean μ and standard deviation σ . The LogNormal distribution can be used to model quantities that are products of many small independent variables. The natural log of PsiLogNormal is a Normal random variable.

Parameters

$\mu, \sigma > 0$

Range of Function Values

$[0, \infty)$

Probability Density Function

$$f(x) = \frac{e^{-\frac{1}{2} \left(\frac{\ln x - \mu'}{\sigma'} \right)^2}}{x \sqrt{2\pi} \sigma'}$$

$$\sigma' = \sqrt{\ln \left(1 + \frac{\sigma^2}{\mu^2} \right)}, \mu' = \ln \left(\frac{\mu^2}{\sqrt{\mu^2 + \sigma^2}} \right)$$

Cumulative Distribution Function

$$F(x) = \Phi \left(\frac{\ln x - \mu'}{\sigma'} \right)$$

$\Phi(a)$ is the Error Function

Mean

$$\mu$$

Variance

$$\sigma^2$$

Skewness

$$\frac{\sigma^3}{\mu^3} + \frac{3\sigma}{\mu}$$

Kurtosis

$$6\left(1 + \frac{\sigma}{\mu}\right)^4 + 2\left(1 + \frac{\sigma}{\mu}\right)^3 + 3\left(1 + \frac{\sigma}{\mu}\right)^2 - 3$$

Median

$$\frac{\mu^2}{\sqrt{\mu^2 + \sigma^2}}$$

Mode

$$\frac{\mu^4}{(\sigma^2 + \mu^2)^{3/2}}$$

PsiLogNorm2

PsiLogNorm2 ($\mu, \sigma, \dots$)

PsiLogNorm2 (μ, σ) is a distribution with a finite lower bound. It can be used to model quantities that are products of many small independent variables. The natural log of PsiLogNorm2 is a Normal random variable. In contrast to PsiLogNormal(), the parameters μ and σ of PsiLogNorm2() are the mean and standard deviation of the *corresponding Normal distribution*.

Parameters

$$\mu$$

$$\sigma > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \frac{e^{-\frac{1}{2}\left(\frac{\ln x - \mu}{\sigma}\right)^2}}{x\sqrt{2\pi}\sigma}$$

Cumulative Distribution Function

$$F(x) = \Phi\left(\frac{\ln x - \mu}{\sigma}\right)$$

$\Phi(a)$ is the Error Function

Mean

$$e^{\mu + \sigma^2/2}$$

Variance

$$(e^{\sigma^2} - 1)e^{2\mu + \sigma^2}$$

Skewness

$$(e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$$

Kurtosis

$$6e^{4\sigma^2} + 2e^{3\sigma^2} + 3e^{2\sigma^2} - 3$$

Median

$$e^{\mu}$$

Mode

$$e^{\mu - \sigma^2}$$

PsiMaxExtreme

`PsiMaxExtreme (m, s, ...)`

`PsiMaxExtreme (m,s)` is the positively skewed form of the Extreme Value distribution, which is the limiting distribution of a very large collection of random observations.

Parameters

m

s > 0

Range of Function Values

$$(-\infty, \infty)$$

Probability Density Function

$$f(x) = \frac{z}{s} e^{-z}, z = e^{\frac{-(x-m)}{s}}$$

Cumulative Distribution Function

Not defined

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiMetalog

`PsiMetalog(min, max, coefficients)`

A general meta-log distribution is defined in models through `PsiMetalog()`. This function has as arguments a vector of coefficients (`coefficients`) and optionally lower (`min`) and upper (`max`) bounds. The metalog distribution family is ***designed*** to be determined from historical data, without requiring a distribution fitting process. They are computed from a set of historical data pairs $\{y, x\}$, where y is a cumulative probability and x is the correspondent percentile. The `coefficients` argument is either a range or an array of 2 to 10 numerical elements.

There are 3 optional cases for this distribution:

- Both *min* and *max* are missing which results in an unbounded distribution

PsiMetalog(, , {33, 3.7, -2.4, 5.7, 10.1})

- Only *min* exists which results in a semi-bounded distribution

PsiMetalog(0, , {33, 3.7, -2.4, 5.7, 10.1})

- Both *min* and *max* exist which results in a bounded distribution

PsiMetalog(0, 55, {33, 3.7, -2.4, 5.7, 10.1})

Note: An exclusive bounding using only the `max` argument is not supported. The `min` and `max` arguments are true lower and upper bounds not to be confused with lower truncate and upper truncate. Optional standard property functions may be used with this distribution, i.e. `PsiTruncate`, `PsiShift`, `PsiCorrelate`, etc. The distribution may also be compounded.

Analytic moments are available only for unbounded distributions. Kurtosis is not computed for more than 5 coefficients.

Note: When deploying an uncertain variable defined using `PsiMetalog()`, no additional distribution properties (i.e. `PsiTruncate`, `PsiCensor`, etc.) will be exported *except* `PsiSeed()`.

Parameters

`min`, `max`, `coefficients`

Range of Function Values

$(-\infty, \infty)$

5-Term Quantile Function (Inverse CDF)

$$M_5(y) := \left(a_1 + \log \left[\frac{y}{1-y} \right] a_2 + \left(\frac{-1}{2} + y \right) \log \left[\frac{y}{1-y} \right] a_3 + \left(\frac{-1}{2} + y \right) a_4 + \left(\frac{-1}{2} + y \right)^2 a_5 \right)$$

See the website, <http://metalogdistributions.com/>, for more information.

2-Term Metalog Moments

$$u'_{1,2} := a_1 \quad \text{mean (first moment) of 2 term metalog}$$

$$u_{2,2} := \frac{1}{3} \pi^2 a_2^2 \quad \text{variance (2nd central moment) of 2 term metalog}$$

$$u_{3,2} := 0 \quad \text{skewness (3rd central moment) of 2 term metalog}$$

$$u_{4,2} := \frac{7}{15} \pi^4 a_2^4 \quad \text{kurtosis (4th central moment) of 2 term metalog}$$

See the website, <http://metalogdistributions.com/>, for more information.

3-Term Metalog Moments

$$u'_{1,3} := u'_{1,2} + \frac{a_3}{2} \quad \text{mean (first moment) of 3 term metalog}$$

$$u_{2,3} := u_{2,2} + \frac{a_3^2}{12} + \frac{1}{36} \pi^2 a_3^2 \quad \text{variance (2nd central moment) of 3 term metalog}$$

$$u_{3,3} := u_{3,2} + \pi^2 a_2^2 a_3 + \frac{1}{24} \pi^2 a_3^3 \quad \text{skewness (3rd central moment) of 3 term metalog}$$

kurtosis (4th central moment) of 3 term metalog

$$u_{4,3} := u_{4,2} + \frac{3}{2} \pi^2 a_2^2 a_3^2 + \frac{7}{30} \pi^4 a_2^2 a_3^2 + \frac{a_3^4}{80} + \frac{1}{24} \pi^2 a_3^4 + \frac{7\pi^4 a_3^4}{1200}$$

See the website, <http://metalogdistributions.com/>, for more information.

PsiMetalog2

PsiMetalog2 (coefficients)

Returns a variate from a meta-logistic distribution approximated by a table of historical data.

The coefficients argument is either a range or an array of two to sixteen (2 to 16) numerical elements.

PsiMetalog2 differs from PsiMetalog in the following ways.

1. Fitting a distribution using PsiMetalog2Fit guarantees a feasible or valid distribution.
2. Fitting is improved when using the PsiMetalog2 distribution, even though PsiMetalog2 lacks bound arguments.
3. PsiMetalog2 behaves the same as PsiMetalog when there are six or fewer coefficients ($n \leq 6$). All statistics and curves will be identical for $n \leq 6$.
4. PsiMetalog2 has three additional statistics: modes, anti-modes and feasible.

Parameters

coefficients

Range of Function Values

$(-\infty, \infty)$

7-Term Quantile Function

See the website, <http://metalogdistributions.com/>, for more information.

Feasibility

Indicates whether the distribution, as defined by its coefficients, is feasible or infeasible.

Feasibility = True: Indicates the fitting succeeded with a feasible distribution, making it suitable for use in a simulation model.

Feasibility = False: Indicates that the fitting succeeded with an infeasible distribution, making it not suitable for use in a simulation model.

Feasibility = N/A: Indicates that fitting failed. The reason is unknown. The distribution with returned coefficients should not be used in a simulation model.

Modes

Returns all local maximums (for the x-axis) on the PDF function. Only Metalog distributions may have more than one mode. This is one major reason why Metalog distributions are superior in fitting complicated data.

Anti-modes

Returns all local minimums on the PDF function. Only Metalog distributions may have anti-modes; other distributions do not exhibit anti-modes. Note: There must be 2 modes and more than 5 coefficients in order for an anti-mode to exist.

See the website, <http://metalogdistributions.com/>, for more information.

PsiMetalogFit

`PsiMetalogfit(num_coef, x_values, y_values)`

PsiMetalogFit() computes 2 to 10 coefficients for PsiMetalog by fitting m pairs representing a CDF. Enter this function as an array formula.

In Analytic Solver Cloud, PsiMetalogFit() returns a [Dynamic Array](#). To use this function in the Cloud, you need only enter the Psi function in one cell as a normal function, i.e., not as a control array. The contents of the Dynamic Array will "spill" down the column. If a nonblank cell is "blocking" the contents of the Dynamic Array, PsiMetalogFit() will return #SPILL until such time as the blockage is removed. When used in the Cloud apps, the first argument, *numCoef*, is a scalar, the number of coefficients to produce.

Example: The formula PsiMetalogFit(5, A2:E2, A1:E1) is array entered into cells A3:E3 where the Y values are contained in cells A1:E1 and the X values are contained in cells A2:E2 column. This function computes 5 coefficients by fitting 5 pairs. Note: User must enter the function as an array formula in a horizontal or vertical range-vector with num_coef cells (in order to see all requested coefficients).

A1 = 0.010, A2 = 10

B1 = 0.100, B2 = 22

C1 = 0.500, C2 = 33

D1 = 0.900, D2 = 43

E1 = 0.990, E2 = 50

Parameters

num_coef, x_values, y_values

Range of Function Values

$(-\infty, \infty)$

PsiMetalog2Fit

`PsiMetalog2Fit(num_coef, x_values, y_values)`

PsiMetalog2Fit() computes 2 to 16 coefficients for PsiMetalog2 by fitting *m* pairs of historical data points and associated probabilities, representing a cumulative distribution function, or CDF (monotonic and increasing). This function only fits for the `num_coef` specified. Note: This function is computationally expensive. It is recommended to use the Fit dialog, instead of this function, for fitting a distribution to a Metalog2 distribution.

PsiMetalog2Fit accepts three arguments: `num_coef`, `x_values` and `y_values`.

numCoef: A scalar; the number of coefficients to produce.

- The minimum value for this argument is 2 and the maximum is 16.

$$2 \leq \text{numCoef} \leq 16$$

- The value for this argument must be less than or equal to the length of the `x_values` and `y_values` arrays.

$$\text{numCoef} \leq \text{Length}(\text{x\_values}/\text{y\_values})$$

x_values: An array containing the historical data points. There is no limit on the number of historical data points that may be supplied.

- Values must be unique. No values may be repeated in the historical data.
- Missing values are not supported.

y_values: (Optional) Contains the associated probabilities for the historical data.

- Probabilities must be unique and should occur in ascending order.
- Probabilities must be between 0 and 1.

$$0 < \text{y\_values} < 1$$

- Missing or 0 values are not supported in the `y_values` array.

$$\text{y\_values} \neq 0$$

PsiMetalog2Fit() returns a [Dynamic Array](#). You need only enter the Psi function in one cell as a normal function, i.e., rather than a control array. The contents of the Dynamic Array will "spill" down the column. If a nonblank cell is "blocking" the contents of the Dynamic Array, PsiMetalog2Fit() will return #SPILL until such time as the blockage is removed.

Example: The formula PsiMetalog2Fit(5, A2:E2, A1:E1) is entered into cell A3 where the Y values are probabilities (between 0 and 1) contained in cells A1:E1 and the X_values are contained in cells A2:E2 column. This function computes 5 coefficients by fitting 5 pairs. Note: User must enter the function as an array formula in a horizontal or vertical range-vector with `num_coef` cells (in order to see all requested coefficients).

A1 = 0.010, A2 = 10

B1 = 0.100, B2 = 22

C1 = 0.500, C2 = 33

D1 = 0.900, D2 = 43

E1 = 0.990, E2 = 50

Parameters

`num_coef`, `x_values`, `y_values`

Range of Function Values

$(-\infty, \infty)$

PsiMetalogSPT

`PsiMetalogSPT(min, max, quantiles, probability)`

This distribution is constructed by a symmetric-percentile triplet (SPT). The cumulative probabilities are {prob, 0.5, 1-prob}, where $0 < \text{prob} < 0.5$. The corresponding quantile set is a 3-element array/range of quantiles. The optional *min* and *max* arguments have the same meaning of lower and upper bounds as in the PsiMegalog. This distribution can be truncated, shifted, correlated, compounded etc. through the use of property functions.

There are 3 optional cases for this distribution:

- Both *min* and *max* are missing which results in an unbounded distribution

PsiMetalogSPT(, , {35, 50, 75}, 0.1)

- Only *min* exists which results in a semi-bounded distribution

PsiMetalogSPT(0, , {35, 50, 75}, 0.1) semi-bounded

- Both *min* and *max* exist which results in a bounded distribution

PsiMetalogSPT(0, 100, {35, 50, 75}, 0.1)

Note: Analytic moments are available only for unbounded distributions. Kurtosis is always computed.

Parameters

min, max, quantiles, prob

Range of Function Values

$(-\infty, \infty)$

Metalog Quantile Function where $n = 3$

$$M^{\text{logit}}(y) = (b_l + b_u e^{M(y)}) / (1 + e^{M(y)}) \quad \text{for } 0 < y < 1$$

$$= b_l \quad \text{for } y = 0$$

$$= b_u \quad \text{for } y = 1$$

Metalog PDF

$$m^{\text{logit}}(y) = m(y) (1 + e^{M(y)})^2 / ((b_u - b_l) e^{M(y)}) \quad \text{for } 0 < y < 1$$
$$= 0 \quad \text{for } y = 0 \text{ or } y = 1$$

where b_l and b_u are user-specified lower and upper bounds respectively and

$$M(y) = a_1 + a_2 \ln \left(\left(\frac{y}{1-y} \right) + a_3 (y - 0.5) \ln \left(\frac{y}{1-y} \right) \right)$$

$$m(y) = \frac{1}{\left[\frac{a_2}{y(1-y)} + a_3 \left(\frac{y-0.5}{y(1-y)} + \ln \left(\frac{y}{1-y} \right) \right) \right]}$$

The constants (a_1, a_2, a_3) can be calculated directly from SPT input parameters:

$$a_1 = \ln(\gamma_{0.5})$$

$$a_2 = \frac{1}{2} \left[\ln \frac{1-\alpha}{\alpha} \right]^{-1} \ln \left[\frac{\gamma_{1-\alpha}}{\gamma_{0.5}^2} \right]$$

$$a_3 = \left[(1 - 2\alpha) \ln \left(\frac{1 - \alpha}{\alpha} \right) \right]^{-1} \ln \left(\frac{\gamma_{1-\alpha} \gamma_\alpha}{\gamma_{0.5}^2} \right)$$

$$\text{where } y_\alpha = \frac{q_\alpha - b_l}{b_u - q_\alpha}, y_{0.5} = \frac{q_{0.5} - b_l}{b_u - q_{0.5}}, y_{1-\alpha} = \frac{q_{1-\alpha} - b_l}{b_u - q_{1-\alpha}}$$

The feasibility requirement is given by:

$$[b_l + b_u \gamma_\alpha^{1-k_\alpha} \gamma_{1-\alpha}^{k_\alpha}] [1 + \gamma_\alpha^{1-k_\alpha} \gamma_{1-\alpha}^{k_\alpha}]^{-1} < q_{0.5} < [b_l + b_u \gamma_\alpha^{k_\alpha} \gamma_{1-\alpha}^{1-k_\alpha}] [1 + \gamma_\alpha^{k_\alpha} \gamma_{1-\alpha}^{1-k_\alpha}]^{-1}$$

$$\text{where } k_\alpha = \frac{1}{2} \left[1 - 1.66711 \left(\frac{1}{2} - \alpha \right) \right]$$

See the website, <http://metalogdistributions.com/>, for more information.

PsiMinExtreme

PsiMinExtreme (m, s, ...)

PsiMinExtreme is the negatively skewed form of the Extreme Value distribution, which is the limiting distribution of a very large collection of random observations.

Parameters

m

s > 0

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{z}{s} e^{-z}, z = e^{\frac{(x-m)}{s}}$$

Cumulative Distribution Function

Not defined

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiMyerson

`PsiMyerson (a,c,b,t,...)`

PsiMyerson(a,c,b,t) is a generalized LogNormal/Normal distribution, specified using the top (a), bottom (b) and 50th (c) percentiles, and an optional tail percentage parameter (t). This distribution is bounded on the side of the narrower percentile range; when both the bottom and top percentile ranges are equal, then this distribution is unbounded.

This distribution, developed by Dr. Roger Myerson, is used to model random variables when the only information available is the percentile values, and optionally, a tail percentage parameter indicating the probability of values being within the specified percentiles. If the specified percentiles are equidistant (measured by the parameter b' below), then the Myerson distribution is equivalent to a Normal distribution. When the 50th percentile is equal to the geometric mean of the top and bottom percentiles, then the Myerson distribution is equivalent to the LogNormal distribution.

Parameters

$$a < c < b$$

$$t \in (0,1)$$

If t is omitted, it is given a default value of 0.5

Range of Function Values

$$[LB, UB)$$

where,

$$LB = -\infty, UB = \infty \text{ if } b' = 1$$

$$LB = c - \frac{(b-c)}{b'-1}, UB = \infty \text{ if } b' > 1$$

$$LB = -\infty, UB = c - \frac{(b-c)}{b'-1} \text{ if } b' < 1$$

and,

$$b' = \frac{b-c}{c-a}$$

Probability Density Function

If $b' \neq 1$,

$$f(x) = \frac{z_{(1-\alpha'/2)}(b'-1)}{((b-c)+(x-c)(b'-1))\ln(b')} f_{N(0,1)}(q)$$

where

$f_{N(0,1)}(q)$ is the PDF of the Standard Normal distribution,

$$q = Z_{(1-\alpha'/2)} \left\{ \frac{\ln \left(\frac{(x-c)(b'-1)}{(b-c)} + 1 \right)}{\ln(b')} \right\}$$

If $b' = 1$,

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

where

$$\mu = c$$

$$\sigma = \frac{(b-c)}{Z_{(1-\alpha'/2)}}$$

and

Z_x = CDFInverse of the Standard Normal distribution at x

Cumulative Distribution Function

If $b' \neq 1$,

$$F(x) = F_{N(0,1)}(q) = \frac{1}{2} \left(\Phi\left(\frac{q}{\sqrt{2}}\right) + 1 \right)$$

where

$F_{N(0,1)}(q)$ is the CDF of the Standard Normal distribution

$$\Phi(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \text{ is the Error function}$$

$$q = Z_{(1-1/2)} \left\{ \frac{\ln\left(\frac{(x-c)(b'-1)}{(b-c)} + 1\right)}{\ln(b')} \right\}$$

If $b' = 1$,

$$F(x) = \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right) \right]$$

where

$\operatorname{erf}(y)$ is the Error Function, and

$$\mu = c$$

$$\sigma = \frac{(b-c)}{Z_{(1-1/2)}}$$

and

Z_x = CDFInverse of the Standard Normal distribution at x

Mean

No closed form

Variance

No closed form

Skewness

No closed form

Kurtosis

No closed form

Median

No closed form

Mode

No closed form

PsiNormal

PsiNormal ($\mu, \sigma, \dots$)

PsiNormal (μ, σ) is an unbounded, symmetric distribution with the familiar *bell curve*, also called a Gaussian distribution. The Normal distribution is widely used in many different kinds of applications. A normal distribution with mean zero and standard deviation one is called a *Standard* normal distribution.

The sum of independent random variables of any shape tends to the Normal distribution.

Parameters

μ

$\sigma > 0$

Range of Function Values

$(-\infty, \infty)$

Probability Density Function

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

Cumulative Distribution Function

$$F(x) = \frac{1}{2} \left[1 + \operatorname{erf} \left(\frac{x-\mu}{\sigma\sqrt{2}} \right) \right]$$

$\operatorname{erf}(y)$ is the Error Function

Mean

μ

Variance

σ^2

Skewness

0

Kurtosis

0

Median

μ

Mode

μ

PsiPareto

PsiPareto ($\theta, a, \dots$)

PsiPareto (θ, a) is a distribution with a finite lower bound a , and shape parameter θ . The Pareto distribution can be used to describe or model wealth distribution, sizes of particles, etc. It is the exponential of an Exponential random variable.

Parameters

$$\theta, a > 0$$

Range of Function Values

$$[a, \infty)$$

Probability Density Function

$$f(x) = \frac{\theta a^\theta}{x^{\theta+1}}$$

Cumulative Distribution Function

$$F(x) = 1 - \left(\frac{a}{x}\right)^\theta$$

Mean

$$\frac{a\theta}{\theta-1} \text{ for } \theta > 1$$

Variance

$$\frac{\theta a^2}{(\theta-1)^2 (\theta-2)} \text{ for } \theta > 2$$

Skewness

$$\frac{2(\theta+1)}{(\theta-3)} \sqrt{\frac{(\theta-2)}{\theta}} \text{ for } \theta > 3$$

Kurtosis

$$\frac{3(\theta-2)(3\theta^2 + \theta + 2)}{\theta(\theta-3)(\theta-4)} \text{ for } \theta > 4$$

Median

$$a 2^{1/\theta}$$

Mode

$$a$$

PsiPareto2

PsiPareto2 (b, q, ...)

PsiPareto2 is an alternate form of the Pareto distribution with a finite lower bound of 0, and a shape parameter q . Like PsiPareto (θ, a), it can be used to describe or model wealth distribution, sizes of particles, etc. It is the exponential of an Exponential random variable.

Parameters

$$b, q > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \frac{qb^q}{(x+b)^{q+1}}$$

Cumulative Distribution Function

$$F(x) = 1 - \left(\frac{b}{x+b} \right)^q$$

Mean

$$\frac{b}{q-1} \text{ for } q > 1$$

Variance

$$\frac{qb^2}{(q-1)^2(q-2)} \text{ for } q > 2$$

Skewness

$$\frac{2(q+1)}{(q-3)} \sqrt{\frac{(q-2)}{q}} \text{ for } q > 3$$

Kurtosis

Not defined

Median

$$\frac{b}{q} 2^{1/q}$$

Mode

$$0$$

PsiPearson5

PsiPearson5 ($\alpha, \beta, \dots$)

PsiPearson5 (α, β) is a distribution with a lower bound of 0, and density similar to that of the LogNormal distribution. The Pearson5 distribution is sometimes called the Inverse Gamma

distribution. It can be used to model time delays when these can possibly take on unbounded (or very large) values.

Parameters

$$\alpha, \beta > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \frac{x^{-(\alpha+1)} e^{-\beta/x}}{\beta^{-\alpha} \Gamma(\alpha)}$$

Cumulative Distribution Function

$$F(x) = 1 - F_G\left(\frac{1}{x}\right)$$

$F_G(y)$ is the Distribution function of a Gamma $\left(\alpha, 1/\beta\right)$

random variable

Mean

$$\frac{\beta}{\alpha - 1} \text{ for } \alpha > 1$$

Variance

$$\frac{\beta^2}{(\alpha - 1)^2 (\alpha - 2)} \text{ for } \alpha > 2$$

Skewness

$$\frac{4\sqrt{\alpha - 2}}{\alpha - 3} \text{ for } \alpha > 3$$

Kurtosis

$$\frac{3(\alpha + 5)(\alpha - 2)}{(\alpha - 3)(\alpha - 4)} \text{ for } \alpha > 4$$

Median

Not defined

Mode

$$\frac{\beta}{\alpha + 1}$$

PsiPearson6

PsiPearson6 ($\alpha_1, \alpha_2, \beta, \dots$)

PsiPearson6 ($\alpha_1, \alpha_2, \beta$) is a distribution with a lower bound of 0, and a mode just beyond the lower bound. The Pearson6 distribution is sometimes called the Beta distribution of the second kind.

If $X_1 \sim \text{Gamma}(\alpha_1, \beta)$ and $X_2 \sim \text{Gamma}(\alpha_2, 1)$ are independent random variables, then X_1/X_2 has a Pearson6 distribution. If X is a random variable with a Pearson6 ($\alpha_1, \alpha_2, 1$) distribution, then $X/(1+X)$ has a Beta (α_1, α_2) distribution.

Parameters

$$\alpha_1, \alpha_2, \beta > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \frac{\left(\frac{x}{\beta}\right)^{\alpha_1-1}}{\beta B(\alpha_1, \alpha_2) \left[1 + \left(\frac{x}{\beta}\right)\right]^{\alpha_1+\alpha_2}}$$

$B(\alpha_1, \alpha_2)$ is the Beta function

Cumulative Distribution Function

$$F(x) = F_B\left(\frac{x}{x+\beta}\right)$$

$F_B(y)$ is the Distribution function of a Beta (α_1, α_2) random variable

Mean

$$\frac{\beta\alpha_1}{\alpha_2-1} \text{ for } \alpha_2 > 1$$

Variance

$$\frac{\beta^2\alpha_1(\alpha_1+\alpha_2-1)}{(\alpha_2-1)^2(\alpha_2-2)} \text{ for } \alpha_2 > 2$$

Skewness

$$\sqrt{\frac{\alpha_2-2}{\alpha_1(\alpha_1+\alpha_2-1)}} \left(\frac{4\alpha_1+2\alpha_2-2}{\alpha_2-3} \right) \text{ for } \alpha_2 > 3$$

Kurtosis

$$\frac{3(\alpha_2 - 2)}{(\alpha_2 - 3)(\alpha_2 - 4)} \left[\frac{2(\alpha_2 - 1)^2 + \alpha_1(\alpha_1 + \alpha_2 - 1)(\alpha_2 + 5)}{\alpha_1(\alpha_1 + \alpha_2 - 1)} \right] \text{ for } \alpha_2 > 4$$

Median

Not defined

Mode

$$\begin{cases} \frac{\beta(\alpha_1 - 1)}{\alpha_2 + 1} & \text{if } \alpha_1 \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

PsiPert

PsiPert (a, c, b, ...)

PsiPert (a, c, b) is a form of the Beta distribution, often used to estimate project completion times in the Program Evaluation and Review Technique, where a is the minimum time, b is the maximum time, and c is the most likely time. These parameters are used to compute the shape parameters α_1, α_2 used in the PsiBeta function, as shown below.

Parameters

$$a < c < b$$

The shape parameters α_1, α_2 can be defined as

$$\alpha_1 = \frac{-5a + b + 4c}{6(b - a)}$$

$$\alpha_2 = \frac{-a + 5b - 4c}{6(b - a)}$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \frac{(x - a)^{\alpha_1 - 1} (b - x)^{\alpha_2 - 1}}{B(\alpha_1, \alpha_2)(b - a)^{\alpha_1 + \alpha_2 - 1}}$$

$B(\alpha_1, \alpha_2)$ is the Beta function

Cumulative Distribution Function

$$F(x) = \frac{B\left(\frac{x-a}{b-a}\right)(\alpha_1, \alpha_2)}{B(\alpha_1, \alpha_2)}$$

$B(\alpha_1, \alpha_2)$ is the Beta function

$B_x(\alpha_1, \alpha_2)$ is the Incomplete Beta function

Mean

$$\frac{a+b+4c}{6}$$

Variance

$$\frac{(b-a)^2 \alpha_1 \alpha_2}{252}$$

Skewness

$$\frac{a+b-4c}{(b-a)} \sqrt{\frac{7}{\alpha_1 \alpha_2}}$$

Kurtosis

$$\frac{3(\alpha_1 + \alpha_2 + 1) \left[2(\alpha_1 + \alpha_2)^2 + \alpha_1 \alpha_2 (\alpha_1 + \alpha_2 - 6) \right]}{\alpha_1 \alpha_2 (\alpha_1 + \alpha_2 + 3)(\alpha_1 + \alpha_2 + 2)}$$

Median

Not defined

Mode

c

PsiRayleigh

`PsiRayleigh ( $\beta, \dots$ )`

`PsiRayleigh ( $\beta$ )` is a distribution with a finite lower bound of 0, a special case of a Weibull distribution. The Rayleigh distribution can be used to model component lifetimes.

If X is a random variable with Rayleigh distribution with parameter $\beta = 1$, then X^2 has a Chi Square distribution with parameter 2 (two degrees of freedom). If X and Y are independent normally distributed random variables with mean zero and variance σ^2 , then $(X^2 + Y^2)^{1/2}$ has a Rayleigh distribution with parameter σ . Thus, a Rayleigh distribution may be used to model the length of a two-dimensional vector whose components are independent and normally distributed.

Parameters

$$\beta > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \frac{xe^{\left(-x^2/2\beta^2\right)}}{\beta^2}$$

Cumulative Distribution Function

$$F(x) = 1 - e^{\left(-x^2/2\beta^2\right)}$$

Mean

$$\beta\sqrt{\frac{\pi}{2}}$$

Variance

$$\frac{(4-\pi)\beta^2}{2}$$

Skewness

$$\frac{2\sqrt{\pi}(\pi-3)}{(4-\pi)^{3/2}}$$

Kurtosis

$$\frac{-3\pi^2+32}{(4-\pi)^2}$$

Median

$$\beta\sqrt{\ln(4)}$$

Mode

$$\beta$$

PsiReciprocal

`PsiReciprocal (min, max)`

PsiReciprocal is a bounded continuous distribution with min and max parameters. This distribution is typically used in numerical analysis, Bayesian statistics and analysis of noise.

Parameters

Min < max

Range of Function Values

[min, max]

Probability Density Function

$$f(x) = \frac{1}{x[\ln(max)-\ln(min)]}$$

Cumulative Distribution Function

$$F(x) = \frac{\ln(x) - \ln(\min)}{\ln(\max) - \ln(\min)}$$

Mean

$$q_1$$

Variance

$$q_1 - q_1^2$$

Skewness

$$\frac{q_3 - 3q_1q_2 + 2q_1^3}{(q_2 - q_1^2)^{\frac{3}{2}}}$$

Kurtosis

$$\frac{q_4 - 4q_1q_3 + 6q_1^2q_2 - 3q_1^4}{(q_2 - q_1^2)^{\frac{3}{2}}}$$

Mode

$$\min$$

PsiStudent

PsiStudent (df, ...)

PsiStudent (df) is an unbounded distribution, symmetric about zero, with a shape similar to that of a Standard Normal distribution, and it approaches the Standard Normal distribution as the degrees of freedom (parameter df) increases. It is also known as the t-distribution, or Student's t-distribution.

The Student or t-distribution frequently arises when estimating the mean of a normally distributed population when the sample size is small. It is also used when the population variance is unknown, and is estimated from a small sample.

Parameters

$df > 0$, integer

Range of Function Values

$$(-\infty, \infty)$$

Probability Density Function

$$f(x) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{\pi df} \Gamma\left(\frac{df}{2}\right)} \left(\frac{df}{x^2 + df}\right)^{\frac{df+1}{2}}$$

$\Gamma(x)$ is the Gamma function

Cumulative Distribution Function

$$F(x) = \frac{1 + B\left(\frac{x^2}{x^2 + df}\right)\left(\frac{1}{2}, \frac{df}{2}\right)}{2}$$

$B_a(x, y)$ is the Incomplete Beta function

Mean

0 if $df > 1$

undefined if $df = 1$

Variance

$$\frac{df}{df - 2} \text{ if } df > 2$$

Skewness

0 if $df > 3$

Kurtosis

$$\frac{3df - 6}{df - 4} \text{ if } df > 4$$

Median

0

Mode

0

PsiTriangGen

`PsiTriangGen (ap, m, br, p, r...)`

`PsiTriangGen (ap, m, br, p, r...)` is a Triangular distribution where the lower and upper bounds are not given as fixed values, but are specified using percentiles. This distribution is usually used to create rough models in situations where little or no data is available. The distribution has a most likely value of m , the p (lower) percentile value is a_p , and the r (upper) percentile value is b_r . Given these values, a `PsiTriangGen (ap, m, br, p, r...)` distribution corresponds to a `PsiTriangular (a, c, b)` distribution with values for the bounds (a and b), and the most likely value (c) computed as shown below.

Parameters

$$p, r \in (0, 1)$$

$$p < r$$

$$a_p < b_r$$

$$a_p \leq m \leq b_r$$

The parameters of the Triangular distribution are defined as

$$a = \frac{a_p - m \sqrt{\frac{p}{q}}}{1 - \sqrt{\frac{p}{q}}}$$

$$c = m$$

$$b = \frac{b_r - m \sqrt{\frac{1-r}{1-q}}}{1 - \sqrt{\frac{1-r}{1-q}}}$$

Here q is a solution to the following equation

$$q = \frac{(m - a_p) \left(1 - \sqrt{\frac{1-r}{1-q}} \right)}{(b_r - m) \left(1 - \sqrt{\frac{p}{q}} \right) + (m - a_p) \left(1 - \sqrt{\frac{1-r}{1-q}} \right)}$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \begin{cases} \frac{2(x-a)}{(c-a)(b-a)} & \text{if } a \leq x \leq c \\ \frac{2(b-x)}{(b-c)(b-a)} & \text{if } c \leq x \leq b \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} \frac{(x-a)^2}{(c-a)(b-a)} & \text{if } a \leq x \leq c \\ 1 - \frac{(b-x)^2}{(b-c)(b-a)} & \text{if } c \leq x \leq b \end{cases}$$

Mean

$$\frac{a+b+c}{3}$$

Variance

$$\frac{a^2 + b^2 + c^2 - ab - ac - bc}{18}$$

Skewness

$$\frac{\sqrt{2}(a+b-2c)(2a-b-c)(a-2b+c)}{5(a^2 + b^2 + c^2 - ab - ac - bc)^{3/2}}$$

Kurtosis

$$12/5$$

Median

$$\begin{cases} a + \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \geq \frac{b-a}{2} \\ b - \sqrt{\frac{(b-a)(b-c)}{2}} & \text{otherwise} \end{cases}$$

Mode

$$c$$

PsiTriangular

`PsiTriangular (a,c,b,...)`

PsiTriangular (a,c,b) is a distribution with lower bound a, upper bound b, and most likely value c. This distribution is usually used to create rough models in situations where little or no data is available. If the parameter c = b, then the distribution is also known as a Right Triangular distribution. If the parameter c = a, then the distribution is also known as a Left Triangular distribution. If X₁ and X₂ are independent Uniform (0,1) random variables, then (X₁+X₂)/2 has a Triangular (0,0.5,1) distribution.

Parameters

$$a \leq c \leq b$$

$$a < b$$

Range of Function Values

$$[a,b]$$

Probability Density Function

$$f(x) = \begin{cases} \frac{2(x-a)}{(c-a)(b-a)} & \text{if } a \leq x \leq c \\ \frac{2(b-x)}{(b-c)(b-a)} & \text{if } c \leq x \leq b \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} \frac{(x-a)^2}{(c-a)(b-a)} & \text{if } a \leq x \leq c \\ 1 - \frac{(b-x)^2}{(b-c)(b-a)} & \text{if } c \leq x \leq b \end{cases}$$

Mean

$$\frac{a+b+c}{3}$$

Variance

$$\frac{a^2 + b^2 + c^2 - ab - ac - bc}{18}$$

Skewness

$$\frac{\sqrt{2}(a+b-2c)(2a-b-c)(a-2b+c)}{5(a^2 + b^2 + c^2 - ab - ac - bc)^{3/2}}$$

Kurtosis

$$12/5$$

Median

$$\begin{cases} a + \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \geq \frac{b-a}{2} \\ b - \sqrt{\frac{(b-a)(b-c)}{2}} & \text{otherwise} \end{cases}$$

Mode

$$c$$

PsiUniform

`PsiUniform (a,b,...)`

`PsiUniform (a,b)` is a flat, bounded distribution with lower bound `a` and upper bound `b`. It is used to represent a random variable that is equally likely to take on any value between a lower and upper

bound. A Uniform (0,1) distribution is also known as a Standard Uniform distribution, and is used to generate many other random variables. If X is a random variable with a Standard Uniform distribution, then $a + (b - a)X$ has a Uniform (a,b) distribution, and $(1 - X)$ has a Standard Uniform distribution.

Parameters

$$a < b$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b \\ 1 & \text{if } x > b \end{cases}$$

Mean

$$\frac{a+b}{2}$$

Variance

$$\frac{(b-a)^2}{12}$$

Skewness

$$0$$

Kurtosis

$$9/5$$

Median

$$\frac{a+b}{2}$$

Mode

Any value in $[a, b]$

PsiWeibull

PsiWeibull ($\alpha, \beta, \dots$)

PsiWeibull (α, β) is a distribution with a finite lower bound of 0. The Weibull distribution is quite flexible and can be used to model weather patterns, material strength, processing and delivery times, and in a variety of reliability engineering applications.

If X is a random variable with Weibull ($1, \beta$) distribution, then it also has the Exponential (β) distribution. In fact, a random variable $X \sim \text{Weibull}(\alpha, \beta)$ if and only if $X^\alpha \sim \text{Exponential}(\beta^\alpha)$. Also, if X is a random variable with Weibull ($2, \beta$) distribution, then it also has the Rayleigh (β) distribution.

Parameters

$$\alpha, \beta > 0$$

Range of Function Values

$$[0, \infty)$$

Probability Density Function

$$f(x) = \alpha \beta^{-\alpha} x^{\alpha-1} e^{-\left(\frac{x}{\beta}\right)^\alpha}$$

Cumulative Distribution Function

$$F(x) = 1 - e^{-\left(\frac{x}{\beta}\right)^\alpha}$$

Mean

$$\frac{\beta}{\alpha} \Gamma\left(\frac{1}{\alpha}\right)$$

$\Gamma(x)$ is the Gamma function

Variance

$$\frac{\beta^2}{\alpha} \left[2\Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha} \Gamma^2\left(\frac{1}{\alpha}\right) \right]$$

Skewness

$$\frac{\frac{3}{\alpha} \Gamma\left(\frac{3}{\alpha}\right) + \frac{6}{\alpha^2} \Gamma\left(\frac{2}{\alpha}\right) \Gamma\left(\frac{1}{\alpha}\right) + \frac{2}{\alpha^3} \Gamma^3\left(\frac{1}{\alpha}\right)}{\left[\frac{2}{\alpha} \Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha^2} \Gamma^2\left(\frac{1}{\alpha}\right) \right]^{3/2}}$$

Kurtosis

$$\frac{-\frac{6}{\alpha^2} \Gamma\left(\frac{1}{\alpha}\right) + \frac{24}{\alpha} \Gamma^2\left(\frac{1}{\alpha}\right) \Gamma\left(\frac{2}{\alpha}\right) - 12\Gamma^2\left(\frac{2}{\alpha}\right) - 12\Gamma\left(\frac{1}{\alpha}\right) \Gamma\left(\frac{3}{\alpha}\right) + 4\alpha \Gamma\left(\frac{4}{\alpha}\right)}{\left[2\Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha} \Gamma^2\left(\frac{1}{\alpha}\right) \right]^2}$$

Median

$$\beta(\ln(2))^{\frac{1}{\alpha}}$$

Mode

$$\beta\left(\frac{\alpha-1}{\alpha}\right)^{\frac{1}{\alpha}} \text{ if } \alpha \geq 1$$

0 otherwise

Discrete Analytic Distributions

PsiBernoulli

`PsiBernoulli (p, ...)`

PsiBernoulli (p) is a discrete distribution that takes on a value of 1 with probability p, and a value of 0 with probability (1-p). A Bernoulli random variable is usually considered as an outcome of an experiment with only two possible outcomes (0 and 1); each experiment is called a ‘Bernoulli Trial’.

Parameters

$$p \in [0,1]$$

Range of Function Values

$$\{0,1\}$$

Probability Mass Function

$$p(x) = \begin{cases} 1-p & \text{if } x = 0 \\ p & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1-p & \text{if } 0 \leq x < 1 \\ 1 & \text{if } x \geq 1 \end{cases}$$

Mean

$$p$$

Variance

$$p(1-p)$$

Skewness

$$\frac{1-2p}{\sqrt{p(1-p)}}$$

Kurtosis

$$\frac{6p^2 - 6p + 1}{p(1-p)}$$

Median

Not defined

Mode

$$\begin{cases} 0 & \text{if } p < \frac{1}{2} \\ 1 & \text{if } p > \frac{1}{2} \\ 0 \text{ and } 1 & \text{if } p = \frac{1}{2} \end{cases}$$

PsiBinomial

`PsiBinomial (n,p,...)`

PsiBinomial (n,p) is a discrete distribution of the number of successes in n independent ‘Bernoulli Trials’ (experiments with exactly two possible outcomes), where p is the success probability in each trial. The Binomial distribution can be used to model the number of winning trades in a trading system, or the number of defective items in a batch.

A random variable X is defined by $X \sim \text{PsiBinomial}(n,p)$ if and only if $n-X \sim \text{PsiBinomial}(n,1-p)$.

The Poisson distribution with parameter λ is a good approximation of the PsiBinomial (n,p) distribution when $n \rightarrow \infty$ and $p \rightarrow 0$, with $\lambda = np$.

Parameters

$n > 0$, integer

$p \in [0,1]$

Range of Function Values

$\{0,1,\dots,n\}$

Probability Mass Function

$$p(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x} & \text{if } x \in \{0, 1, \dots, n\} \\ 0 & \text{otherwise} \end{cases}$$

where $\binom{n}{x}$ is the binomial coefficient,

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ \sum_{i=0}^{\lfloor x \rfloor} \binom{n}{i} p^i (1-p)^{n-i} & \text{if } 0 \leq x \leq n \\ 1 & \text{if } x > n \end{cases}$$

Mean

$$np$$

Variance

$$np(1-p)$$

Skewness

$$\frac{1-2p}{\sqrt{np(1-p)}}$$

Kurtosis

$$\frac{6p^2 - 6p + 1}{np(1-p)}$$

Median

$$\text{one of } \{\lfloor np \rfloor, \lfloor np \rfloor - 1, \lfloor np \rfloor + 1\}$$

Mode

$$\begin{cases} p(n+1) \text{ and } p(n+1)-1 & \text{if } p(n+1) \text{ is integer} \\ \lfloor p(n+1) \rfloor & \text{otherwise} \end{cases}$$

PsiGeometric

`PsiGeometric (p, ...)`

PsiGeometric (p) is a discrete distribution of the number of failures before the first success in a sequence of independent ‘Bernoulli Trials’ (experiments with exactly two possible outcomes), where p

is the success probability in each trial. The Geometric distribution can be used to model the number of losing trades before the first winning trade, the number of items passing inspection before the first defective item appears in a batch, etc.

PsiGeometric (p) can be considered as a discrete analog of the (continuous) Exponential distribution. If $X_1, X_2, \dots, X_n$ are independent geometrically distributed random variables with parameters $p_1, p_2, \dots, p_n$, then $X = \min(X_1, X_2, \dots, X_n)$ is also a Geometrically distributed random variable with parameter $p = 1 - [(1-p_1)(1-p_2)\dots(1-p_n)]$. Additionally, if $X_1, X_2, \dots, X_n$ are Geometrically distributed random variables with parameter p , then their sum is Negative Binomially distributed with parameters n, p .

Parameters

$$p \in [0, 1]$$

Range of Function Values

$$\{0, 1, \dots\}$$

Probability Mass Function

$$p(x) = \begin{cases} p(1-p)^x & \text{if } x \in \{0, 1, \dots\} \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 1 - (1-p)^{\lfloor x \rfloor + 1} & \text{if } x \geq 0 \\ 1 & \text{otherwise} \end{cases}$$

Mean

$$\frac{1-p}{p}$$

Variance

$$\frac{1-p}{p^2}$$

Skewness

$$\frac{2-p}{\sqrt{1-p}}$$

Kurtosis

$$\frac{9p^2 - 17p + 9}{(1-p)^2}$$

Median

$$\frac{\ln(0.5)}{\ln(1-p)} - 1$$

Mode

0

PsiHyperGeo

PsiHyperGeo (n, D, M, ...)

PsiHyperGeo (n,D,M) is a discrete distribution of the number of successes in n successive trials drawn without replacement from a finite population of size M, when it is known that there are exactly D failures in the population. The Hypergeometric distribution can be used to model ‘good’ and defective parts in a manufacturing process.

A Hypergeometric distribution can be approximated by a Binomial distribution with parameters n, p = D/M, when M is very large as compared to n.

Parameters

$$M \in \{0, 1, \dots\}$$

$$n, D \in \{0, 1, \dots, M\}$$

Range of Function Values

$$\{\max(0, n - M + D), \dots, \min(D, n)\}$$

Probability Mass Function

$$p(x) = \frac{\binom{D}{x} \binom{M-D}{n-x}}{\binom{M}{n}}$$

Cumulative Distribution Function

$$F(x) = \sum_{i=1}^x \frac{\binom{D}{i} \binom{M-D}{n-i}}{\binom{M}{n}}$$

Mean

$$\frac{nD}{M}$$

Variance

$$\frac{n \left(\frac{D}{M} \right) (M-n) \left(1 - \frac{D}{M} \right)}{M-1}$$

Skewness

$$\frac{(M-2D)(M-2n)}{M-2} \sqrt{\frac{(M-1)}{nD(M-D)(M-n)}}$$

Kurtosis

$$\left[\frac{M^2(M-1)}{n(M-2)(M-3)(M-n)} \right] \left[\frac{M(M+1)-6M(M-n)}{D(M-D)} + \frac{3n(M-n)(N+6)}{M^2} - 6 \right]$$

Median

Not defined

Mode

$$\frac{(n+1)(D+1)}{M+2} \text{ and } \frac{(n+1)(D+1)}{M+2} - 1 \text{ if } \frac{(n+1)(D+1)}{M+2} \text{ is integral}$$

$$\left\lfloor \frac{(n+1)(D+1)}{M+2} \right\rfloor \text{ otherwise}$$

PsiIntUniform

PsiIntUniform (a,b,...)

PsiIntUniform (a,b) is a discrete distribution with equal probability at each integer value between the lower and upper bounds (a and b). It is used as a rough estimate of the true distribution when the only information we have is that the random variable takes integer values between a and b, and each of these values are equally likely.

Parameters

a,b integers

a < b

Range of Function Values

$$\{a, a+1, \dots, b-1, b\}$$

Probability Mass Function

$$p(x) = \begin{cases} \frac{1}{b-a+1} & \text{if } a \leq x \leq b, \text{ and } x \text{ integer} \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{\lfloor x \rfloor - a + 1}{b - a + 1} & \text{if } a \leq x \leq b \\ 1 & \text{if } x > b \end{cases}$$

Mean

$$\frac{a+b}{2}$$

Variance

$$\frac{(b-a+1)^2-1}{12}$$

Skewness

$$0$$

Kurtosis

$$\frac{6\{(b-a+1)^2+1\}}{5\{(b-a+1)^2-1\}}$$

Median

$$\frac{a+b}{2}$$

Mode

Not defined

PsiLogarithmic

`PsiLogarithmic (p, ...)`

PsiLogarithmic is a discrete distribution with a lower bound of 1. It is used to describe the diversity of a sample.

Parameters

$$p \in (0,1)$$

Range of Function Values

$$\{1,2,3,\dots\}$$

Probability Mass Function

$$p(x) = \begin{cases} \frac{-1}{\ln(1-p)} \frac{p^x}{x} & \text{if } x \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$F(x) = 1 + \frac{B_p(x+1, 0)}{\ln(1-p)}$, $B_p(a, b)$ is the incomplete beta function

$$B_p(a, b) = \int_0^p t^{a-1} (1-t)^{b-1} dt$$

Mean

$$\frac{-p}{(1-p)\ln(1-p)}$$

Variance

$$-p \frac{p + \ln(1-p)}{(1-p)^2 \ln^2(1-p)}$$

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

1

PsiNegBinomial

`PsiNegBinomial (s, p, ...)`

PsiNegBinomial (s,p) is a discrete distribution of the number of trials required to obtain s number of successes in a sequence of independent ‘Bernoulli Trials’, where the success probability in each trial is p. The Negative Binomial distribution can be used to describe the number of items that pass inspection before the sth defective item is found.

If $X_1, X_2, \dots, X_s$ are independent Geometrically distributed random variables each with parameter p, then their sum is Negative Binomially distributed, with parameters s and p. Additionally, the Geometric distribution with parameter p is the same as a Negative Binomial distribution with parameters s = 1 and p; hence, the Geometric distribution is a special case of a Negative Binomial distribution.

Parameters

$$p \in [0, 1]$$

$s > 0$, integer

Range of Function Values

$$\{0, 1, 2, 3, \dots\}$$

Probability Mass Function

$$p(x) = \begin{cases} \binom{s+x-1}{x} p^s (1-p)^x & \text{if } x \in \{0, 1, \dots\} \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} \sum_{i=0}^{\lfloor x \rfloor} \binom{s-i-1}{i} p^s (1-p)^i & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Mean

$$\frac{s(1-p)}{p}$$

Variance

$$\frac{s(1-p)}{p^2}$$

Skewness

$$\frac{2-p}{\sqrt{s(1-p)}}$$

Kurtosis

$$\frac{p^2 - 6p + 1}{s(1-p)} + 3$$

Median

Not defined

Mode

$$\begin{cases} \frac{s(1-p)-1}{p} \text{ and } \frac{s(1-p)-1}{p} + 1 & \text{if } \left(\frac{s(1-p)-1}{p} \right) \text{ is integer} \\ \left\lfloor \frac{s(1-p)-1}{p} \right\rfloor + 1 & \text{otherwise} \end{cases}$$

PsiPoisson

`PsiPoisson` ($\lambda, \dots$)

`PsiPoisson` (λ) is a discrete distribution of the number of events that occur in an interval of time, when the events occur at a known average rate, and each occurrence is independent of the time of occurrence of the previous event.

The Poisson distribution with parameter λ can be approximated by a Normal distribution with mean λ and variance λ , for large values of λ . If $X_1, X_2, \dots, X_n$ are independent Poisson random variables with parameters $\lambda_1, \lambda_2, \dots, \lambda_n$, then their sum is also a Poisson random variable with parameter $\lambda_1 + \lambda_2 + \dots + \lambda_n$.

Parameters

$$\lambda > 0$$

Range of Function Values

$$\{0, 1, \dots\}$$

Probability Mass Function

$$p(x) = \begin{cases} \frac{e^{-\lambda} \lambda^x}{x!} & \text{if } x \in \{0, 1, \dots\} \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ e^{-\lambda} \sum_{i=0}^{\lfloor x \rfloor} \frac{\lambda^i}{i!} & \text{if } x \geq 0 \end{cases}$$

Mean

$$\lambda$$

Variance

$$\lambda$$

Skewness

$$\sqrt{\frac{1}{\lambda}}$$

Kurtosis

$$\frac{1}{\lambda} + 3$$

Median

Not applicable

Mode

$$\begin{cases} \lambda \text{ and } \lambda - 1 & \text{if } \lambda \text{ is integer} \\ \lfloor \lambda \rfloor & \text{otherwise} \end{cases}$$

Custom Distributions

PsiCumul

`PsiCumul (a,b, {x1,x2,...,xn}, {p1,p2,...,pn}, ...)`

`PsiCumul (a,b, {x1,x2,...,xn}, {p1,p2,...,pn},...)` is a custom continuous distribution with lower and upper bounds equal to a and b respectively, and with user specified values $x_1, x_2, \dots, x_n$ and corresponding cumulative probabilities $p_1, p_2, \dots, p_n$.

Parameters

$$a < b$$

$$a \leq x_i \leq b \forall i = 1, 2, \dots, n$$

$$0 \leq p_i \leq 1 \forall i = 1, 2, \dots, n$$

$$p_i < p_{i+1} \forall i = 1, 2, \dots, n-1$$

$$x_i < x_{i+1} \forall i = 1, 2, \dots, n-1$$

Define the boundary parameters as

$$x_0 = a, x_{n+1} = b, p_0 = 0, p_{n+1} = 1$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \frac{p_{i+1} - p_i}{x_{i+1} - x_i} \text{ if } x_i \leq x \leq x_{i+1}$$

Cumulative Distribution Function

$$F(x) = p_i + (p_{i+1} - p_i) \left(\frac{x - x_i}{x_{i+1} - x_i} \right) \text{ if } x_i \leq x \leq x_{i+1}$$

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiCumulD

`PsiCumul (min,max,{x1,x2,...,xn},{pmax,...,pmin},...)`

PsiCumulD is a complimentary distribution to PsiCumul(). PsiCumulD is a cumulative distribution with n points between a minimum and a maximum, user specified values, x₁,x₂,...,x_n, and corresponding cumulative descending probabilities p for each x. See PsiCumul for more information.

PsiDiscrete

`PsiDiscrete ({x1,x2,...,xn},{p1,p2,...,pn},...)`

PsiDiscrete ({x₁,x₂,...,x_n},{p₁,p₂,...,p_n},...) is a custom discrete distribution that takes on values {x₁,x₂,...,x_n} with probabilities {p₁,p₂,...,p_n} respectively.

Parameters

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

$\{p_1, p_2, \dots, p_n\}$

The probabilities p_i are first normalized so that they sum to one

Range of Function Values

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

Probability Density Function

$$f(x) = \begin{cases} p_i & \text{if } x = x_i \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < x_1 \\ \sum_{i=1}^s p_i & \text{if } x_s \leq x < x_{s+1}, s < n \\ 1 & \text{otherwise} \end{cases}$$

This assumes that $x_i \leq x_{i+1} \forall i = 1, 2, \dots, n-1$

Mean

$$\sum_{i=1}^n x_i p_i = \mu$$

Variance

$$\sum_{i=1}^n (\mu - x_i)^2 p_i = \sigma^2$$

Skewness

$$\frac{\sum_{i=1}^n (x_i - \mu)^3 p_i}{\sigma^3}$$

Kurtosis

$$\frac{\sum_{i=1}^n (x_i - \mu)^4 p_i}{\sigma^4}$$

Median

$$x_s \text{ where } s = \min \left(j = 1, 2, \dots, n : \sum_{i=1}^j p_i \geq 0.5 \right)$$

This assumes that $x_i \leq x_{i+1} \forall i = 1, 2, \dots, n-1$

Mode

$$x_{\arg \max_{i=1,2,\dots,n} (p_i)}$$

PsiDisUniform

`PsiDisUniform` ($\{x_1, x_2, \dots, x_n\}, \dots$)

`PsiDisUniform` ($\{x_1, x_2, \dots, x_n\}, \dots$) is a custom discrete distribution that takes on values $\{x_1, x_2, \dots, x_n\}$ with equal probability. It is similar to the `PsiDiscrete` distribution except that no probabilities are specified – instead all x values are equally likely to occur. (In the equations below, each $p_i = 1/n$.) `PsiDisUniform` can be used to *resample* a set of past observations $\{x_1, x_2, \dots, x_n\}$.

Parameters

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

These values have the corresponding probabilities as

$$p_i = \frac{1}{n} \forall i = 1, 2, \dots, n$$

Range of Function Values

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

Probability Density Function

$$f(x) = \begin{cases} p_i & \text{if } x = x_i \\ 0 & \text{otherwise} \end{cases}$$

Cumulative Distribution Function

$$F(x) = \begin{cases} 0 & \text{if } x < x_1 \\ \sum_{i=1}^s p_i & \text{if } x_s \leq x < x_{s+1}, s < n \\ 1 & \text{otherwise} \end{cases}$$

This assumes that $x_i \leq x_{i+1} \forall i = 1, 2, \dots, n-1$

Mean

$$\sum_{i=1}^n x_i p_i = \mu$$

Variance

$$\sum_{i=1}^n (\mu - x_i)^2 p_i = \sigma^2$$

Skewness

$$\frac{\sum_{i=1}^n (x_i - \mu)^3 p_i}{\sigma^3}$$

Kurtosis

$$\frac{\sum_{i=1}^n (x_i - \mu)^4 p_i}{\sigma^4}$$

Median

$$x_s \text{ where } s = \min \left(j = 1, 2, \dots, n : \sum_{i=1}^j p_i \geq 0.5 \right)$$

This assumes that $x_i \leq x_{i+1} \forall i = 1, 2, \dots, n-1$

Mode

$$x_{\arg \max_{i=1,2,\dots,n} (p_i)}$$

PsiGeneral

`PsiGeneral (a,b, {x1,x2,...,xn}, {w1,w2,...,wn}, ...)`

`PsiGeneral (a,b, {x1,x2,...,xn}, {w1,w2,...,wn},...)` is a custom continuous distribution with lower and upper bounds equal to `a` and `b` respectively, and with user specified values `x1,x2,...,xn` and corresponding weights `w1,w2,...,wn`. This is similar to a `PsiCumul (a,b, {x1,x2,...,xn}, {p1,p2,...,pn},...)` distribution, where the probabilities are calculated using the weights as shown below.

Parameters

$$a < b$$

$$a \leq x_i \leq b \forall i = 1, 2, \dots, n$$

$$x_i < x_{i+1} \forall i = 1, 2, \dots, n-1$$

The cumulative probabilities are defined as
$$p_i = \sum_{k=1}^i \frac{w_k}{\sum_{j=1}^n w_j}$$

Range of Function Values

$$[a, b]$$

Probability Density Function

$$f(x) = \frac{p_{i+1} - p_i}{x_{i+1} - x_i} \text{ if } x_i \leq x \leq x_{i+1}$$

Cumulative Distribution Function

$$F(x) = p_i + (p_{i+1} - p_i) \left(\frac{x - x_i}{x_{i+1} - x_i} \right) \text{ if } x_i \leq x \leq x_{i+1}$$

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiHistogram

`PsiHistogram (a, b, {w1, w2, ..., wn} , ...)`

`PsiHistogram (a, b, {w1, w2, ..., wn} , ...)` is a custom continuous distribution with lower and upper bounds equal to a and b respectively, and with user specified weights w₁, w₂, ..., w_n corresponding to n subintervals of equal size. This is similar to a `PsiCumul (a, b, {x1, x2, ..., xn}, {p1, p2, ..., pn} , ...)` distribution, where the probabilities are calculated using the weights as shown below, and the interval defined by the bounds a and b is divided into subintervals of equal size as described below.

Parameters

$a < b$

The interval $[a, b]$ is divided into n subintervals
of equal size $\{x_1, x_2, \dots, x_n\}$

$$x_i = a + i \left(\frac{b - a}{n} \right)$$

The cumulative probabilities are defined as $p_i = \frac{\sum_{k=1}^i w_k}{\sum_{j=1}^n w_j}$

Range of Function Values

$[a, b]$

Probability Density Function

$$f(x) = \frac{p_{i+1} - p_i}{x_{i+1} - x_i} \text{ if } x_i \leq x \leq x_{i+1}$$

Cumulative Distribution Function

$$F(x) = p_i + (p_{i+1} - p_i) \left(\frac{x - x_i}{x_{i+1} - x_i} \right) \text{ if } x_i \leq x \leq x_{i+1}$$

Mean

Not defined

Variance

Not defined

Skewness

Not defined

Kurtosis

Not defined

Median

Not defined

Mode

Not defined

PsiMetalog

`PsiMetalog(min, max, coefficients)`

A general meta-log distribution is defined in models through `PsiMetalog()`. This function has as arguments a vector of coefficients (`coefficients`) and optionally lower (`min`) and upper (`max`) bounds. The metalog distribution family is *designed* to be determined from historical data, without requiring a

distribution fitting process. They are computed from a set of historical data pairs $\{y, x\}$, where y is a cumulative probability and x is the correspondent percentile. The coefficients argument is either a range or an array of 2 to 10 numerical elements.

There are 3 optional cases for this distribution:

- Both *min* and *max* are missing which results in an unbounded distribution

PsiMetalog(, , {33, 3.7, -2.4, 5.7, 10.1})

- Only *min* exists which results in a semi-bounded distribution

PsiMetalog(0, , {33, 3.7, -2.4, 5.7, 10.1})

- Both *min* and *max* exist which results in a bounded distribution

PsiMetalog(0, 55, {33, 3.7, -2.4, 5.7, 10.1})

Note: An exclusive bounding using only the max argument is not supported. The min and max arguments are true lower and upper bounds not to be confused with lower truncate and upper truncate. Optional standard property functions may be used with this distribution, i.e. PsiTruncate, PsiShift, PsiCorrelate, etc. The distribution may also be compounded.

Analytic moments are available only for unbounded distributions. Kurtosis is not computed for more than 5 coefficients.

Parameters

min, max, coefficients

Range of Function Values

$(-\infty, \infty)$

5-Term Quantile Function (Inverse CDF)

$$M_5(y) := \left(a_1 + \log\left[\frac{y}{1-y}\right]a_2 + \left(\frac{-1}{2} + y\right)\log\left[\frac{y}{1-y}\right]a_3 + \left(\frac{-1}{2} + y\right)a_4 + \left(\frac{-1}{2} + y\right)^2a_5\right)$$

See the website, <http://metallogdistributions.com/>, for more information.

2-Term Metalog Moments

$$u'_{1,2} := a_1 \quad \text{mean (first moment) of 2 term metalog}$$

$$u_{2,2} := \frac{1}{3}\pi^2a_2^2 \quad \text{variance (2nd central moment) of 2 term metalog}$$

$$u_{3,2} := 0 \quad \text{skewness (3rd central moment) of 2 term metalog}$$

$$u_{4,2} := \frac{7}{15}\pi^4a_2^4 \quad \text{kurtosis (4th central moment) of 2 term metalog}$$

See the website, <http://metallogdistributions.com/>, for more information.

3-Term Metalog Moments

$$u'_{1,3} := u'_{1,2} + \frac{a_3}{2} \quad \text{mean (first moment) of 3 term metalog}$$

$$u_{2,3} := u_{2,2} + \frac{a_3^2}{12} + \frac{1}{36}\pi^2a_3^2 \quad \text{variance (2nd central moment) of 3 term metalog}$$

$$u_{3,3} := u_{3,2} + \pi^2a_2^2a_3 + \frac{1}{24}\pi^2a_3^3 \quad \text{skewness (3rd central moment) of 3 term metalog}$$

kurtosis (4th central moment) of 3 term metalog

$$u_{4,3} := u_{4,2} + \frac{3}{2}\pi^2a_2^2a_3^2 + \frac{7}{30}\pi^4a_2^2a_3^2 + \frac{a_3^4}{80} + \frac{1}{24}\pi^2a_3^4 + \frac{7\pi^4a_3^4}{1200}$$

See the website, <http://metallogdistributions.com/>, for more information.

PsiMetalogSPT

`PsiMetalogSPT(min, max, quantiles, probability)`

This distribution is constructed by a symmetric-percentile triplet (SPT). The cumulative probabilities are $\{\text{prob}, 0.5, 1-\text{prob}\}$, where $0 < \text{prob} < 0.5$. The corresponding quantile set is a 3-element array/range of quantiles. The optional *min* and *max* arguments have the same meaning of lower and upper bounds as in the PsiMegalog. This distribution can be truncated, shifted, correlated, compounded etc. through the use of property functions.

There are 3 optional cases for this distribution:

- Both *min* and *max* are missing which results in an unbounded distribution

PsiMetalogSPT(, , {35, 50, 75}, 0.1)

- Only *min* exists which results in a semi-bounded distribution

PsiMetalogSPT(0, , {35, 50, 75}, 0.1) semi-bounded

- Both *min* and *max* exist which results in a bounded distribution

PsiMetalogSPT(0, 100, {35, 50, 75}, 0.1)

Note: Analytic moments are available only for unbounded distributions. Kurtosis is always computed.

Parameters

min, max, quantiles, prob

Range of Function Values

$(-\infty, \infty)$

Metalog Quantile Function where $n = 3$

$M^{\text{logit}}(y) = (b_l + b_u e^{M(y)}) / (1 + e^{M(y)})$ for $0 < y < 1$

$= b_l$ for $y = 0$

$= b_u$ for $y = 1$

Metalog PDF

$m^{\text{logit}}(y) = m(y) (1 + e^{M(y)})^2 / ((b_u - b_l) e^{M(y)})$ for $0 < y < 1$
 $= 0$ for $y = 0$ or $y = 1$

where b_l and b_u are user-specified lower and upper bounds respectively and

$$M(y) = a_1 + a_2 \ln \left(\left(\frac{y}{1-y} \right) + a_3 (y - 0.5) \ln \left(\frac{y}{1-y} \right) \right)$$
$$m(y) = \frac{1}{\left[\frac{a_2}{y(1-y)} + a_3 \left(\frac{y-0.5}{y(1-y)} + \ln \left(\frac{y}{1-y} \right) \right) \right]}$$

The constants (a_1, a_2, a_3) can be calculated directly from SPT input parameters:

$$a_1 = \ln(\gamma_{0.5})$$

$$a_2 = \frac{1}{2} \left[\ln \frac{1-\alpha}{\alpha} \right]^{-1} \ln \left[\frac{\gamma_{1-\alpha}}{\gamma_{0.5}^2} \right]$$

$$a_3 = \left[(1 - 2\alpha) \ln \left(\frac{1-\alpha}{\alpha} \right) \right]^{-1} \ln \left(\frac{\gamma_{1-\alpha} \gamma_{\alpha}}{\gamma_{0.5}^2} \right)$$

$$\text{where } y_\alpha = \frac{q_\alpha - b_l}{b_u - q_\alpha}, y_{0.5} = \frac{q_{0.5} - b_l}{b_u - q_{0.5}}, y_{1-\alpha} = \frac{q_{1-\alpha} - b_l}{b_u - q_{1-\alpha}}$$

The feasibility requirement is given by:

$$[b_l + b_u \gamma_\alpha^{1-k_\alpha} \gamma_{1-\alpha}^{k_\alpha}] [1 + \gamma_\alpha^{1-k_\alpha} \gamma_{1-\alpha}^{k_\alpha}]^{-1} < q_{0.5} < [b_l + b_u \gamma_\alpha^{k_\alpha} \gamma_{1-\alpha}^{1-k_\alpha}] [1 + \gamma_\alpha^{k_\alpha} \gamma_{1-\alpha}^{1-k_\alpha}]^{-1}$$

$$\text{where } k_\alpha = \frac{1}{2} \left[1 - 1.66711 \left(\frac{1}{2} - \alpha \right) \right]$$

See the website, <http://metalogdistributions.com/>, for more information.

Stochastic Information Packets (SIPs)

As described by Dr. Savage in *OR/MS Today*, the simplest element of a coherent Stochastic Library is a *Stochastic Information Packet* or SIP, which is just a list of trial values for a single uncertain variable. An example might be 1,000 sample values, in a specific sequence, for the future price of oil at some point or interval of time. In a Monte Carlo simulation, trials will be drawn from the SIP *in the order in which they were generated*.

Uncertain variables may be dependent in ways not measured by traditional correlation. SIPs for several different uncertain variables may be generated as a group, in a manner that preserves the statistical dependence between them. If a Monte Carlo simulation draws trials in the specific order given by the SIPs, this dependence will be reflected in the simulation model results.

In the Solver SDK, one SIP is naturally represented by a column of cell values, and a SIP with n variables is most easily represented by a DoubleMatrix with n number of rows and number of columns equal to the number of simulation trials (*prob.Solver.NumTrials()*). The data for this table of cell values may be easily drawn from a relational database or multidimensional data warehouse, or it may be created and maintained directly in the programming language of your choice.

Creating a SIP/SLURP

Of course, a SIP must first be created before it can be used. In some cases, you may have a data source from which you can draw SIP data directly. For example, if you are working with demographic data such as age, family size, income, and taxes paid, you might be able to use a representative sample directly in multiple SIPs. But in many cases, an expert (perhaps like you) will have to select appropriate analytic probability distributions and their parameters, determine whether and how they should be correlated, and then generate the trial data through a Monte Carlo process.

The Solver SDK is a great tool for *creating* SIPs, as well as using them. It supports a wide range of analytic distributions, shifting and truncation of distributions, and rank order correlation of different distributions. When you run a simulation, the trial data is generated and can easily be saved – this will create a SIP.

Creating a SIP/SLURP from a Data Source

There are two ways to create a SIP or a SLURP in the SDK when providing data directly, in the main body of the program or in the evaluator. Please see the candy store example below which illustrates how to create SIP/SLURPs using both methods.

Imagine that you are the owner of a candy store and you would like to simulate next week's Gross Profit using sales from the previous week, shown in the table below.

Daily Sales	Chocolate Bars	Juju Beans	Chocolate Nuggets	Eclairs
Monday	9	1	3	6
Tuesday	10	2	2	6

Wednesday	6	1	5	6
Thursday	8	1	7	6
Friday	11	1	3	6

Since we have four products, chocolate bars, juju beans, chocolate nuggets and éclairs, we will create a problem with four uncertain variables.

First, we create a simulation Problem with 4 uncertain variables, one for each product, and one uncertain function, gross profit.

```
CProblem prob(Solver_Type_Simulate, 4, 1);
```

The line of code below sets the number of trials to 5, one for each day of the week.

```
prob.Solver.NumTrials = 5;
```

Now we create and populate the DoubleMatrix, *mySipValues*, with four rows and the number of columns equal to *prob.Solver.NumTrials* or 5. Note that we are using NZBgn and NZEnd to speed up the population of the DoubleMatrix *mySipValues*. For more information on NZBgn and NZEnd, please see the section "Passing Sparse Matrices" in the chapter, "Designing Your Application".

```
CDoubleMatrix mySipValues(4, prob.Solver.NumTrials);
mySipValues.NZBgn();
mySipValues.Value[0][0] = 9;
mySipValues.Value[0][1] = 10;
mySipValues.Value[0][2] = 6;
mySipValues.Value[0][3] = 8;
mySipValues.Value[0][4] = 11;
mySipValues.Value[1][0] = 1;
mySipValues.Value[1][1] = 2;
mySipValues.Value[1][2] = 1;
mySipValues.Value[1][3] = 1;
mySipValues.Value[1][4] = 1;
mySipValues.Value[2][0] = 3;
mySipValues.Value[2][1] = 2;
mySipValues.Value[2][2] = 5;
mySipValues.Value[2][3] = 7;
mySipValues.Value[3][4] = 3;
mySipValues.Value[3][0] = 6;
mySipValues.Value[3][1] = 6;
mySipValues.Value[3][2] = 6;
mySipValues.Value[3][3] = 6;
mySipValues.Value[3][4] = 6;
mySipValues.NZEnd();
```

We must call the line of code below to pass the DoubleMatrix *mySipValues* to the *Variables[0]* object.

```
prob.Variables[0].AllTrials = mySipValues;
```

Alternatively, you could also use the `PsiSip` Distribution object to pass these same values as in the example below.

```
prob.Model.VarDistribution[0]=CDistribution(L"PsiSip(9,10,6,8,11)");
prob.Model.VarDistribution[1] = CDistribution(L"PsiSip(1,2,1,1,1)");
prob.Model.VarDistribution[2] = CDistribution(L"PsiSip(3,2,5,7,3)");
prob.Model.VarDistribution[3] = CDistribution(L"PsiSip(6,6,6,6,6)");
```

One more way to create and pass a SIP/SLURP is by creating a new `Distribution`. In the example code below, we first create four arrays (`SipValues1`, `SipValues2`, `SipValues3`, `SipValues4`), create 4 distributions (`dist1`, `dist2`, `dist3`, `dist4`) of type `Distribution_Type_Sip`, set the number of parameters (sampled values) to 5, and then pass the sampled values to the four arrays, `SipValues1`, `SipValues2`, `SipValues3`, and `SipValues4`.

```
double SipValues1[5] = { 9, 10, 6, 8, 11 };
CDistribution dist1;
dist1.DistType = Distribution_Type_Sip;
dist1.NumParams = 5;
dist1.Params = SipValues1;
double SipValues2[5] = { 1, 2, 1, 1, 1 };
CDistribution dist2;
dist2.DistType = Distribution_Type_Sip;
dist2.NumParams = 5;
dist2.Params = SipValues2;
double SipValues3[5] = { 3, 2, 5, 7, 3 };
CDistribution dist3;
dist3.DistType = Distribution_Type_Sip;
dist3.NumParams = 5;
dist3.Params = SipValues3;
double SipValues4[5] = { 6, 6, 6, 6, 6 };
CDistribution dist4;
dist4.DistType = Distribution_Type_Sip;
dist4.NumParams = 5;
dist4.Params = SipValues4;
```

Next we create and pass the evaluator of type `Eval_Type_Sample`, `CSIPEvaluator`, in the example code below. Finally, we ask the SDK to run a simulation by calling `prob.Solver.Simulate()`.

```
CSIPEvaluator evalSip;
prob.Evaluators[Eval_Type_Sample].Evaluator = & evalSip;
prob.Solver.Simulate();
```

The evaluator, `CSIPEvaluator`, below will be called 5 times or the number of trials. Each time the simulator is called, values from the table above are passed to the `pVar` array. For example, the first time that the evaluator is called, $pVar[0] = 9$, $pVar[1] = 1$, $pVar[2] = 3$ and $pVar[3] = 6$. The second time that the evaluator is called, $pVar[0] = 10$, $pVar[1] = 2$, $pVar[2] = 2$ and $pVar[3] = 6$. The fifth and last time that the evaluator is called, $pVar[0] = 11$, $pVar[1] = 1$, $pVar[2] = 3$ and $pVar[3] = 6$.

```

class CSIPEvaluator : public IEvaluator
{
public:
virtual _Engine_Action Evaluate(CEvaluator& evaluator)
{
    CProblem& p = evaluator.Problem;
    double bars, beans, nuggets, eclairs;
    const double* pVar = p.VarUncertain.Value;
    double* pFcn = p.FcnUncertain.Value;
    bars = pVar[0] * (2.5 - 1.5);
    beans = pVar[1] * (1.75 - 1.05);
    nuggets = pVar[2] * (3.5 - 2.25);
    eclairs = pVar[3] * (4 - 2.95);
    pFcn[0] = bars + beans + nuggets + eclairs;
    return Engine_Action_Continue;
}
}

```

Alternatively, it is also possible to supply the SIP/SLURP values directly to the SDK inside of the *Eval_Type_Sample* evaluator. For this example, we create a simulation *Problem*, set the *Solver Type* to *Simulate*, and add one uncertain function, *grossProfit*. Notice that our *Problem* has no variables.

```

CProblem prob;
prob.Solver.SolverType = Solver_Type_Simulate;
CFunction grossProfit(Function_Type_Uncertain, L"grossProfit",1);
prob.Functions.Add(grossProfit);

```

Next we set the number of trials to 5 for each day of the week.

```

prob.Solver.NumTrials = 5;

```

Next we create and pass the evaluator of type *Eval_Type_Sample*, *CSIPEvaluator*, and ask the SDK to run a simulation by calling *prob.Solver.Simulate()*

```

CSIPEvaluator evalSip;
prob.Evaluators[Eval_Type_Sample].Evaluator = & evalSip;
prob.Solver.Simulate();

```

When passing the SIP/SLURP values in the evaluator, you will need to utilize a global counter and global arrays as shown here.

```

int i;
double barsData[] = {9,10,6,8,11};
double beansData[] = {1,2,1,1,1};
double nuggetsData[] = {3,2,5,1,3};
double eclairsData[] = {6,6,6,6,6};

```

The evaluator below, *CSIPEvaluator*, will be called 5 times or the number of trials. Each time the simulator is called, values from the *barsData*, *beansData*, *nuggetsData*, and *eclairsData* global arrays are passed sequentially to the evaluator to compute the uncertain function. Afterwards, the value for

the uncertain function, *pFcn*, is passed to the SDK and the SDK is asked to continue solving (*Engine_Action_Continue*).

```
CProblem& p = evaluator.Problem;
double bars, beans, nuggets, eclairs;
bars = barsData[i] * (2.5 - 1.5);
beans = beansData[i] * (1.75 - 1.05);
nuggets = nuggetsData[i] * (3.5 - 2.25);
eclairs = eclairsData[i] * (4 - 2.95);
pFcn[0] = bars + beans + nuggets + eclairs;
i++;
return Engine_Action_Continue;
```

On each trial, the values in the global arrays (*barsData*, *beansData*, *nuggetsData* and *eclairsData*) are passed to the evaluator. For example, the first time that the evaluator is called, *barsData*[0] = 9, *beansData*[1] = 1, *nuggetsData*[2] = 3 and *eclairsData*[3] = 6. The second time that the evaluator is called, *barsData*[0] = 10, *beansData*[1] = 2, *nuggetsData*[2] = 2 and *eclairsData*[3] = 6. The fifth and last time that the evaluator is called, *barsData*[0] = 11, *beansData*[1] = 1, *nuggetsData*[2] = 3 and *eclairsData*[3] = 6.

Creating a SIP/SLURP from a Psi Distribution

It is also possible to create a SIP/SLURP using values sampled from one of the Psi Distribution functions such as PsiNormal, Psi Beta, etc. To do so we must create a new Distribution object, pass the Distribution type, *Distribution_Type_Sip*, specify the number of trials that you would like to sample, and then pass the values to the Distribution.

To begin, we create a simulation Problem with 4 uncertain variables (one for each product) and one uncertain function, Gross Profit.

```
CProblem prob;
prob.Solver.SolverType = Solver_Type_Simulate;
CVariable distVar(Variable_Type_Uncertain, L"distVar", 4);
prob.Variables.Add(distVar);
CFunction GrossProfit(Function_Type_Uncertain, L"GrossProfit", 1);
prob.Functions.Add(GrossProfit);
```

Next, we create the four different distributions using PsiNormal, PsiBeta, PsiGamma and PsiTriangGen,

```
prob.Model.VarDistribution[0] = CDistribution(L"PsiNormal(0,1)");
prob.Model.VarDistribution[1] = CDistribution(L"PsiBeta(1.5,3)");
prob.Model.VarDistribution[2] = CDistribution(L"PsiGamma(4,1)");
prob.Model.VarDistribution[3] =
    CDistribution(L"PsiTriangGen(1,2,4,0.1,0.9)");
```

and add upper and lower bounds to each using *ControlParams*[1] to specify the lower bounds and *ControlParams*[2] to specify the upper bounds.

```
prob.Model.VarDistribution[0].ControlParams[1] = 0;
prob.Model.VarDistribution[0].ControlParams[2] = 20;
```

```

prob.Model.VarDistribution[1].ControlParams[1] = 0;
prob.Model.VarDistribution[1].ControlParams[2] = 20;
prob.Model.VarDistribution[2].ControlParams[1] = 0;
prob.Model.VarDistribution[2].ControlParams[2] = 20;
prob.Model.VarDistribution[3].ControlParams[1] = 0;
prob.Model.VarDistribution[3].ControlParams[2] = 20;

```

The next few lines of code create four different arrays (*SipValues1*, *SipValues2*, *SipValues3*, *SipValues4*) of size 5 (number of days in the week) to hold the sample values from the four Distributions (PsiNormal, PsiBeta, PsiGamma and PsiTriangGen).

```

double SipValuesBars[5];
double SipValuesBeans[5];
double SipValuesNuggets[5];
double SipValuesEclairs[5];

```

Next we create four new Distributions (*dist1*, *dist2*, *dist3*, *dist4*), set the Distribution type for each Distribution (*Distribution_Type_Sip*), set the number of parameters for each Distribution with *dist.NumParams* (i.e. the number of sampled values, in this example, 5), and then pass the sampled values to the *SipValuesX* arrays.

```

CDistribution dist1 ;
dist1.DistType = Distribution_Type_Sip;
dist1.NumParams = 5;
dist1.Params = SipValuesBars;
CDistribution dist2 ;
dist2.DistType = Distribution_Type_Sip;
dist2.NumParams = 5;
dist2.Params = SipValuesBeans;
CDistribution dist3 ;
dist3.DistType = Distribution_Type_Sip;
dist3.NumParams = 5;
dist3.Params = SipValuesNuggets;
CDistribution dist4 ;
dist4.DistType = Distribution_Type_Sip;
dist4.NumParams = 5;
dist4.Params = SipValuesEclairs;

```

Next we create the evaluator of type *Eval_Type_Sample* and set the number of trials to 5 (5 days in a week).

```

prob.Solver.NumTrials = 5;
CSIPEvaluator evalSip;
prob.Evaluators[Eval_Type_Sample].Evaluator = & evalSip;

```

Now we run a simulation to obtain trial values for each distribution which will be passed to our evaluator to compute the uncertain functions.

```
prob.Solver.Simulate();
```

Our evaluator computes the uncertain function values using the sampled values obtained from the 4 uncertain variables, PsiNormal, PsiBeta, PsiGamma and PsiTriangGen.

```
CProblem& p = evaluator.Problem;

double bars, beans, nuggets, eclairs;

const double* pVar = p.VarUncertain.Value;

double* pFcn = p.FcnUncertain.Value;

bars = pVar[0] * (2.5 - 1.5);
beans = pVar[1] * (1.75 - 1.05);
nuggets = pVar[2] * (3.5 - 2.25);
eclairs = pVar[3] * (4 - 2.95);

pFcn[0] = bars + beans + nuggets + eclairs;

return Engine_Action_Continue;
```

The first time the evaluator is called, the first element of the SipValuesBars (SipValuesBars[0]) array will be passed to pVar[0], the first element of the SipValuesBeans (SipValuesBeans[0]) array will be passed to pVar[1], the first element of the SipValuesNuggets (SipValuesNuggets[0]) array will be passed to pVar[2], and the first element of the SipValuesEclairs (SipValuesEclairs[0]) array will be passed to pVar[3]. The second time the evaluator is called, the second element of the SipValuesBars (SipValuesBars[1]) array will be passed to pVar[0], the second element of the SipValuesBeans (SipValuesBeans[1]) array will be passed to pVar[1], the second element of the SipValuesNuggets (SipValuesNuggets[1]) array will be passed to pVar[2], and the second element of the SipValuesEclairs (SipValuesEclairs[1]) array will be passed to pVar[3]. This continues until the last and fifth element of all the arrays (SipValuesBars[4], SipValuesBeans[4], SipValuesNuggets[4], and SipValuesEclairs[4]) are passed to pVar[0], pVar[1], pVar[2], and pVar[3], respectively.

For examples in additional languages, please click the links: [C#](#), [Procedural API](#), or [COM](#).

Dependence and Correlation

Unless you specify otherwise, Monte Carlo simulation assumes that each of your uncertain variables is *independent* of all the other uncertain variables. When two variables are independent, the value of one variable on a given trial conveys no information about the value of the other variable. When two variables are *dependent*, there is a statistical relationship between them: On average, the value of one variable on a given trial *does* convey information about the value of the other variable on that trial.

Measuring Observed Correlation

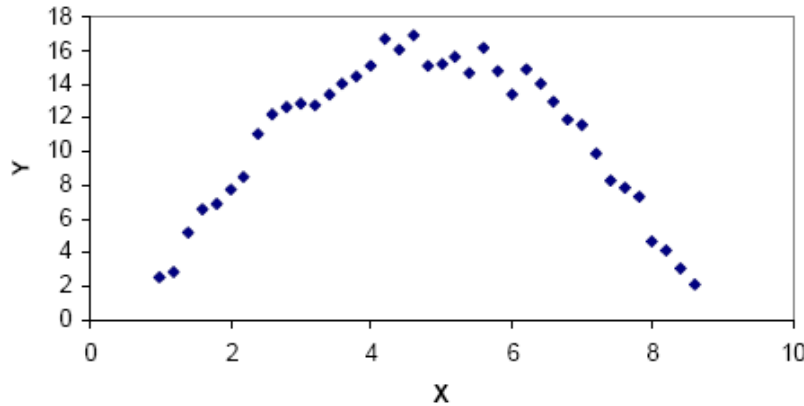
Correlation is a statistical measure of the degree to which one variable is related to another. When we observe that two variables are correlated, it may be that one variable is dependent on the other, or that both variables are dependent on a third variable, *or* that the correlation appeared by chance and there is no real dependence of one variable on the other.

The most common parametric measure of correlation is the **Pearson product moment correlation coefficient**. You can use the **PsiCorrelation** function to compute this correlation coefficient, across the trials of a simulation, for any pair of uncertain functions or variables in your model. The value of a product moment correlation coefficient can range from -1 to +1.

A correlation coefficient captures a simple *linear* relationship between two variables; it cannot capture all of the ways one variable may depend on another. A low or zero correlation coefficient between two uncertain variables or functions does *not* necessarily mean that the variables or functions are

independent – the two variables or functions might have a strong relationship, but one that cannot be captured by a single correlation coefficient.

The figure below is a scatter plot of two variables X and Y. There is an obvious relationship between these two variables – when X is 1, Y is about 2, when X is 5, Y is about 15, when X is 9, Y is about 2, and so on – yet the Pearson product moment correlation coefficient between X and Y is zero.



More complex relationships among variables can be expressed in several ways; one very general way is by creating a SIP (Stochastic Information Packet), as described above under "Stochastic Information Packets (SIPs)".

Inducing Correlation Among Uncertain Variables

If you can compute a value for an uncertain function, Y, with a formula based on an uncertain variable, X, then certainly Y is dependent on X. But in many modeling situations, both Y and X are *uncertain* variables, you have no way to directly compute one from the other, yet Y may *statistically depend* on X. For example, mortgage interest rates depend on bond market interest rates (since mortgages are pooled and sold as securities in the bond market), and both rates depend on inflation expectations, but it is quite difficult to specify a formula that relates these variables.

In cases like these, you'll need to specify the statistical dependence between uncertain variables using a correlation matrix. This is called *inducing correlation* between uncertain variables, that would otherwise be considered independent. **For example code on how to create a correlation matrix in the Solver SDK, please click the appropriate link: [C++](#), [C#](#), [Procedural API](#), or [COM](#).**

The values inside the correlation matrix are **Spearman rank order correlation coefficients**. These are *nonparametric* measures of correlation that are computed from a rank ordering of the trial values drawn for both variables. It can be used to induce correlations between *any* two uncertain variables, whether they have the same or different analytic distributions, or even custom distributions. Like the product moment correlation coefficient, its value can range from -1 to +1.

Meaning of Rank Correlation Coefficients

- A correlation coefficient of +1 forces the sampled values of the uncertain variables to be exactly positively correlated, meaning that the p^{th} percentile value from one distribution will be sampled on the same trial as the p^{th} percentile value from the other distribution. Coefficients from 0 to +1 will produce varying degrees of positive correlation.
- A correlation coefficient of -1 forces the sampled values of the uncertain variables to be exactly negatively correlated, meaning that the p^{th} percentile value from one distribution will be sampled on the same trial as the $(100-p^{\text{th}})$ percentile value from the other distribution. Coefficients from 0 to -1 will produce varying degrees of negative correlation.

- A correlation coefficient of 0 means there is no induced relationship between the variables. In practice, one usually uses coefficients less than +1 or -1, and uses 0 only in a correlation matrix that defines relationships among several variables (see below).

GetCorrelation

```
double *coeff = new double [3];
prob.Variables(x).GetCorrelation(y, coeff)
prob.Functions(x).GetCorrelation(y, coeff)
```

GetCorrelation returns the Pearson product moment correlation coefficient between two uncertain variables or two uncertain functions. Correlation is a measure of linear dependence between two uncertain variables or functions. The correlation coefficient can take on values between -1 and +1. A correlation of -1 indicates a perfect negative correlation (the cells move linearly in opposite directions); a correlation of +1 indicates a perfect positive correlation (the cells move linearly in the same direction). If the two random variables are independent, then their correlation coefficient is zero; but if the correlation coefficient is zero, this does not necessarily mean that the two variables are independent. For more information, see "Dependence and Correlation" above.

Pearson's product moment correlation coefficient between random variables X and Y is defined as:

$$\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X - E[X])(Y - E[Y])]}{\sigma_X \sigma_Y}$$

In Monte Carlo simulation, this value is computed from the sample values $x[]$ and $y[]$ over n trials as:

$$r_{x,y} = \frac{n \sum_{i=1}^n x_i y_i - \left(\sum_{i=1}^n x_i \right) \left(\sum_{i=1}^n y_i \right)}{\sqrt{\left[n \sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i \right)^2 \right] \left[n \sum_{i=1}^n y_i^2 - \left(\sum_{i=1}^n y_i \right)^2 \right]}}$$

When *prob.Variables(x).GetCorrelation(y, coeff)* is executed, the *GetCorrelation* method copies row y (varying from 0 to numvars - 1) of the correlation matrix into the double array *coeff*. In addition, only correlation coefficients of variable block x with respect to the variable y are copied into the double array *coeff*, as well. The size of the double array *coeff* must be the same size as variable block x . Likewise, *prob.Functions(x).GetCorrelation(y, coeff)* behaves in the same manner.

For example, imagine two blocks of variables, the first with 3 variables and the second with 1 variable.

- *prob.Variables[0].GetCorrelation(0, coeff)* will return three correlation coefficients between variable 0 in block 0 and variables 0, 1, 2 in block 0 (the first row of the correlation matrix).
- *prob.Variables[0].GetCorrelation(1, coeff)* will return three correlation coefficients between variable 1 in block 0 and variables 0, 1, 2 in block 0 (the second row of the correlation matrix).
- *prob.Variables[0].GetCorrelation(2, coeff)* will return three correlation coefficients between variable 2 in block 0 and variables 0, 1, 2 in block 0 (the third row of the correlation matrix).
- *prob.Variables[0].GetCorrelation(3, coeff)* will return three correlation coefficients between variable 3 in block 1 and variables 0, 1, 2 in block 0 (the fourth row of the correlation matrix).

For the lines of code above, the double array *coeff* must have **three** elements.

- *prob.Variables[1].GetCorrelation(0, coeff)* will return one correlation coefficient between variable 0 in block 1 and variable 0 in block 0 (the first row of the correlation matrix).

- *prob.Variables [1].GetCorrelation (1, coeff)* will return one correlation coefficient between variable 0 in block 1 and variable 1 in block 0 (the second row of the correlation matrix).
- *prob.Variables [1].GetCorrelation (2, coeff)* will return one correlation coefficient between variable 0 in block 1 and variable 2 in block 0 (the third row of the correlation matrix).
- *prob.Variables [1].GetCorrelation (3, coeff)* will return one correlation coefficient between variable 0 in block 1 and variable 3 in block 1 (the fourth row of the correlation matrix);

For the lines of code above, the double array *coeff* must have **one** element.

For more example code, click the appropriate link: [C#](#), [Procedural API](#), or [COM](#).

GetFrequency

GetFrequency (freq_type, number_of_bins, bin_bounds)

GetFrequency returns a DoubleMatrix of frequencies describing the distribution of trial values for the specified uncertain function. Use *GetFrequency* to easily draw a histogram chart in the programming language of your choice. In C# and COM, the number of bins argument is omitted.

The *freq_type* argument affects the contents of each element of the array result:

Frequency_Type_Density - Each element contains the frequency of trial values falling into the corresponding bin (like a probability density function).

Frequency_Type_Cumulative - Each element contains the cumulative frequency of trial values falling into the corresponding bin plus all lower bins (like a cumulative distribution function).

Frequency_Type_RevCumulative - Each element contains the cumulative frequency of trial values falling into the corresponding bin plus all higher bins (like a reverse cumulative distribution function).

The *bin_bounds* argument is an array of values (e.g. { 5, 10, 15, 20 }) containing the upper limit of trial values that should fall into the corresponding bin. The values should be in strictly increasing order. The number of elements in the array result will be one more than the number of values or cells in the *bin_bounds* argument; the last element contains the number of trial values larger than the highest bin bound value.

In the example code below the double array *binbounds* contains upper limits for seven bins, -150, -100, -50, 0, 50, 100, and 150. A DoubleMatrix with 1 row and 8 columns will be returned (7 bins + 1). Each element in the matrix will hold the number of occurrences for a particular bin depending on the Frequency Type specified in the first argument.

```
double binbounds[] = { -150, -100, -50, 0, 50, 100, 150 };
CDoubleMatrix vargetfreq = prob.VarUncertain.GetFrequency
    (Frequency_Type_Cumulative, 7, binbounds);
CDoubleMatrix fcnggetfreq = prob.FcnUncertain.GetFrequency
    (Frequency_Type_Cumulative, 7, binbounds);
```

For more example code, click the appropriate link: [C#](#), [Procedural API](#), or [COM](#).

PSI Statistics Functions

The SDK offers statistical functions to obtain results for both the uncertain variables and the uncertain functions included in the simulation problem.

CITrials

```
prob.VarUncertain.Statistics.CITrials  
(confidence_level,tolerance)[index];  
  
prob.FcnUncertain.Statistics.CITrials  
(confidence_level,tolerance)[index];
```

CITrials returns the estimated number of simulation trials needed to ensure that the specified uncertain variable's or function's sample mean value (returned by the *Mean()* 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 lies within the confidence interval. To ensure that *N* output cells lie within confidence intervals at level $1 - \alpha$ (e.g. $0.95 = 1 - 0.05$), use a confidence level of α / N .

CoeffVar

```
prob.VarUncertain.Statistics.CoeffVar(percentile)[index];  
prob.FcnUncertain.Statistics.CoeffVar(percentile)[index];
```

CoeffVar finds the coefficient of variation for the specified uncertain function. This function is defined as the ratio of the standard deviation to the mean and is calculated as:

$$cv = \frac{\sigma}{\mu}$$

This statistic measures the magnitude of the variability in relation to the mean of the population.

CondValueAtRisk

```
prob.VarUncertain.Statistics.CondValueAtRisk(percentile)[index];  
prob.FcnUncertain.Statistics.CondValueAtRisk(percentile)[index];
```

CondValueAtRisk returns the conditional Value at Risk for the specified uncertain variable or 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.

Like *ValueAtRisk*, *CondValueAtRisk* returns a loss as a positive number. It is computed as the negative of the mean value of the specified uncertain function for the trials that lie between Minimum and Percentile(1-percentile), inclusive.

ExpGain

```
prob.VarUncertain.Statistics.ExpGain[index];  
prob.FcnUncertain.Statistics.ExpGain[index];
```

ExpGain returns the average of all positive data multiplied by 1 - percentrank of 0 among all data. It is always a positive number.

ExpGainRatio

```
prob.VarUncertain.Statistics.ExpGainRatio[index];  
prob.FcnUncertain.Statistics.ExpGainRatio[index];
```

ExpGain returns the expected gain ratio for a specified uncertain function. This function is calculated as:

$$ExpGainRatio = \frac{ExpGain}{ExpGain + |ExpLoss|}$$

This value ranges between 0 and 1 inclusive.

ExpLoss

```
prob.VarUncertain.Statistics.ExpLoss[index];
prob.FcnUncertain.Statistics.ExpLoss[index];
```

ExpLoss returns the average of all negative data multiplied by the percentrank of 0 among all data. It is always a negative number.

ExpLossRatio

```
prob.VarUncertain.Statistics.ExpLossRatio[index];
prob.FcnUncertain.Statistics.ExpLossRatio[index];
```

ExpLossRatio returns the expected loss ratio for a specified uncertain function. This function is calculated as:

$$ExpLossRatio = \frac{ExpLoss}{ExpGain + |ExpLoss|}$$

This value ranges between 0 and 1 inclusive.

ExpValMargin

```
prob.VarUncertain.Statistics.ExpValMargin[index];
prob.FcnUncertain.Statistics.ExpValMargin[index];
```

ExpValMargin is calculated as:

$$ExpValMargin = ExpGainRatio - ExpLossRatio.$$

This statistic ranges between -1 and 1 inclusive.

Kurtosis

```
prob.VarUncertain.Statistics.Kurtosis[index];
prob.FcnUncertain.Statistics.Kurtosis[index];
```

PsiKurtosis returns the kurtosis for the specified uncertain function *cell*. Kurtosis is the 4th moment and measures the peakedness of the distribution of trial values. It is computed as:

$$kurtosis(X) = \frac{n \sum_{i=1}^n (x_i - \mu)^4}{\left[\sum_{i=1}^n (x_i - \mu)^2 \right]^2}$$

where μ is the mean of the trial values. A higher kurtosis indicates a distribution with a sharper peak and heavier tails, and that more of the variability is due to a small number of extreme outliers or values; a lower kurtosis indicates a distribution with a rounded peak and that more of the variability is due to many modest-sized values.

Maximum

```
prob.VarUncertain.Statistics.Max[index];
```

```
prob.FcnUncertain.Statistics.Max[index];
```

Max returns the maximum value attained by the specified uncertain variable or function over all the trials in the simulation.

Mean

```
prob.VarUncertain.Statistics.Mean[index];
```

```
prob.FcnUncertain.Statistics.Mean[index];
```

Mean returns the mean value for the specified uncertain variable or function. The mean or average value is the 1st moment of the distribution of trials and is computed as:

$$\mu = \frac{\sum_{i=1}^n x_i}{n}$$

The mean is frequently used as a measure of central tendency or the "middle" of an uncertain variable or function; but for skewed distributions, care must be taken in using the mean as a measure of central tendency because the mean is easily distorted by extreme outlier values.

MeanAbsDev

```
prob.VarUncertain.Statistics.MeanAbsDev[index]
```

```
prob.FcnUncertain.Statistics.MeanAbsDev[index]
```

MeanAbsDev returns the average of the absolute deviations of the specified uncertain variables' or function's 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.

MeanCI

```
prob.VarUncertain.Statistics.MeanCI(confidence_level)[index];
```

```
prob.FcnUncertain.Statistics.MeanCI(confidence_level)[index];
```

MeanCI returns the confidence "half-interval" for the estimated mean value (returned by the *Mean()* function) for the specified uncertain function, at the specified *confidence level* (for example 0.95 or 0.99). If μ is the value returned by *Mean()* and δ is the value returned by *MeanCI()*, 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 $\sigma^2(n)$ is the sample variance from n trial values, $\alpha = 1 - \text{confidence\_level}$, 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_i s 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.

Median

```
prob.VarUncertain.Statistics.MeanAbsDev[index]
```

```
prob.FcnUncertain.Statistics.MeanAbsDev[index]
```

`PsiMean` returns the median value for the specified uncertain function *cell*. The median value is the 50th percentile of the distribution of trials and is computed as:

$$\begin{aligned} &X_{(n+1)/2} && \text{if } n \text{ is odd} \\ &\frac{X_{n/2} + X_{(n/2)+1}}{2} && \text{if } n \text{ is even} \end{aligned}$$

The median is a very useful statistic for measuring the center of a distribution.

Minimum

```
prob.VarUncertain.Statistics.Min[index];
```

```
prob.FcnUncertain.Statistics.Min[index];
```

Min returns the minimum value attained by the specified uncertain variable or function over all the trials in the simulation.

Mode

```
prob.VarUncertain.Statistics.Mode[index];
```

```
prob.FcnUncertain.Statistics.Mode[index];
```

Mode returns the mode of the specified uncertain variable or function. For discrete distributions, this is the most frequently occurring value (where the probability mass function has its greatest value). For continuous distributions, `Mode()` returns the *half-sample mode* as defined by D.R. Bickel, a robust estimator that is less sensitive to outliers than most other estimators of location.

NumErrors

```
prob.VarUncertain.Statistics.NumErrors;
```

```
prob.FcnUncertain.Statistics.NumErrors;
```

NumErrors returns the number of error or "failed" trials in the most recent simulation. Trial values resulting in an error will not be included in the computations of the statistics such as mean, mode, etc. ($\text{NumErrors} = \text{NumTrials} - \text{NumValues}$).

NumValues

```
prob.VarUncertain.Statistics.NumValues;
```

```
prob.FcnUncertain.Statistics.NumValues;
```

NumValues returns the number of successful trials in the most recent simulation. ($\text{NumValues} = \text{NumTrials} - \text{NumErrors}$).

Percentile

```
prob.VarUncertain.Percentile(percentile) [index];
```

```
prob.FcnUncertain.Percentile(percentile) [index];
```

Percentile returns a percentile (.01-.99) value for the specified uncertain variable or function. This means that m (or $m\%$) of the simulation trials have values less than the returned value, where m is the percentile.

Range

```
prob.VarUncertain.Statistics.NumValues [index];
```

```
prob.FcnUncertain.Statistics.NumValues [index];
```

Range returns the range of the specified uncertain function *cell*. The range is the difference between the maximum and minimum values attained in the distribution of trial values.

SemiDeviation

```
prob.VarUncertain.Statistics.SemiDeviation(q,target) [index];
```

```
prob.FcnUncertain.Statistics.SemiDeviation(q,target) [index];
```

SemiDeviation returns the semideviation for the specified uncertain variable or function, relative to the *target*, if specified. If the *target* is omitted, the mean value is used. This is a *one-sided* measure of dispersion of values of the uncertain variable or function. The semideviation is the square root of the semivariance, described directly below. If a q argument different from 2 is specified, *SemiDeviation()* returns the q th root of the lower partial moment at power q of the uncertain variable or function.

SemiVariance

```
prob.VarUncertain.Statistics.SemiVariance(q,target) [index];
```

```
prob.FcnUncertain.Statistics.SemiVariance(q,target) [index];
```

SemiVariance returns the semivariance for the specified uncertain variable or function, if the argument q is omitted, or the 'lower partial moment' for the function, if an argument q different from 2 is specified. The semivariance is computed relative to the *target* if specified, or relative to the mean value if *target* is omitted. This is a measure of the dispersion of values of an uncertain variable or 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:

$$\frac{1}{n} \sum_{i=1}^n (t - x_i)_+^q$$

$$(x)_+ = \max(x, 0)$$

t = target value

All trials – not just the trials with downside deviations – are included in n . Again if q is different from 2, the result is called the ‘lower partial moment.’

SigmaCP

```
prob.VarUncertain.Statistics.SigmaCP(lower_limit,
upper_limit)[index];

prob.FcnUncertain.Statistics.SigmaCP(lower_limit,
upper_limit)[index];
```

A Six Sigma index, SigmaCP predicts what the process is capable of producing if the process mean is centered between the lower and upper limits. This index assumes the process output is normally distributed.

$$Cp = \frac{UpperSpecificationLimit - LowerSpecificationLimit}{6\hat{\sigma}}$$

where $\hat{\sigma}$ is the estimated standard deviation of the process.

SigmaCPK

```
prob.VarUncertain.Statistics.SigmaCPK(lower_limit,
upper_limit)[index];

prob.FcnUncertain.Statistics.SigmaCPK(lower_limit,
upper_limit)[index];
```

A Six Sigma index, SigmaCPK predicts what the process is capable of producing if the process mean is not centered between the lower and upper limits. This index assumes the process output is normally distributed and will be negative if the process mean falls outside of the lower and upper specification limits.

$$Cpk = \frac{\min(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{3\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

SigmaCPKLower

```
prob.VarUncertain.Statistics.SigmaCPKLower(lower_limit)[index];

prob.FcnUncertain.Statistics.SigmaCPKLower(lower_limit)[index];
```

A Six Sigma index, SigmaCPKLower calculates the one-sided Process Capability Index based on the lower specification limit. This index assumes the process output is normally distributed.

$$Cp, lower = \frac{\hat{\mu} - LowerSpecificationLimit}{3\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

SigmaCPKUpper

```
prob.VarUncertain.Statistics.SigmaCPKUpper(lower_limit)[index];  
prob.FcnUncertain.Statistics.SigmaCPKUpper(lower_limit)[index];
```

A Six Sigma index, SigmaCPKUpper calculates the one-sided Process Capability Index based on the upper specification limit. This index assumes the process output is normally distributed.

$$Cp, upper = \frac{UpperSpecificationLimit - \hat{\mu}}{3\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

SigmaCPM

```
prob.VarUncertain.Statistics.SigmaCPM(lower_limit, upper_limit,  
target)[index];  
prob.FcnUncertain.Statistics.SigmaCPM(lower_limit, upper_limit,  
target)[index];
```

A Six Sigma index, SigmaCPM calculates the capability of the process around a target value. This index is referred to as the Taguchi Capability Index. This index assumes the process output is normally distributed and is always positive.

$$Cpm = \frac{\hat{Cp}}{\sqrt{1 + (\frac{\hat{\mu} - T}{\hat{\sigma}})^2}}$$

where $\hat{Cp}$ is the process capability (SigmaCP), $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and T is the target process mean.

SigmaDefectPPM

```
prob.VarUncertain.Statistics.SigmaDefectPPM(lower_limit,  
upper_limit)[index];  
prob.FcnUncertain.Statistics.SigmaDefectPPM(lower_limit,  
upper_limit)[index];
```

A Six Sigma index, SigmaDefectPPM calculates the Defective Parts per Million.

$$DPMO = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}}) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}})) * 1000000$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaDefectShiftPPM

```
prob.VarUncertain.Statistics.SigmaDefectShiftPPM(lower_limit,  
upper_limit)[index];  
prob.FcnUncertain.Statistics.SigmaDefectShiftPPM(lower_limit,  
upper_limit)[index];
```

A Six Sigma index, SigmaDefectShiftPPM calculates the Defective Parts per Million with an added shift.

$$DPMOShift = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)) * 1000000$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaDefectShiftPPMLower

```
prob.VarUncertain.Statistics.SigmaDefectShiftPPMLower(lower_limit,
shift) [index];
```

```
prob.FcnUncertain.Statistics.SigmaDefectShiftPPMLower(lower_limit,
shift) [index];
```

A Six Sigma index, SigmaDefectShiftPPMLower calculates the Defective Parts per Million, with a shift, below the lower specification limit.

$$DPMOShift, lower = (\delta^{-1}(\frac{LowerSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$$

where $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaDefectShiftPPMUpper

```
prob.VarUncertain.Statistics.SigmaDefectShiftPPMUpper(upper_limit,
shift) [index];
```

```
prob.FcnUncertain.Statistics.SigmaDefectShiftPPMUpper(upper_limit,
shift) [index];
```

A Six Sigma index, SigmaDefectShiftPPMUpper calculates the Defective Parts per Million, with a shift, above the lower specification limit.

$$DPMOShift, upper = (\delta^{-1}(\frac{UpperSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$$

where $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaK

```
prob.VarUncertain.Statistics.SigmaK(lower_limit,
upper_limit) [index];
```

```
prob.FcnUncertain.Statistics.SigmaK(lower_limit,
upper_limit) [index];
```

A Six Sigma index, SigmaK calculates the Measure of Process Center and is defined as:

$$1 - \frac{2 * MIN(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{UpperSpecificationLimit - LowerSpecificationLimit}$$

where $\hat{\mu}$ is the process mean.

SigmaLowerBound

```
prob.VarUncertain.Statistics.SigmaLowerBound(number_stdev) [index];
```

```
prob.FcnUncertain.Statistics.SigmaLowerBound(number_stdev) [index];
```

A Six Sigma index, SigmaLowerBound calculates the Lower Bound as a specific number of standard deviations below the mean and is defined as, $\hat{\mu} - \hat{\sigma} * \#StandardDeviations$, where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

SigmaProbDefectShift

```
prob.VarUncertain.Statistics.SigmaProbDefectShift(lower_limit,
upper_limit, shift)[index];
```

```
prob.FcnUncertain.Statistics.SigmaProbDefectShift(lower_limit,
upper_limit, shift)[index];
```

A Six Sigma index, SigmaProbDefectShift calculates the Probability of Defect, with a shift, outside of the upper and lower limits. This statistic is defined as:

$$1 - \delta^{-1}\left(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right) + \delta^{-1}\left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaProbDefectShiftLower

```
prob.VarUncertain.Statistics.SigmaProbDefectShiftLower(lower_limit,
shift)[index];
```

```
prob.FcnUncertain.Statistics.SigmaProbDefectShiftLower(lower_limit,
shift)[index];
```

A Six Sigma index, SigmaProbDefectShiftLower calculates the Probability of Defect, with a shift, outside of the lower limit. This statistic is defined as:

$$\delta^{-1}\left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaProbDefectShiftUpper

```
prob.VarUncertain.Statistics.SigmaProbDefectShiftUpper(upper_limit,
shift)[index];
```

```
prob.FcnUncertain.Statistics.SigmaProbDefectShiftUpper(upper_limit,
shift)[index];
```

A Six Sigma index, SigmaProbDefectShiftUpper calculates the Probability of Defect, with a shift, outside of the upper limit. This statistic is defined as:

$$1 - \delta^{-1}\left(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaSigmaLevel

```
prob.VarUncertain.Statistics.SigmaSigmaLevel(lower_limit,
upper_limit, shift)[index];
```

```
prob.FcnUncertain.Statistics.SigmaSigmaLevel(lower_limit,
upper_limit, shift)[index];
```

A Six Sigma index, SigmaSigmaLevel calculates the Process Sigma Level with a shift. This statistic is defined as:

$$1 - \delta^{-1}\left(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right) + \delta\left(\delta^{-1}\left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right) - Shift\right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process δ is the standard normal cumulative distribution function, and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaUpperBound

```
prob.VarUncertain.Statistics.SigmaUpperBound(number_stdev)[index];
```

```
prob.FcnUncertain.Statistics.SigmaUpperBound(number_stdev)[index];
```

A Six Sigma index, SigmaUpperBound calculates the Upper Bound as a specific number of standard deviations above the mean and is defined as:

$$\hat{\mu} - \hat{\sigma} * \#StandardDeviations$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

SigmaYield

```
prob.VarUncertain.Statistics.SigmaYield(lower_limit, upper_limit,
shift)[index];
```

```
prob.FcnUncertain.Statistics.SigmaYield(lower_limit, upper_limit,
shift)[index];
```

A Six Sigma index, SigmaYield calculates the Six Sigma Yield with a shift, or the fraction of the process that is free of defects. This statistic is defined as:

$$\delta^{-1}\left(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right) - \delta^{-1}\left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

SigmaZLower

```
prob.VarUncertain.Statistics.SigmaZLower(lower_limit)[index];
```

```
prob.FcnUncertain.Statistics.SigmaZLower(lower_limit)[index];
```

A Six Sigma index, SigmaZLower calculates the number of standard deviations of the process that the lower limit is below the mean of the process. This statistic is defined as:

$$\frac{\hat{\mu} - LowerSpecificationLimit}{\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

SigmaZMin

```
prob.VarUncertain.Statistics.SigmaZMin(lower_limit,  
upper_limit)[index];
```

```
prob.FcnUncertain.Statistics.SigmaZMin(lower_limit,  
upper_limit)[index];
```

A Six Sigma index, SigmaZLower calculates the minimum of SigmaZLower and SigmaZUpper. This statistic is defined as:

$$\frac{MIN(\hat{\mu} - LowerSpecificationLimit, UpperSpecificationLimit - \hat{\mu})}{\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

SigmaZUpper

```
PsiSigmaZUpper(cell, upper_limit, simulation)  
prob.VarUncertain.Statistics.SigmaZUpper(upper_limit)[index];
```

```
prob.FcnUncertain.Statistics.SigmaZUpper(upper_limit)[index];
```

A Six Sigma index, SigmaZUpper calculates the number of standard deviations of the process that the upper limit is above the mean of the process. This statistic is defined as:

$$\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

Skewness

```
prob.Variables[i].Statistics.Skewness[index];
```

```
prob.Functions[i].Statistics.Skewness[index];
```

Skewness returns the skewness for the specified uncertain variable or function. Skewness is the 3rd moment of an uncertain function, and describes the asymmetry of its distribution. Skewness can be either positive or negative: Positive skewness implies that the distribution is right skewed (longer right tails), and negative skewness implies that the distribution is left skewed (longer left tails). Skewness is computed as:

$$skewness(X) = \frac{\sqrt{n} \sum_{i=1}^n (x_i - \mu)^3}{\left[\sum_{i=1}^n (x_i - \mu)^2 \right]^{3/2}}$$

where μ is the mean of the trial values.

StdDev

```
prob.VarUncertain.Statistics.StdDev[index];
```

```
prob.FcnUncertain.Statistics.StdDev[index];
```

StdDev returns the standard deviation for the specified uncertain variable or function. Standard deviation is a measure of the dispersion of an uncertain function, and accounts for both positive and negative deviations from the mean. The square of standard deviation is the Variance. Standard deviation is defined as:

$$stddev(X) = \sqrt{E[X^2] - (E[X])^2}$$

The sampled population standard deviation is given by

$$stddev(X) = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \mu)^2}$$

where $E[.]$ is the expected value, and μ is the mean of the trial values. As a rough rule, about $\frac{3}{4}$ of the values of any uncertain function are within two standard deviations from the mean. A large standard deviation indicates that most of the trial values are away from the mean, and a small standard deviation indicates that most of the trial values are close to the mean.

StdDevCI

```
prob.VarUncertain.Statistics.StdDevCI (confidence level) [index];
```

```
prob.FcnUncertain.Statistics.StdDevCI (confidence level) [index];
```

StdDevCI returns the confidence ‘half-interval’ for the estimated standard deviation of the simulation trials (returned by the *StdDev()* function) for the specified uncertain variable or function, at *confidence level* (for example 0.95 or 0.99). If σ is the value returned by *StdDev()* and δ is the value returned by *StdDevCI()*, the true mean is estimated to lie within the interval $\sigma - \delta$ to $\sigma + \delta$.

If $\sigma^2(n)$ is the sample variance from n trial values, $\alpha = 1 - \text{confidence level}$, 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:

$$t_{n-1, 1-\alpha/2} \sigma(n) \sqrt{\frac{k-1}{4(n-1)}}$$

See also the description of the *MeanCI()* function.

Target

```
prob.VarUncertain.Statistics.PsiTarget (target value) [index];
```

```
prob.FcnUncertain.Statistics.PsiTarget (target value) [index];
```

Target returns the cumulative frequency of the target value in the distribution of trial values for the specified uncertain variable or function. This function returns the proportion of simulated values that are less than or equal to *target value*.

ValueAtRisk

```
prob.VarUncertain.Statistics.ValueAtRisk (percentile) [index];
```

```
prob.FcnUncertain.Statistics.ValueAtRisk (percentile) [index];
```

ValueAtRisk returns the Value at Risk for the specified uncertain variable or 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. So we compute *ValueAtRisk (percentile)* as *–Percentile(1 - percentile)*.

For example, *ValueAtRisk(0.95)[0] = –PsiPercentile(0.05)[0]*.

Variance

```
prob.VarUncertain.Statistics.Variance[variableindex];
```

```
prob.FcnUncertain.Statistics.Variance[variableindex];
```

Variance returns the variance for the specified uncertain variable or function. Like standard deviation, variance is a measure of the spread or dispersion of the distribution of trial values for the specified uncertain variable or function, and takes into account both positive and negative deviations from the mean. The square root of variance is the standard deviation. The variance is the 2nd moment of the distribution of trials and is computed as:

$$\text{var}(X) = E[X^2] - (E[X])^2$$

The sampled population variance is given by

$$\text{var}(X) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \mu)^2$$

Distribution Control Parameters

Control Parameters modify the behavior of the PSI Distribution function to which they are assigned. There are five such controlparams: shift, lower cut, upper cut, seed, lower censor and upper censor. It is possible to both **get** and **set** each control parameter.

ControlParams[0]	Shift	Used to shift the domain of the uncertain variable's distribution by the specified amount. A positive number will shift the distribution to the right while a negative number will shift the distribution to the left.
ControlParams[1]	Lower Cutoff	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.
ControlParams[2]	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, then another sample is taken.
ControlParams[3]	Seed	Used to set a random number seed for Monte Carlo samples generated for a distribution function, that will override any general seed value specified for the simulation model. It is most often used in an analytic distribution that is being sampled for a SIP.
ControlParams[4]	Lower Censor	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. The Lower Censor can also be set by using the Parameter, <i>Lower Censor</i> , as described in the following chapter, Solver SDK Parameters .
ControlParams[5]	Upper Censor	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. The Upper Censor can also be set by using the Parameter, <i>Upper Censor</i> , as described in the following chapter, Solver SDK Parameters .
ControlParams[6]	Base Case	Used to restrict the value of the uncertain variable to the value specified.

For example, `prob.Model.VarDistribution[0] = CDistribution(L"PsiNormal(0,1)")` specifies a Normal distribution with mean 0 and standard deviation 1: Sample values drawn from this distribution *could* be any number from ‘minus infinity’ to ‘plus infinity’ (though sample values near 0 are more *likely* to be drawn). If you want to truncate the distribution so that sample values always lie within the range from -10 to 10, then you would need to use `ControlParam[1]` for the lower bound and `ControlParam[2]` for the upper bound. (See below.)

`prob.Model.VarDistribution[0].ControlParams[1] = -10`
`prob.Model.VarDistribution[0].ControlParams[2] = 10.`

SDK Simulation Parameters

The following parameters allow users to set global default bounds on the probability distributions of all uncertain variables or to run a specific simulation when multiple simulations are being performed.

CensorType

Name	CensorType
Default	0
Min	0
Max	2
Type	Integer

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 Cutoff options. 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 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.

When you use Censor bounds, random samples from the distribution that lie above the upper bound are set equal to the upper bound, and samples that lie below the lower bound are set equal to the lower bound; this causes a “buildup of probability mass” at the bounds – which is appropriate in some situations, but not in others.

Cutoff Type

Name	CutoffType
Default	0
Min	0

Max	2
Type	Integer

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.

When you use Cutoff bounds, random samples from the distribution are effectively rescaled to lie within the lower and upper bounds.

Run Specific Simulation

Name	RunSpecificSim
Default	-1 – Off
Min	-1
Max	2,147,483,647
Type	Integer

When performing multiple simulations, use this parameter to run an individual simulation (i.e. 3rd simulation of 10). A value of "-1" turns this parameter off.

Accessing Statistics from Different Simulations

If you are performing multiple simulations at a time, you can access the simulation of your choice by using `prob.Solver.SimulationIndex`. For example, if you are performing three simulations at a time, you could use the sample code below to find the mean of each uncertain variable (in the first block of uncertain variables) for each simulation:

```
int i;
int j;
for (i = 0; i < prob.Solver.NumSimulations; i++)
{
    prob.Solver.SimulationIndex = i;
    for (j = 0; j < nvars; j++)
    {
        cout << " Mean      = " << prob.Variables[0].Statistics.Mean[i]
              << endl;
    }
}
```

Distribution Fitting

The results of a Monte Carlo simulation are a *sample* of possible values for an uncertain function. Solver SDK can attempt to fit an analytic distribution and parameters to this sample of values. Since the uncertain function may depend on *several* uncertain variables, and this dependence can take arbitrary forms from simple arithmetic to complex functions and table lookups, you may or may not be able to fit an analytic distribution to an uncertain function. If you *can*, this may give you insight into the behavior of the function you are modeling.

To fit a distribution to the sample values, we must first pass the sample values in an array. In this example, the *sample* array contains fifty 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, 1.4, -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}
```

We then declare a new Distribution, `dist`.

```
CDistribution dist;
```

Then pass the *sample* array to the new *dist* Distribution via the *InitSample()* property.

```
dist.InitSample(50, sample);
```

Finally, we call `dist.AutoFit` with three arguments, *Type* (True or False), *Level_of_Significance* (value from 0 to 1), and the optional argument, *FitAll* (True or False).

```
dist.AutoFit(True, 0.95, True);
```

AutoFit

```
dist.AutoFit(True, 0.95, True);
```

For the *Type*, pass **True** if the uncertain variable's values are highly divisible – such as most prices, volumes, interest rates, exchange rates, weights, and distances – or **False** if the underlying physical process involves discrete, countable entities.

The second argument, *Level_of_Significance*, is a value between .9 and .99 which is used in the statistical goodness-of-fit test to determine how well the distribution fits the data. A value of .9 would result in a more conservative fitting (i.e. less accurate).

The third argument, *FitAll*, is optional. When *FitAll* is set to **True**, the default, the SDK internally tries to fit all continuous or discrete distributions (depending on the *Type* 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**.

This function will reject any distribution where *FitTest* > *FitLimit*. See below for explanations of both statistics.

FitSuccess

The property *dist.FitSuccess()* will return **True** if a distribution within the level of significance is found and **False** if not. Afterwards, you will be able to get the name (using *dist.Name*) and the parameter values (using *dist.get_NumParams()* for the number of parameters and *dist.params()* for the values of the parameters) of the distribution found to be the closest fit for the values in the *sample* array.

```
if (dist.FitSuccess)
{
    cout << "Fit succeeded" << endl();
    cout << "Distribution Type = " << dist.Name << endl();
    for (int i=0; i< dist.get_NumParams();i++)
        cout << "Param[" << i << "] = " << dist.Params[i] << endl();
}
```

AutoFitAll

Alternatively, we could also call *dist.AutoFitAll* instead of *Dist.AutoFit*. *AutoFitAll* returns a collection of 20 of the best fitting distributions. *AutoFitAll* takes three arguments, *Type* (**True** or **False**), *SortType* (0, 1 or 2), and *Allow_Shifted_Distributions* (**True** or **False**).

As in *FitAll*, pass **True** for the *Type* argument if the uncertain variable's values are highly divisible – such as most prices, volumes, interest rates, exchange rates, weights, and distances – or **False** if the underlying physical process involves discrete, countable entities.

SortType specifies the test type for Goodness of Fit. For Continuous data, you can choose to rank the fitted distributions by the **Chi-Square**, **Kolmogorov-Smirnov**, or **Anderson-Darling** statistic. For Discrete data, only the Chi-Square statistic is meaningful. Pass *DistFit_SortCS* for Chi Square, *DistFit_SortKS* for Kolmogorov – Smirnov (the default), and *DistFit_SortAD* for Anderson – Darling. (The Level of Significance for *AutoFitAll* is .9.)

The argument *Allow Shifted Distributions* allows the Solver SDK to shift the center of analytic distribution to better fit the sample data; sometimes this is not desirable. Pass **True** to allow the shift and **False** to disable the shift.

Twenty – one distributions are listed in order starting with the distribution with the lowest *FitTest*.

Fit

To fit a specific distribution to your data, for example a Normal or Beta distribution, call *dist.Fit()* with three arguments, *Type* (DistFit_Type enum), *Level_of_Significance* (value from 0 to 1), and the optional argument, *Shift* (True or False). If the specified distribution can not be fit to the data, the Fit method will throw an exception.

```
dist.Fit(DistFit_Beta, 0.95, True);
```

Pass a DistFit enum for the *Type* argument. For a complete list of distributions that can be fit, see the enum *DistFit_Type*.

The second argument, *Level_of_Significance*, is a value between .9 and .99 which is used in the statistical goodness-of-fit test to determine how well the distribution fits the data. A value of .9 would result in a more conservative fitting (i.e. less accurate).

The optional third argument *Allow Shifted Distributions* allows the Solver SDK to shift the center of analytic distribution to better fit the sample data; sometimes this is not desirable. Pass **True** to allow the shift and **False** to disable the shift.

IndepTest

In order to have a reliable fit, the sample to which we fit the distributions should have independent or uncorrelated data. When *IndepTest* returns a small number, this means that the sample is suitable to be used for fitting. The book, "Simulation Modeling & Analysis" states that the data can be considered independent if *IndepTest* is close to 0 while giving an example of dependence as a value of .8. As a result, it is not advised to fit any data with an *IndepTest* value > .5.

```
double test;  
test = dist.IndepTest;
```

FitStatistic

This value indicates how well the sample fits to the given distribution. The smaller the number, the better the fit.

```
double test;  
test = dist.FitStatistic;
```

FitTest

The value returned by *FitTest* is similar to the value returned by *FitStatistic*, but it has been corrected for the size of the sample. This correction yields a more conservative value. This value is used when determining the *FitLimit*. If calling *AutoFit*, the SDK will automatically reject a distribution when *FitTest* > *FitLimit*.

```
double test;  
test = dist.FitTest;
```

FitLimit

If *FitTest* > *FitLimit* for a given distribution, it is assumed that the data is not coming from said distribution.

```
double test;
```

```
test = dist.FitLimit;
```

Risk Solver Engine Options

The following options are used by Risk Solver Engine when running a simulation or solving stochastic optimization, or simulation optimization models. To set a parameter for the Risk Solver Engine, use the syntax below.

C:	<code>SolverEngParamSet(prob, L"Param_Name", <value>);</code>
C#:	<code>problem.Engine.Params["Param_Name"].Value = <value>;</code>
C++:	<code>problem.Engine.Params[L" Param_Name"].Value =<value> ;</code>
JAVA:	<code>problem.Engine().Params().Item("Param_Name").Value(<value>;</code>
MATLAB Procedural:	<code>SolverSDK('EngineParamSet', prob, 'Param_Name', <value>;</code>
MATLAB OOP:	<code>problem.Engine.Params('Param_Name').Value = <value>;</code>
PYTHON:	<code>problem.Engine.Params["Param_Name"].Value = <value></code>
R:	<code>par <- problem\$Engine\$Params["Param_Name"] par\$Value <- <value></code>
VB.NET:	<code>prob.Engine.Params("Param_Name").Value = <value></code>

Random Seed

Parameter Name: "RandomSeed", Value: integer value > 0

Setting the random number seed to a **nonzero value** (any number of your choice is OK) ensures that the *same* sequence of random numbers is used for each simulation. When the seed is zero, the random number generator is initialized from the system clock, so the sequence of random numbers will be different in each simulation. If you need the results from one simulation to another to be strictly comparable, you should set the seed.

You can specify a random seed for each uncertain variable if you wish, ControlParams[3] for the uncertain variable. The seed value you set for Risk Solver Engine affects only uncertain variables that do *not* use ControlParams[3].

Example Code for setting the Random Seed of an uncertain variable

```
//specify the distributions of 2 uncertain variables  
myprob.Model.VarDistribution[0] = new Distribution("PsiNormal(2000,500)");  
myprob.Model.VarDistribution[1] = new Distribution("PsiUniform(200,300)");  
//set the Random Seed using ControlParams[3] for the 1st distribution  
myprob.Model.VarDistribution[0].ControlParams[3] = 1234;  
//set the Random Seed using ControlParams[3] for the 2nd distribution  
myprob.Model.VarDistribution[1].ControlParams[3] = 5678;
```

Random Number Generation and Sampling

On each Monte Carlo trial, *sample values* are drawn from the probability distributions represented by the Distribution functions in your model. Sample values are computed by first drawing a “random number” between 0 and 1, then transforming this uniform random sample value into a sample value that:

- Constrains the samples drawn to obtain better coverage of the sample space, where each Distribution function is a ‘dimension’ of that space
- Ensures that the frequency distribution of samples drawn properly reflects the shape and parameters of the Distribution function
- Ensures that the samples drawn for multiple Distribution functions properly reflect the correlation of distributions with each other

Sampling Method

Parameter Name: "SamplingMethod", Value: 1, 2, or 3

The sampling method: 1 = standard Monte Carlo (default), 2 = Latin Hypercube, 3 = Sobol numbers.

In standard Monte Carlo sampling, numbers generated by the chosen random number generator are used directly to obtain sample values for the uncertain functions in the model. With this method, the variance or estimation error in computed samples for uncertain functions is inversely proportional to the square root of the number of trials; hence to cut the error in half, four times as many trials are required.

Solver SDK provides two other sampling methods than can significantly improve the ‘coverage’ of the sample space, and thus reduce the variance in computed samples for output functions. This means that you can achieve a given level of accuracy (low variance or error) with fewer trials.

Latin Hypercube Sampling. Latin Hypercube sampling begins with a stratified sample in each dimension (one for each uncertain variable), which constrains the random numbers drawn to lie in a set of subintervals from 0 to 1. Then these one-dimensional samples are combined and randomly permuted so that they ‘cover’ a unit hypercube in a stratified manner. This often reduces the variance of uncertain functions.

Sobol numbers (Randomized QMC). Sobol numbers are an example of so-called “Quasi Monte Carlo” or “low-discrepancy numbers,” which are generated with a goal of coverage of the sample space rather than “randomness” and statistical independence. Analytic Solver adds a “random shift” to Sobol numbers, which improves their statistical independence. Sobol numbers are frequently used in quantitative finance applications, where they are often effective at reducing variance.

Random Number Streams

Parameter Name: "RandomStreams", Value: 0 or 1

1 = Generate independent streams of random numbers for each distribution function.
0 (default) = Use a single stream for all distribution functions

Most Monte Carlo simulation tools generate a single sequence of random numbers, taking values consecutively from this sequence to obtain samples for each of the distributions in a model. This introduces a subtle dependence between the samples for all distributions in one trial. In many applications, the effect is too small to make a difference – but in some cases, found in financial engineering and other demanding applications, better results are obtained if independent random number sequences (streams) are used for each distribution in the model. Solver SDK offers this capability for Monte Carlo sampling and Latin Hypercube sampling; it does not apply to Sobol numbers.

If specify a Random Seed for an uncertain variable, the uncertain variable defined by that distribution always has an independent stream of random numbers, regardless of the setting of this option.

Random Number Generator

Parameter Name: "RandomGenerator", Value: 0, 1, 2 or 3

0 for Park-Miller with safeguards, 1 for CMRG generator (default), 2 for WELL generator, 3 for Mersenne Twister.

In common applications, any good random number generator is sufficient – but for challenging applications (for example in financial engineering) that involve many uncertain variables and many thousands of trials, the advanced features of Solver SDK can make a real difference.

Computer-generated numbers are never truly “random,” since they are always computed by an algorithm – they are called *pseudorandom* numbers. A random number generator is designed to quickly generate sequences of numbers that are as close to statistically independent as possible. Eventually, an algorithm will generate the same number seen sometime earlier in the sequence, and at this point the sequence will begin to repeat. The *period* of the random number generator is the number of values it can generate before repeating.

A long period is desirable, but there is a tradeoff between the length of the period and the degree of statistical independence achieved within the period. Hence the Risk Solver Engine, bundled within Solver SDK, offers a choice of four random number generators:

- **Park-Miller** “Minimal” Generator with Bayes-Durham shuffle and safeguards. This generator has a period of $2^{31}-2$. Its properties are good, but the following choices are usually better.
- Combined Multiple Recursive Generator (**CMRG**) of L’Ecuyer. This generator has a period of 2^{191} , and excellent statistical independence of samples within the period.
- Well Equidistributed Long-period Linear (**WELL1024**) generator of Panneton, L’Ecuyer and Matsumoto. This very new generator combines a long period of 2^{1024} with very good statistical independence.
- **Mersenne Twister** generator of Matsumoto and Nishimura. This generator has the longest period of $2^{19937}-1$, but the samples are not as “equidistributed” as for the WELL1024 and CMRG generators.
- **HDR (4)** Random Number Generator, designed by Doug Hubbard. Permits simulations running on various computer platforms to generate identical or independent streams of random numbers.

CLT Threshold

Parameter Name: "CLTThreshold", Value: 1 <= integer value <= 1000

Solver SDK includes the ability to sum multiple independent random variables using compound distributions. A compound distribution generates values for the sum of N independent identically distributed uncertain variables. A distribution is made compound through the use of the PsiCompound() property.

When calculating a compound distribution, SolverSDK first tries to compute the distribution analytically. For example "`=PsiExponential(par, PsiCompound(N))`" can be computed as `PsiGamma(N, par)`. If Analytic Solver is unable to compute a compound distribution analytically, but the frequency of the severity function (N) is greater than the value for the CLT Threshold option, then the distribution will be

computed according to the Central Limit Theorem as PsiNormal(m, s). (The parameters m and s will be computed analytically from the corresponding analytical moments of the severity distribution.) Otherwise, the compound distribution will be computed using Monte Carlo simulation to sum up N independent variates of the severity distribution. The maximum value allowed for this option is 1000 while the minimum value allowed is 1. The default setting is 100.

For more information on compound distributions, see the *Examples: Simulation and Risk Analysis* chapter in the *Analytic Solver User Guide*.

Psi Functions for Analytic Solver

Analytic Solver for Desktop and Cloud Excel offer multiple Psi function calls for optimization, distributions, statistics, properties, etc. All Psi function calls may be used in a Solver SDK model. For documentation of all Psi functions calls, see the Analytic Solver Reference Guide.

Unsupported Excel Functions

Most Excel arithmetic or intrinsic operations, as well as almost all Excel built-in functions may be used in conjunction with Solver SDK Platform. However, there are a few Excel functions that are not supported by the SDK. These functions include: Cell(), Call(), Dollar(), Encodeurl(), EuroConvert(), FieldValue(), FilterXML(), Fixed(), GetPivotDataID(), Hyperlink(), Info(), Register.ID(), Replace(), Roman(), RTD(), Search(), SQL.Request(), WebService() and all Cube functions (CubeKPIMember(), CubeMember(), CubeMemberProperty(), CubeRankedMember(), CubeSet(), CubeSetCount() and CubeValue()).

Solver Result Messages

Introduction

When either *prob.Solver.Optimize* or *prob.Solver.Simulate* is executed, a Solver Result status is always returned. Please see below for an explanation of each status code.

Plug-in Solver engines return the same codes and display the same messages as the Solver SDK engines whenever possible, but they can also return custom result codes (starting with 1000) and display custom messages, as described in their individual documentation.

Simulation Status Result Codes

-10. Simulate_Status_NotExecuted

Simulation was not executed.

-1. Simulate_Status_Lic_Problem

This result is returned if the Solver SDK cannot find its licensing information, if the licensing information is invalid, or if you have a time-limited evaluation license that has expired. Please call Frontline Systems at (775) 831-0300, or send email to us at Support@solver.com for further assistance.

0. Simulate_Status_Optimal

A result code of 0 indicates that the simulation has finished successfully.

6. Simulate_Status_User_Abort

This result is only returned if you stop the SDK during the solution process, such as returning *Engine_Action_Abort* in your evaluator.

11. Simulate_Status_Memory_Dearth

This result is returned when the Solver is unable to allocate the memory it needs to solve the problem. However, since Microsoft Windows supports a "virtual memory" much larger than your available RAM by swapping data to your hard disk, before you see this message you are likely to notice that solution times have greatly slowed down, and the hard disk activity light in your PC is flickering during the solution process. You can save some memory by closing any Windows applications not being used by your program and closing programs that run in the System Tray.

12. Simulate_Status_Interpret_Err

This result is rarely returned. However, when it is returned, it indicates an interpretation error has occurred.

999. Simulate_Status_Exception

This result is returned when an exception has been caught by a try/catch system.

Optimization Status Result codes

-10. Optimize_Status_NotExecuted

Optimization was not executed.

-1. Optimize_Status_Lic_Problem

This result is returned if the Solver SDK cannot find its licensing information, if the licensing information is invalid, or if you have a time-limited evaluation license that has expired. Please call Frontline Systems at (775) 831-0300, or send email to us at support@solver.com for further assistance.

0. Optimize_Status_Optimal

This means that the Solver has found the optimal or "best" solution under the circumstances. The exact meaning depends on whether you are solving a linear or quadratic, smooth nonlinear, global optimization, or integer programming problem, as outlined below. Solvers for non-smooth problems rarely if ever display this result, because they have no way of testing the solution for true optimality.

If you are solving a *linear* programming problem with the LP Simplex Solver (included in the Solver SDK Pro), the Solver has found the *globally* optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective. It is possible that there are other solutions with the *same* objective value, but all such solutions are linear combinations of the current decision variable values.

If you are solving a *linear* programming problem or a *convex quadratic* programming problem with the LP/Quadratic Solver (included in the Solver SDK Platform), the Solver has found the *globally* optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective. It is possible that there are other solutions with the *same* objective value, but all such solutions are linear combinations of the current decision variable values.

If you are solving a *linear* (LP), *convex quadratic* (QP) or *quadratically constrained* (QCP), or *second order cone programming* (SOCP) problem with the SOCP Barrier Solver (included in the Solver SDK Platform), the Solver has found the *globally* optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective. It's possible that there are other solutions with the *same* objective value, but all such solutions are linear combinations of the current decision variable values.

If you are solving a *smooth nonlinear* optimization problem with no integer constraints, the GRG Solver has found a *locally* optimal solution: There is no other set of values for the decision variables *close to the current values* and satisfying the constraints that yields a better value for the objective. In general, there *may* be other sets of values for the variables, far away from the current values, which yield better values for the objective and still satisfy the constraints.

If you are solving a *mixed-integer* programming problem (any problem with integer constraints) using a Solver SDK product, this result means that the Branch & Bound method has found a solution satisfying the constraints (including the integer constraints) with the "best possible" objective value. If the problem is linear or quadratic, the true integer optimal solution has been found. If the problem is smooth nonlinear, the Branch & Bound process has found the best of the locally optimal solutions found for subproblems by the nonlinear Solver.

1. Optimize_Status_Converged

This means that the Solver has found a series of "best" solutions that satisfy the constraints, and that have very similar objective function values; however, no single solution strictly satisfies the Solver's test for optimality. The exact meaning depends on whether you are solving a smooth nonlinear problem with the GRG Solver or a non-smooth problem with the Evolutionary Solver.

When the GRG Solver is being used, this result means that the objective function value is changing very slowly as the Solver progresses from point to point. More precisely, the Solver stops if the absolute value of the relative (i.e. percentage) change in the objective function, in the last few iterations, is less than the Convergence tolerance. A *poorly scaled* model is more likely to trigger this stopping condition, even if the Use Automatic Scaling option is set to True. If you are sure that your model is well scaled, you should consider why it is that the objective function is changing so slowly. For more information, see the discussion of "GRG Solver Stopping Conditions" below.

When the Evolutionary Solver is being used, this result means that the "fitness" of members of the current population of candidate solutions is changing very slowly. More precisely, the Evolutionary Solver stops if 99% or more of the members of the population have "fitness" values whose relative (i.e. percentage) difference is less than the Convergence tolerance setting. The "fitness" values incorporate both the objective function and a penalty for infeasibility, but since the Solver has found some feasible solutions, this test is heavily weighted towards the objective function values. If you believe that the Solver is stopping prematurely when this test is satisfied, you can make the Convergence tolerance smaller, but you may also want to increase the Mutation Rate and/or the Population Size, in order to increase the diversity of the population of trial solutions. For more information, see the discussion of "Evolutionary Solver Stopping Conditions" below.

2. Optimize_Status_No_Remedies

This means that the Solver has found solutions that satisfy the constraints, but it has been unable to further improve the objective, even though the tests for optimality and convergence have not yet been satisfied. The exact meaning depends on whether you are solving a smooth nonlinear problem with the GRG Solver or a non-smooth problem with the Evolutionary Solver.

When the GRG Solver is being used, this status is very rarely returned. It means that the model is *degenerate* and the Solver is probably *cycling*. One possibility worth checking is that some of your constraints are redundant, and should be removed. For more information, see the discussion of "GRG Solver Stopping Conditions" below.

When the Evolutionary Solver is being used, this result is much more common. It means that the Solver has been unable to find a new, better member of the population whose "fitness" represents a relative (percentage) improvement over the current best member's fitness of more than the Tolerance value on the Limit Options dialog tab, in the amount of time specified by the Max Time without Improvement option in the same dialog. Since the Evolutionary Solver has no way of testing for optimality, it will normally stop with either *Optimize_Status_Optimal* or *Optimize_Status_Converged* if you let it run for long enough. If you believe that this result is appearing prematurely, you can either make the Tolerance value smaller (or even zero), or increase the amount of time allowed by the Max Time without Improvement option. For more information, see the discussion of "Evolutionary Solver Stopping Conditions" below.

3. Optimize_Status_Iterate_Limit

This message is returned when the Solver has completed the maximum number of iterations, or trial solutions, allowed by the Iterations parameter setting. You may increase the value for the Iterations parameter setting, but you should also consider whether re-scaling your model or adding constraints might reduce the total number of iterations required.

If you are solving a *mixed-integer* programming problem (any problem with integer constraints), this result is relatively unlikely to be returned. The Evolutionary Solver uses the Max Subproblems and Max Feasible Solutions parameters, and the Branch & Bound method (employed by the other Solver engines on problems with integer constraints) uses the Max Subproblems and Max Integer Solutions parameters, to control the overall solution process. The count of iterations against which the Iteration limit is compared is reset on each new subproblem, so this limit usually is not reached.

4. Optimize_Status_Unbounded

This message appears when the Solver is able to increase (if you are trying to Maximize) or decrease (for Minimize) without limit the value calculated by the objective or Set Cell, while still satisfying the constraints. Remember that, if you've selected Minimize, the objective may take on negative values

without limit unless this is prevented by the constraints or bounds on the variables. Set the Assume Non-Negative parameter to True to impose ≥ 0 bounds on all variables.

If the objective is a linear function of the decision variables, it can *always* be increased or decreased without limit (picture it as a straight line), so the Solver will seek the extreme value that still satisfies the constraints. If the objective is a nonlinear function of the variables, it may have a "natural" maximum or minimum (for example, $=A1*A1$ has a minimum at zero), or no such limit (for example, $=\text{LOG}(A1)$ increases without limit).

If you receive this result, you may have forgotten a constraint, or failed to anticipate values for the variables that allow the objective to increase or decrease without limit. The final values for the variable cells, the constraint left hand sides and the objective should provide a strong clue about what happened.

The Evolutionary Solver *never* displays this result, because it has no way of systematically increasing (or decreasing) the objective function, which may be non-smooth. If you have forgotten a constraint, the Evolutionary Solver *may* find solutions with very large (or small) values for the objective – thereby making you aware of the omission – but this is not guaranteed.

5. Optimize_Status_Unfeasible

This result occurs when the Solver could not find *any* combination of values for the decision variables that allows all of the constraints to be satisfied *simultaneously*. If you are using the LP/Quadratic Solver or the SOCP Barrier Solver, and the model is well scaled, the Solver has determined for *certain* that there is no feasible solution.

If you are using the nonlinear GRG Solver, the GRG method (which always starts from the initial values of the variables) was unable to find a feasible solution; but there could be a feasible solution far away from these initial values, which the Solver might find if you run it with different initial values for the variables.

If you are using the Evolutionary Solver, the evolutionary algorithm was unable to find a feasible solution; it might succeed in finding one if you run it with different initial values for the variables and/or increase the Precision parameter value (which reduces the infeasibility penalty, thereby allowing the evolutionary algorithm to explore more "nearly feasible" points).

If you are solving a problem with chance constraints using simulation optimization, this result means that the Solver could find no solution that satisfies these constraints to the chance measures (such as 95%) that you specified. If you 'relax' the chance measures (to say 90%) and solve again, it's possible that a feasible solution will be found.

In any case, you should first look for conflicting constraints, i.e. conditions that *cannot* be satisfied simultaneously. Most often this is due to choosing the wrong relation (e.g. $\leq$ instead of $\geq$) on an otherwise appropriate constraint. The easiest way to do this is to use `problem.Solver.IISFind()` to create an Irreducibly Infeasible Set. `IISFind()` can take time for the LP/Quadratic Solver or SOCP Barrier Solver, and *more* time for the GRG Solver. It is not available for the Evolutionary Solver.) For an example on using `IISFind()`, please see below.

If the status of `prob.Solver.Optimize()` is `Optimize_Status_Unfeasible`, then call `problem.Solver.IISFind` to create an Irreducible Infeasible Set.

```
if(status == Optimize_Status_Unfeasible)
{
    problem.Solver.IISFind();
}
```

Cycle through the variables that are included in the Irreducible Infeasible Set. A `BoundStatus = 0` indicates that the lower bound of the variable is contributing to the infeasibility. A `BoundStatus = 1` indicates that both the upper and lower bounds of the variable are contributing to the infeasibility and a `BoundStatus = 2` indicates that the upper bound of the variable is contributing to the infeasibility.

```
int i;
```

```

for(i = 0; i < problem.Solver.OptIIS.NumBounds; i++)
cout << "Bound[" << problem.Solver.OptIIS.BoundIndex[i] << "],
status is " << problem.Solver.OptIIS.BoundStatus[i] << endl();

Cycle through the functions that are included in the Irreducible Infeasible Set. A ConstraintStatus = 0
indicates that the lower bound of the constraint is contributing to the infeasibility. A BoundStatus = 1
indicates that both the upper and lower bounds of the constraint are contributing to the infeasibility and
a BoundStatus = 2 indicates that the upper bound of the constraint is contributing to the infeasibility.

for(i = 0; i < problem.Solver.OptIIS.NumConstraints; i++)
cout << "Constraint[" << problem.Solver.OptIIS.ConstraintIndex[i] <<
"], status is " << problem.Solver.OptIIS.ConstraintStatus[i] <<
endl();
}

```

6. Optimize_Status_User_Abort

This result is only returned if you stop the SDK during the solution process, such as returning *Engine_Action_Abort* inside a Progress evaluator.

7. Optimize_Status_Linear_Invalid

In the Solver SDK, this result is returned if you are using an evaluator to compute the objective and constraints and you've selected the LP Simplex in the Solver SDK Pro or the LP/Quadratic Solver in the Solver SDK Platform and the Solver's tests determine that the constraints are not linear functions of the variables or the objective is not a linear or convex quadratic function of the variables; *or* if you've selected the SOCP Barrier Solver (included only in the Solver SDK Platform) and the Solver's tests determine that the constraints or the objective are not linear or convex quadratic functions of the variables.

Field-installable Solver engines can also display this message. If you've selected the Large-Scale LP/QP Solver or the XPRESS Solver Engine, this message appears if the constraints are not linear functions of the variables or the objective is not a linear or convex quadratic function of the variables. If you've selected the MOSEK Solver Engine (Standard Edition), this message appears if the constraints or the objective are not linear or convex quadratic functions of the variables.

If you receive this message, examine the formulas for the objective and constraints inside of your evaluator for nonlinear or non-smooth functions or operators applied to the decision variables.

8. Optimize_Status_Bad_Dataset

This message is returned when the Solver determines that your model is too large for the Solver engine that is selected. You'll have to select – or possibly install – another Solver engine appropriate for your problem, or else reduce the number of variables, constraints, or integer variables to proceed.

You can check the size limits of the Solver engine you are using, by calling *prob.Engine.Limit()*.

9. Optimize_Status_Float_Err

This message appears when the Solver recalculates your evaluator using a new set of values for the decision variables and discovers an error value when computing the constraints or objective function. Inspecting the evaluator for errors in calculations will usually indicate the source of the problem.

The Evolutionary Solver and the field-installable OptQuest Solver rarely, if ever, display this message – since they maintain a *population* of candidate solutions and can generate more candidates without relying on derivatives, they can simply discard trial solutions that result in error values in the objective or the constraints. If you have a model that frequently yields error values for trial solutions generated by the Solver, and you are unable to correct or avoid these error values by altering your formulas or by imposing additional constraints, you can still use the Evolutionary Solver or OptQuest Solver to find (or make progress towards) a "good" solution.

10. Optimize_Status_Time_Out

This message appears when the Solver has run for the maximum time (number of seconds) specified for the Max Time parameter. You may increase the value for this parameter, but you should also consider whether re-scaling your model or adding constraints might reduce the total solution time required.

11. Optimize_Status_Memory_Dearth

This result is returned when the Solver is unable to allocate the memory it needs to solve the problem. However, since Microsoft Windows supports a "virtual memory" much larger than your available RAM by swapping data to your hard disk, before you see this message you are likely to notice that solution times have greatly slowed down, and the hard disk activity light in your PC is flickering during the solution process. You can save some memory by closing any Windows applications not being used by your program and closing programs that run in the System Tray.

12. Optimize_Status_Interpret_Err

This result is rarely returned. However, when it is returned, it indicates an interpretation error has occurred.

13. Optimize_Status_API_Err

This result is rarely returned. However, when it is returned, it indicates a fatal API error has occurred.

14. Optimize_Status_Incumb_Cand

If you are solving a mixed-integer programming problem (any problem with integer constraints) with a *non-zero value* for the integer Tolerance parameter, this result indicates that the Branch & Bound method has found a solution satisfying the constraints (including the integer constraints) where the relative difference of this solution's objective value from the *true* optimal objective value does not exceed the integer Tolerance setting. This may actually *be* the true integer optimal solution; however, the Branch & Bound method did not take the extra time to search all possible remaining subproblems to "prove optimality" for this solution. If all subproblems *were* explored (which can happen even with a non-zero Tolerance in some cases), the Solver SDK will return the result, *Optimize_Status_Optimal*.

15. Optimize_Status_Bbnode_Limit

If you are using the Evolutionary Solver, this result is returned when the Solver has found the maximum number of feasible solutions (values for the variables that satisfy all constraints) allowed by the Max Feasible Sols parameter. You may increase the value for this parameter to avoid receiving this result.

If you are using one of the other Solver engines on a problem with integer constraints, this result is returned when the Solver has found the maximum number of integer solutions (values for the variables that satisfy all constraints, including the integer constraints) allowed by the Max Integer Sols parameter setting. You may increase the value for this parameter, but you should also consider whether the problem is formulated correctly, and whether you can add constraints to "tighten" the formulation. If you are using the LP/Quadratic Solver engine, try setting the Cuts and Heuristics parameter to a more aggressive setting.

16. Optimize_Status_Bpmips_Limit

If you are using the Evolutionary Solver, this message appears when the Solver has explored the maximum number of subproblems allowed by the Max Subproblems parameter. You may increase the value for this parameter to avoid receiving this result.

If you are using one of the other Solver engines on a problem with integer constraints, this message appears when the Solver has explored the maximum number of integer subproblems (each one is a "regular" Solver problem with additional bounds on the variables) allowed by the Max Subproblems parameter setting. You may increase the value for this parameter, but you should also consider whether the problem is formulated correctly, and whether you can add constraints to "tighten" the

formulation. If you are using the LP/Quadratic Solver engine, try activating a more aggressive strategy for the Cuts and Heuristics parameter.

17. Optimize_Status_Probable

If you are using the multistart methods for global optimization, with either the nonlinear GRG Solver or a field-installable nonlinear Solver engine (by setting the Global Optimization parameters), this result is returned when the multistart method's Bayesian test has determined that all of the locally optimal solutions have *probably* been found; the solution found is the best of these locally optimal solutions, and is *probably* the globally optimal solution to the problem.

The Bayesian test initially assumes that the number of locally optimal solutions to be found is equally likely to be 1, 2, 3, ... etc. up to infinity, and that the relative sizes of the regions containing each locally optimal solution follow a uniform distribution. After each run of the nonlinear GRG Solver or field-installable Solver engine, an updated estimate of the most probable total number of locally optimal solutions is computed, based on the number of subproblems solved and the number of locally optimal solutions found so far. When the number of locally optimal solutions actually found so far is within one unit of the most probable total number of locally optimal solutions, the multistart method stops and displays this message.

18. Optimize_Status_Bounds_Missing

If you are using the Evolutionary Solver or the multistart methods for global optimization, and you have set the "Require Bounds on Variables" parameter to 1 (it is set to 1 by default), this result will be returned. You should add the missing bounds and try again. (Lower bounds of zero can be applied to all variables by calling *prob.VarDecision.NonNegative()*.) You *must* define bounds on all variables in order to use the OptQuest Solver. For the Evolutionary Solver or the multistart methods, such bounds are not absolutely required (you can set the "Require Bounds on Variables" parameter to False), but they are a practical necessity if you want the Solver to find good solutions in a reasonable amount of time.

19. Optimize_Status_Bounds_Conflict

This result is returned if you have both a binary or alldifferent constraint on a decision variable and a $\leq$ or $\geq$ constraint on the same variable (that is inconsistent with the binary or alldifferent specification), or if the same decision variable appears in more than one alldifferent constraint. Binary integer variables always have a lower bound of 0 and an upper bound of 1; variables in an alldifferent group always have a lower bound of 1 and an upper bound of N, where N is the number of variables in the group. You should check that the binary or alldifferent constraint is correct, and ensure that alldifferent constraints apply to non-overlapping groups of variables. If a $\leq$ or $\geq$ constraint causes the conflict, remove it if possible and try to solve again.

20. Optimize_Status_Bounds_Inconsist

This result is returned if you've defined lower and upper bounds on a decision variable, where the lower bound is greater than the upper bound.

21. Optimize_Status_Derivative_Error

This result is returned when the Jacobian Evaluator (*Evaluator_Type_Gradient*) has not been able to compute exact gradients.

22. Optimize_Status_Cone_Overlap

This result is returned if the same decision variable appears in more than one cone constraint. You can define as many cone constraints as you want, but each one must constrain a different group of decision variables.

23. Optimize_Status_Formula_Uncertain

This result can only be returned when a model, created with Analytic Solver for Excel containing uncertainty has been loaded into the Solver SDK Platform.

This result occurs when an objective or constraint containing uncertain parameters has not been summarized to a single value (such as a mean or percentile value) or transformed into a set of single-valued constraints. To continue, the Excel model must be corrected before being loaded in to the Solver SDK Platform. Please refer to the Frontline Solver Reference Guide for help with troubleshooting this result. (See Solver Result Message 23 in the Solver Result Messages chapter.)

26. Optimize_Status_Robust_Infeas

This result can only be returned when a model created with Analytic Solver for Excel containing uncertainty has been loaded into the Solver SDK Platform.

This result may be returned when you load and solve a model with uncertainty and chance constraints using robust optimization created with Analytic Solver for Excel. When you do this, the Psi Interpreter inside Solver SDK Platform transforms your original model with uncertainty into a *robust counterpart* model that is a conventional optimization problem without uncertainty.

This result means that the Solver could not find a feasible solution to the robust counterpart problem. It does not *necessarily* mean that there is no feasible solution to the original problem; the robust counterpart is an *approximation* to the problem defined by your chance constraints that may yield conservative solutions which *over-satisfy* the chance constraints.

When this result appears, you can set the SDK model parameter *ChanceAutoAdjust* to 1 and resolve. (See *CModelParam* in the **Solver SDK Reference Guide** for details on this parameter). The SDK will then re-solve the problem, automatically adjusting the sizes of robust optimization *uncertainty sets* created for the chance constraints, in an effort to find a feasible solution.

If you have already set this parameter to 1 or the parameter *Auto Adjust Chance Constraints* on the Platform tab of the Task Pane in Analytic Solver for Excel has been set to True, then the automatic improvement algorithm has already been tried and this algorithm was unable to find a feasible solution. In this case, you should proceed as described for result code 5, *Optimize_Status_Unfeasible*. Look for conflicting constraints, i.e. conditions that *cannot* be satisfied simultaneously, perhaps due to choosing the wrong relation (e.g. $\leq$ instead of $\geq$) on an otherwise appropriate constraint.

27. Optimize_Status_Robust_Conservative

This result can only be returned when a model created with Analytic Solver for Excel containing uncertainty has been loaded into the Solver SDK Platform.

This result may be returned when you solve a model with uncertainty and chance constraints using robust optimization. When you do this, the Solver transforms your original model with uncertainty into a *robust counterpart* model that is a conventional optimization model without uncertainty.

The message means that the Solver found an optimal solution to the robust counterpart model, but when this solution was tested against your *original* model (using Monte Carlo simulation to test satisfaction of the chance constraints), the solution *over-satisfied* the chance constraints; this normally means that the solution is ‘conservative’ and the objective function value can be further improved.

When this message appears, try setting the parameter, *ChanceAutoAdjust* to 1 and resolve. (See *CModelParam* in the **Solver SDK Reference Guide** for details on this parameter.) The Solver will then re-solve the problem, automatically adjusting the sizes of robust optimization *uncertainty sets* created for the chance constraints, in an effort to improve the solution.

An alternative course of action is to *manually* adjust the Chance measures of selected chance constraints, and re-solve the problem. The automatic improvement algorithm uses general-purpose methods to find an improved solution; you may be able to do better by adjusting Chance measures based on your knowledge of the problem. Please see below for an example of how to adjust the Chance measure inside the Solver SDK Platform.

The lines of code below illustrate how to change the chance measure for the constraint located in the Excel cell C23 to 90% for all four programming languages.

```
C/C++: prob.Functions[L"C23"].Chance = 0.90;
```



```
C#: prob.Functions["C23"].Chance = 0.90;
VB.NET: prob.Functions("C23").Chance = 0.90
VB6: prob.Functions("C23").Chance= 0.90
```

28. Optimize_Status_Robust_Converge

This result can only be returned when a model created with Analytic Solver for Excel containing uncertainty has been loaded into the Solver SDK Platform.

This result may be returned when you solve a model with uncertainty and chance constraints using robust optimization, and *Auto Adjust Chance Constraints* is set to True on the Platform tab of the Task Pane in Analytic Solver for Excel. It means that the Solver has found the best ‘improved solution’ it can; the normal constraints are satisfied, and the chance constraints are satisfied to the Chance level that you specified.

This is usually a very good solution, but it does not rule out the possibility that you may be able to find an even better solution by manually adjusting Chance measures based on your knowledge of the problem, and re-solving. The lines of code below illustrate how to change the chance measure for the constraint located in the Excel cell C23 to 90% for all four programming languages.

```
C/C++: prob.Functions[L"C23"].Chance = 0.90;
C#: prob.Functions["C23"].Chance = 0.90;
VB.NET: prob.Functions("C23").Chance = 0.90
VB6: prob.Functions("C23").Chance= 0.90
```

29. Optimize_Status_Robust_Improve

This result can only be returned when a model created with Analytic Solver for Excel containing uncertainty has been loaded into the Solver SDK Platform.

This result may be returned when you solve a model with uncertainty and chance constraints using robust optimization, and you’ve set the option *Auto Adjust Chance Constraints* to True on the Platform tab of the Task Pane in Analytic Solver for Excel. It means that the Solver could not find an improved solution that satisfies *all* of the chance constraints to the Chance level that you specified in the Solver Parameters dialog. Typically in this case, some of the chance constraints will be satisfied to the level you specified, or even *over*-satisfied, but others will be *under*-satisfied.

You may find that the solution is acceptable, but if the chance constraints *must* be satisfied to the level you specified, further work will be required. You may be able to find an improved solution by manually adjusting Chance measures based on your knowledge of the problem, and re-solving. The lines of code below change the chance measure for the constraint located in the Excel cell C23 to 90% for all four programming languages.

```
C/C++: prob.Functions[L"C23"].Chance = 0.90;
C#: prob.Functions["C23"].Chance = 0.90;
VB.NET: prob.Functions("C23").Chance = 0.90
VB6: prob.Functions("C23").Chance= 0.90
```

999. Optimize_Status_Exception

This result is returned when an exception has been caught by a try/catch system.

Problems with Poorly Scaled Models

A *poorly scaled* model is one that computes values of the objective, constraints, or intermediate results that differ by several orders of magnitude. A classic example is a financial model that computes a dollar amount in millions or billions and a return or risk measure in fractions of a percent. Because of

the finite precision of computer arithmetic, when these values of very different magnitudes (or others derived from them) are added, subtracted, or compared – in the user's model or in the Solver's own calculations – the result will be accurate to only a few significant digits. After many such steps, the Solver may detect or suffer from "numerical instability."

The effects of poor scaling in a large, complex optimization model can be among the most difficult problems to identify and resolve. It can cause Solver engines to return results such as *Optimize_Status_Unfeasible*, *Optimize_Status_No_Remedies*, or even *Optimize_Status_Linear_Invalid*, with results that are suboptimal or otherwise very different from your expectations. The effects may not be apparent to you, given the initial values of the variables, but when the Solver explores Trial Solutions with very large or small values for the variables, the effects will be greatly magnified.

Dealing with Poor Scaling

Most Solver engines include a Use Automatic Scaling parameter. When this option is set to True, the Solver rescales the values of the objective and constraint functions internally in order to minimize the effects of poor scaling. But this can only help with the Solver's own calculations – it cannot help with poorly scaled results that arise *in the middle of the solution process*.

The best way to avoid scaling problems is to carefully choose the "units" implicitly used in your model so that all computed results are within a few orders of magnitude of each other. For example, if you express dollar amounts in units of (say) millions, the actual numbers computed on your Problem may range from perhaps 1 to 1,000,000.

Limitations on Smooth Nonlinear Optimization

Nonlinear problems are intrinsically more difficult to solve than linear problems, and there are fewer guarantees about what the Solver can do. If your smooth nonlinear problem is **convex**, the Solver will normally find the *globally optimal* solution (subject to issues of poor scaling and the finite precision of computer arithmetic). But if your problem is **non-convex**, the Solver will normally find only a *locally optimal* solution, close to the starting values of the decision variables, when you click Solve.

When dealing with a **non-convex** problem, it is a good idea to run the Solver starting from several different sets of initial values for the decision variables. Since the Solver follows a path from the starting values (guided by the direction and curvature of the objective function and constraints) to the final solution values, it will normally stop at a peak or valley closest to the starting values you supply. By starting from more than one point – ideally chosen based on your own knowledge of the problem – you can increase the chances that you have found the best possible "optimal solution." An easy way to do this is to set the **multistart** parameter to True to *automatically* run the Solver from multiple starting points.

GRG Solver Stopping Conditions

It is helpful to understand what the nonlinear GRG Solver can and cannot do, and what each of the possible Solver Result Messages means for this Solver engine. At best, the GRG Solver alone – like virtually all "classical" nonlinear optimization algorithms – can find a *locally optimal* solution to a reasonably *well-scaled*, non-convex model. At times, the Solver will stop *before* finding a locally optimal solution, when it is making very slow progress (the objective function is changing very little from one trial solution to another) or for other reasons.

Locally Versus Globally Optimal Solutions

When the first message (*Optimize_Status_Optimal*) appears, it means that the GRG Solver has found a *locally optimal* solution – there is no other set of values for the decision variables close to the current values that yields a better value for the objective function. Figuratively, this means that the Solver has

found a "peak" (if maximizing) or "valley" (if minimizing) – but if the model is non-convex, there may be other taller peaks or deeper valleys far away from the current solution. Mathematically, this message means that the Karush - Kuhn - Tucker (KKT) conditions for local optimality have been satisfied (to within a certain tolerance, related to the Precision parameter).

When Solver has Converged to the Current Solution

When the GRG Solver's second stopping condition is satisfied (*before* the KKT conditions are satisfied), the second result *Optimize_Status_Converged* is returned. This means that the objective function value is changing very slowly for the last few iterations or trial solutions. More precisely, the GRG Solver stops if the absolute value of the *relative* change in the objective function is less than the value for the Convergence parameter for the last 5 iterations. While the default value of 1E-4 (0.0001) is suitable for most problems, it may be too large for some models, causing the GRG Solver to stop prematurely when this test is satisfied, instead of continuing for more iterations until the KKT conditions are satisfied.

A *poorly scaled* model is more likely to trigger this stopping condition, even if the Use Automatic Scaling parameter is set to True. So it pays to design your model to be reasonably well scaled in the first place: The typical values of the objective and constraints should not differ from each other, or from the decision variable values, by more than three or four orders of magnitude.

If you are getting this message when you are seeking a locally optimal solution, you can change the setting in the Convergence edit box to a smaller value such as 1E-5 or 1E-6; but you should also consider why it is that the objective function is changing so slowly. Perhaps you can add constraints or use different starting values for the variables, so that the Solver does not get "trapped" in a region of slow improvement.

When Solver Cannot Improve the Current Solution

The third stopping condition, which yields the result *Optimize_Status_No_Remedies*, occurs only rarely. It means that the model is *degenerate* and the Solver is probably *cycling*. The issues involved are beyond the level of this Reference Guide. One possibility worth checking is that some of your constraints are redundant, and should be removed. If this suggestion doesn't help and you cannot reformulate the problem, try using the Evolutionary Solver. To go further with the GRG Solver, you may need specialized consulting assistance.

GRG Solver with Multistart Methods

The multistart methods for global optimization included in Solver SDK can overcome some of the limitations of the GRG Solver alone, but they are not a panacea. The multistart methods will automatically run the GRG Solver (or a field-installable nonlinear Solver engine) from a number of starting points and will display the best of several locally optimal solutions found, as the probable globally optimal solution. Because the starting points are selected at random and then "clustered" together, they will provide a reasonable degree of "coverage" of the space enclosed by the bounds on the variables. The *tighter the variable bounds* you specify and the longer the Solver runs, the better the coverage.

However, the performance of the multistart methods is generally limited by the performance of the GRG Solver on the subproblems. If the GRG Solver stops prematurely due to slow convergence, or fails to find a feasible point on a given run, the multistart method can improve upon this only by finding another starting point from which the GRG Solver can find a feasible solution, or a better locally optimal solution, by following a different path into the same region.

If the GRG Solver reaches the same locally optimal solution on many different runs initiated by the multistart method, this will tend to decrease the Bayesian estimate of the number of locally optimal solutions in the problem, causing the multistart methods to stop relatively quickly. In many cases this indicates that the globally optimal solution has been found – but you should always inspect and think

about the solution, and consider whether you should run the GRG Solver manually from starting points selected based on your knowledge of the problem.

Nonlinear GRG Solver and Integer Constraints

Like the multistart methods, the performance of the Branch & Bound method on nonlinear problems with integer constraints is limited by the performance of the GRG Solver on the subproblems. If the GRG Solver stops prematurely due to slow convergence, or fails to find a feasible point on a given run, this may prevent the Branch & Bound method from finding the true integer optimal solution. In most cases, the combination of the Branch & Bound method and the GRG Solver will at least yield a relatively good integer solution. However, if you are unable to find a sufficiently good solution with this combination of methods, consider using one of the field-installable nonlinear Solver engines.

Limitations on Non-Smooth Optimization

Non-smooth problems – where the objective and/or constraints are computed with discontinuous or non-smooth Excel functions – are the most difficult types of optimization problems to solve. There are few, if any, guarantees about what the Solver (or *any* optimization method) can do with these problems.

The most common discontinuous function is the IF function where the conditional test is dependent on the decision variables. Common non-smooth functions are ABS, MIN and MAX, INT and ROUND.

If your optimization problem contains discontinuous or non-smooth functions, your simplest course of action is to use the Evolutionary Solver to find a "good" solution. You should read the section "Evolutionary Solver Stopping Conditions" below and the discussion earlier in this chapter of specific Solver Result Messages, to ensure that you understand what the various messages say about your model. You can try using the nonlinear GRG Solver, or even the linear Simplex Solver, on problems of this type, but you should be aware of the effects of non-smooth functions on these Solver engines, which are summarized below.

Effect on the Nonlinear GRG and Simplex Solvers

A smooth nonlinear solver, such as the GRG Solver, relies on derivative or gradient information to guide it towards a feasible and optimal solution. Since it is unable to compute the gradient of a function at points where the function is discontinuous, or to compute curvature information at points where the function is non-smooth, it cannot guarantee that any solution it finds to such a problem is truly optimal. In practice, the GRG Solver can sometimes deal with discontinuous or non-smooth functions that are "incidental" to the problem, but as a general statement, this Solver engine requires smooth nonlinear functions for the objective and constraints.

If you try to solve a problem with non-smooth or discontinuous functions with the LP/Quadratic or LP Simplex Solver it is possible – though very unlikely – that the linearity test performed by the Solver will not detect the discontinuities and will proceed to try to solve the problem. (This probably means that the functions *are* linear over the range considered by the linearity test – but there are no guarantees at all that the solution found is optimal!)

Evolutionary Solver Stopping Conditions

It is helpful to understand what the Evolutionary Solver can and cannot do, and what each of the possible Solver Result Messages means for this Solver engine. At best, the Evolutionary Solver – like other genetic or evolutionary algorithms – will be able to find a *good* solution to a reasonably *well-scaled* model. Because the Evolutionary Solver does not rely on derivative or gradient information, it cannot determine whether a given solution is optimal – so it never really *knows* when to stop. Instead, the Evolutionary Solver stops and returns a solution either when certain heuristic rules (discussed

below) indicate that further progress is unlikely, or else when it exceeds a limit on computing time or effort that you've set.

"Good" Versus Optimal Solutions

The Evolutionary Solver makes almost no assumptions about the mathematical properties (such as continuity, smoothness or convexity) of the objective and the constraints. Because of this, it *actually has no concept of an "optimal solution,"* or any way to test whether a solution is optimal. The Evolutionary Solver knows only that a solution is "better" in comparison to other solutions found earlier. It may sometimes find the true optimal solution, on models with a limited number of variables and constraints; on such models, the heuristic stopping rules discussed below may cause the Solver to stop at an appropriate time and report this solution. But the Evolutionary Solver will not be able to *tell you* that this solution is optimal.

When you use the Evolutionary Solver, you may find – like other users of genetic and evolutionary algorithms – that you spend a lot of time running and re-running the Solver, trying to find better solutions. This is an inescapable consequence of using a Solver engine that makes few or no assumptions about the nature of the problem functions. You can never be sure whether you've found the best solution, or what the payoff might be of running the evolutionary algorithm for a longer time. When the Evolutionary Solver stops, you may very well find that, if you keep the resulting solution and restart the Evolutionary Solver, it will find an even better solution. You may also find that starting the GRG Solver from the point where the Evolutionary Solver stops will yield a better (sometimes *much* better) solution.

When Solver returns Optimize_Status_Converged

This message means that the "fitness" of members of the current population of trial solutions is changing very slowly. More precisely, the Evolutionary Solver stops if 99% or more of the members of the population have "fitness" values whose relative (i.e. percentage) difference is less than the Convergence tolerance parameter setting. This condition may mean that the Solver has found a globally optimal solution – if so, new members of the population (that replace other, less fit members) will tend to "crowd around" this solution. However, it may also mean that the population has lost diversity – a common problem in genetic and evolutionary algorithms – and hence the evolutionary algorithm is unable to generate new and better solutions through mutation or crossover of current population members.

When Solver returns Optimize_Status_No_Remedies

This message means that the Solver has been unable to find a new, better member of the population whose "fitness" represents a relative (percentage) improvement over the current best member's fitness of more than the Tolerance parameter value, in the amount of time specified by the Max Time without Improvement parameter setting. Under this heuristic stopping rule, the Evolutionary Solver will continue searching for better solutions as long as it is making the degree of progress that you have indicated via the Tolerance value; if it is unable to make that much progress in the time you've specified, the Solver will stop and report the best solution found.

Evaluating a Solution Found by the Evolutionary Solver

Once you have a solution from the Evolutionary Solver, what can you do with it? Here are some ideas:

1. Keep the resulting solution, restart the Evolutionary Solver from that solution, and see if it is able to find an even better solution in a reasonable length of time.
2. Tighten the Convergence and Tolerance parameter settings, increase the Max Subproblems and Max Feasible Sols parameter settings, and restart the Evolutionary Solver. This will take more time, but will allow the Solver to explore more possibilities.

3. Increase the Population Size and/or the Mutation Rate, and restart the Evolutionary Solver. This will also take more time, but will tend to increase the diversity of the population and the portion of the search space that is explored.
4. Keep the resulting solution, switch to the GRG Solver and start it from that solution, and see if it finds the same or a better solution. If the GRG Solver stops with the result `Optimize_Status_Optimal` you may have found at least a *locally optimal* point (but remember that this test depends on smoothness of the problem functions).

As you work with the Evolutionary Solver, you will appreciate its ability to find "good" solutions to previously intractable optimization problems, but you will also come to appreciate its limitations. The Evolutionary Solver allows you to spend less time analyzing the mathematical properties of your model, and still obtain "good" solutions – but as with the GRG Engine and the multistart methods, it is not a panacea.

If your problem is large, or if the payoff from a true optimal solution is significant, you may want to invest more effort to formulate a model that satisfies the requirements of a smooth nonlinear optimization problem, or even an integer linear problem. There are many techniques you can use to replace non-smooth functions with smooth nonlinear or integer linear expressions. With enough work, you may be able to obtain a significantly better solution with the other Solver engines, and to know with some certainty whether or not you have found the optimal solution.

Solver SDK Parameters

Setting the Parameters

Please see below for example code on how to set a parameter in C++, C#, the Procedural API, and in COM. To set, substitute the parameter's name for "Parameter" and the value of the parameter for "x".

C++: `problem.Engine.Params[L"Parameter"].Value = x;`

C#: `Problem.Engine.Params["Parameter"].Value = x;`

Procedural API: `SolverEngParamSet(prob, L"Parameter", x);`

COM: `problem.Engine.Params("Parameter").Value = x`

Common SDK Optimization Parameters

The following parameters are used to control all Solver SDK engines, including additional "plug in" engines.

Max Time

Name	MaxTime
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for Max Time determines the maximum time in seconds that the SDK will run before it stops. For problems with integer constraints, this is the total time taken to solve all subproblems explored by the Branch & Bound method.

Iterations

Name	Iterations
Default	2,147,483,647
Min	0
Max	2,147,483,647

Type	Integer

The value for the number of Iterations determines the maximum number of iterations ("pivots" for the Simplex Solver, or major iterations for the GRG Solver) that the Solver may perform on one problem. A new "Trial Solution" is generated on each iteration. For problems with integer constraints, the Iterations setting determines the maximum number of iterations for any *one* subproblem.

Precision

Name	Precision
Default	1.0e-6
Min	1.0e-4
Max	1.0e-9
Type	Double

The number entered here determines how closely the calculated values of the constraint left hand sides must match the right hand sides in order for the constraint to be satisfied. (This option is not used with the LP/Quadratic or SOCP engines.) With the default setting of 1.0E-6 (0.000001), a calculated left hand side of -1.0E-7 (0.0000001) would satisfy a constraint such as $A1 \geq 0$. Use caution in making this number much smaller, since the finite precision of computer arithmetic virtually ensures that the calculated values will differ from the expected or "true" values by a small amount. On the other hand, setting the Precision to a much larger value would cause constraints to be satisfied too easily. If your constraints are not being satisfied because the values you are calculating are very large (say in millions or billions of dollars), consider adjusting your formulas and data to work in *units of millions*, or setting the "Scaling" parameter (see explanation below) instead of altering the Precision setting. Generally, this setting should be kept in the range from 1.0E-6 (0.000001) to 1.0E-4 (0.0001).

Precision and Integer Constraints

Another use of Precision is determining whether an integer constraint, such as $A1:A5 = \text{integer}$, $A1:A5 = \text{binary}$ or $A1:A5 = \text{alldifferent}$, is satisfied. If the difference between the decision variable's value and the closest integer value is less than the Precision setting, the variable value is treated as an integer.

Use Automatic Scaling

Name	Scaling
Default	0 – On
Min	-1 – Off
Max	1 – On
Type	Double

When Automatic Scaling is turned on (set to 0 or 1), the Solver will attempt to scale the values of the objective and constraint functions internally in order to minimize the effects of a poorly scaled model. A *poorly scaled* model is one that computes values of the objective, constraints, or intermediate results that differ by several orders of magnitude. Poorly scaled models may cause difficulty for both linear and nonlinear solution algorithms, due to the effects of finite precision computer arithmetic.

If your model is nonlinear and you turn on Automatic Scaling, *make sure that the initial values for the decision variables are "reasonable,"* i.e. of roughly the same magnitudes that you expect for those variables at the optimal solution. The effectiveness of the Automatic Scaling option depends on how well these starting values reflect the values encountered during the solution process.

Irreducible Infeasible Set – Bounds

Name	IISBounds
Default	0 – Include Bounds
Min	0
Max	1 – No Bounds
Type	Integer

Instead of the full Irreducible Infeasible Set (IIS), which analyzes both the constraints and variable bounds in your model and attempts to eliminate as many of them as possible, you can produce the IIS minus the variable bounds. The IIS minus the variable bounds analyzes only the constraints while keeping the variable bounds in force. This may be sufficient to isolate the source of the infeasibility, but you must take into account the bounds on all of the variables when analyzing the IIS.

Controlling Use of Processor Cores

Name	NumThreads
Default	0 – Use Maximum Number of Threads Available
Min	0
Max	Number of cores on machine.
Type	Integer

Solver SDK uses Windows facilities to create multiple “threads” of execution that can be run on different processor cores. If you have other compute-intensive applications running at the same time as Solver SDK, you can control the number of cores used for each Problem. The default value of 0 means: “use as many threads as there are processor cores in the machine.” (The actual number of threads used may vary dynamically during execution.) You can instead set this to a specific number of threads.

Run Specific Optimization

Name	RunSpecificOpt
Default	-1
Min	-1
Max	2,147,483,647
Type	Integer

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

LP Simplex Solver Options

In the Solver SDK Pro, the linear Simplex Solver is included as a bundled engine. The parameters above plus the Pivot Tolerance and Reduced Cost Tolerance parameters below are all the parameters used by the Simplex Solver.

Pivot Tolerance

Name	PivotTol
Default	1.0e-7
Min	0
Max	1
Type	Double

The Simplex method looks for a non-zero matrix element to pivot upon in its basic iteration. Any matrix element with absolute value less than this tolerance is treated as zero for this purpose.

Reduced Cost Tolerance

Name	ReducedTol
Default	1e-5
Min	1e-9
Max	1e-4
Type	Double

The Simplex method looks for a variable to enter the basis that has a negative reduced cost. The candidates are only those variables that have reduced costs less than the negative value of this tolerance.

Solution Tolerance

Name	Solution
Default	1e-4
Min	1e-9
Max	1e-5
Type	Double

Any calculated basic variable whose absolute value is less than this tolerance is treated as zero.

LP Simplex Solver MIP Parameters

This section describes the parameters which control the LP Simplex engine when solving a mixed integer problem. The parameters above plus the parameters in this section (Use Dual Simplex for Subproblems, Preprocessing and Probing, Bounds Improvement, Optimality Fixing, Primal Heuristic, Gomory Cuts, Gomory Passes, Knapsack Cuts and Knapsack Passes) are all the parameters used by the Simplex Solver.

Max Subproblems

Name	MaxSubproblems
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Subproblems parameter places a limit on the number of subproblems that may be explored by the Branch & Bound algorithm before the SDK stops with the status result, *Optimize_Status_Bpmips_Limit*. Each subproblem is a "regular" Solver problem with additional bounds on the variables. In a problem with integer constraints, the Max Subproblems limit should be used in preference to the Iterations limit.

Max Feasible (Integer) Solutions

Name	MaxIntegerSols
	MaxFeasibleSols
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Integer Solutions parameter places a limit on the number of "candidate" integer solutions found by the Branch & Bound algorithm before the SDK stops with the status result, *Optimize_Status_Bpnode_Limit*. Each candidate integer solution satisfies all of the constraints, including the integer constraints; the SDK retains the integer solution with the best objective value so far, called the "incumbent."

It is entirely possible that, in the process of exploring various subproblems with different bounds on the variables, the Branch & Bound algorithm may find the same integer solution (set of values for the decision variables) more than once; the Max Integer Solutions limit applies to the total number of integer solutions found, not the number of "distinct" integer solutions.

Integer Tolerance

Name	IntTolerance
Default	0
Min	0
Max	1.0
Type	Double

When you solve an integer programming problem, it often happens that the Branch & Bound method will find a good solution fairly quickly, but will require a great deal of computing time to find (or verify that it has found) the optimal integer solution. The Integer Tolerance setting may be used to tell the Solver to stop if the best solution it has found so far is "close enough."

The Branch & Bound process starts by finding the optimal solution without considering the integer constraints (this is called the *relaxation* of the integer programming problem). The objective value of the relaxation forms the initial "best bound" on the objective of the optimal *integer* solution, which can be no better than this. During the optimization process, the Branch & Bound method finds "candidate" integer solutions, and it keeps the best solution so far as the "incumbent." By eliminating alternatives

as it proceeds, the B&B method also tightens the "best bound" on how good the integer solution can be.

Each time the Solver finds a new incumbent – an improved all-integer solution – it computes the maximum percentage difference between the objective of this solution and the current best bound on the objective:

$$\frac{\text{Objective of incumbent} - \text{Objective of best bound}}{\text{Objective of best bound}}$$

If the absolute value of this maximum percentage difference is equal to or less than the Integer Tolerance, the Solver will stop and report the current integer solution as the optimal result, with the status, *Optimize_Status_Incumb_Cand*. If you set the Integer Tolerance to zero, the Solver will "prove optimality" by continuing to search until all alternatives have been explored and the optimal integer solution has been found. This could take a great deal of computing time.

Integer Cutoff

Name	IntCutoff
Default	2.0e+30
Min	-2.0e+30
Max	2.0e+30
Type	Double

This option provides another way to save time in the solution of mixed-integer programming problems. If you know the objective value of a feasible integer solution to your problem – possibly from a previous run of the same or a very similar problem – you can enter this objective value for the Integer Cutoff parameter. This allows the Branch & Bound process to *start* with an "incumbent" objective value (as discussed above under Integer Tolerance) and avoid the work of solving subproblems whose objective can be no better than this value. If you enter a value for this parameter, you must be *sure* that there is an integer solution with an objective value at least this good: A value that is too large (for maximization problems) or too small (for minimization) may cause the SDK to skip solving the subproblem that would yield the optimal integer solution.

Use Dual Simplex for Subproblems

Name	UseDual
Default	2 - Dual
Min	1 – Primal
Max	2
Type	Integer

This option appears only in the Solver SDK Pro, when the LP Simplex Solver is selected. When this parameter is set to 2, the Solver uses the Dual Simplex method, starting from an advanced basis, to solve the subproblems generated by the Branch & Bound method. When it is cleared, the Solver uses the Primal Simplex method to solve the subproblems. Use of this option will often speed up the solution of problems with both general integer (e.g. A1:A5 = integer) and binary integer (e.g. A1:A5 = binary) variables.

The subproblems of an integer programming problem are based on the relaxation of the problem, but have additional or tighter bounds on the variables. The solution of the relaxation (or of a more direct "parent" of the current subproblem) provides an "advanced basis" which can be used as a starting point for solving the current subproblem, potentially in fewer iterations. This basis may not be primal feasible due to the additional or tighter bounds on the variables, but it is always dual feasible. Because

of this, the Dual Simplex method is usually faster than the Primal Simplex method when starting from an advanced basis.

The Dual Simplex method proceeds from a dual feasible solution to a dual optimal (and hence primal feasible) solution by dealing with the additional or tighter bounds on the variables. The Solver SDK Pro employs an advanced "bound-swapping" Dual Simplex method that becomes faster as the bounds become tighter and apply to more variables.

Preprocessing and Probing

The Preprocessing and Probing parameters appear only in the Solver SDK Pro, when the LP Simplex Solver engine is selected. Use of these options can dramatically improve solution time on problems with many 0-1 or binary integer variables. Any of them may be selected independently, but the best speed gains are often realized when they are used in combination – particularly Probing / Feasibility, Bounds Improvement and Optimality Fixing.

Probing / Feasibility

Name	ProbingFeasibility
Default	0 - Off
Min	0
Max	1 – On
Type	Integer

The Probing strategy allows the SDK 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 that can be discovered through Probing. For example, your model may have a constraint such as:

$$x_1 + x_2 + x_3 + x_4 + x_5 \leq 1$$

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.)

In a problem with a "clique" of 5 binary integer variables, without Probing, the Branch & Bound process might have to generate and solve as many as $2^5 = 32$ subproblems, with different combinations of bounds on these variables. With Probing, the number of subproblems is reduced from 32 to 5. As the number of binary integer variables increases, this strategy can clearly improve solution time.

Bounds Improvement

Name	BoundsImprovement
Default	0 - Off
Min	0
Max	1 – On
Type	Integer

The Bounds Improvement strategy allows the SDK to tighten the bounds on variables that are *not* 0-1 or binary integer variables, based on the values that have been derived for the binary variables, before the problem is solved. Tightening the bounds usually reduces the effort required by the Simplex

method (or other method) to find the optimal solution, and in some cases it leads to an immediate determination that the subproblem is infeasible and need not be solved.

Optimality Fixing

Name	OptimalityFixing
Default	0 - Off
Min	0
Max	1 – On
Type	Integer

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. But **Optimality Fixing will yield incorrect results if you have bounds on variables, such as $A1:A5 \geq 10$ and $A1:A5 \leq 10$, which create "implied" equalities**, instead of explicit equalities such as $A1:A5 = 10$. Watch out for situations such as $A1:A5 \geq 10$ and $A3:D3 \leq 10$, which creates an implied equality constraint on A3. Implied equalities of this sort are never a good practice, but they *must* be avoided in order to use Optimality Fixing.

Primal Heuristic

Name	PrimalHeuristic
Default	0 - Off
Min	0
Max	1 – On
Type	Integer

When this box is checked, the Solver uses heuristic methods to attempt to discover an integer feasible solution early in the Branch & Bound process. (A *heuristic method* is one that has been found to succeed frequently in practice, but is not *guaranteed* to succeed – in this case, to find an integer feasible solution.) The specific method used in the LP Simplex Solver is derived from the "local search" literature, where it has been found to be quite effective, especially on 0-1 integer programming problems.

If an integer feasible solution is found via the primal heuristic, it serves as an "incumbent" that enables the Branch & Bound method to cut off the exploration of other subproblems, because they cannot yield an integer solution better than the known incumbent, and thereby save time overall. Of course, if an integer feasible solution is *not* found, the time spent on the primal heuristic is wasted. Hence, you should experiment with this option to see if it yields faster solutions on your problem.

Variable Reordering and Pseudocost Branching

A Variable Reordering option was used in earlier versions of the Solver SDK products to improve the order in which the Branch & Bound algorithm chose binary integer variables to "branch" upon. It is ignored in recent versions of the SDK. The Solver SDK Platform uses a better strategy, known as "pseudocost branching," at all times.

Branching on an integer variable places tighter bounds on this variable in all subproblems derived from the current branch. In the case of a binary integer variable, branching forces the variable to be 0 or 1 in the subproblems. Tighter bounds on certain variables may have a large impact on the values that can be assumed by *other* variables in the problem. Ideally, the SDK will branch on these variables first.

For example, you might have a binary integer variable that determines whether or not a new plant will be built, and other variables that then determine whether certain manufacturing lines will be started up. If the SDK branches upon the plant-building variable first, forcing it to be 0 or 1, this will eliminate

many other possibilities that would otherwise have to be considered during the solution of each subproblem.

Pseudocost branching enables the SDK to automatically choose variables for branching, once each integer variable has been branched upon at least once. But at the beginning of the solution process, the order in which integer variables are chosen for branching is guided overall by the order in which they appear in the list below. You may be able to improve performance by manually ordering the variables in the outlined list, based on your knowledge of the problem. In the example above, you would list the plant-building binary integer variable first in the outlined list, so it will be branched upon before any other integer variable.

Cut Generation

The Cut Generation options (Gomory Cuts and Passes, and Knapsack Cuts and Passes) are available for the LP Simplex Solver in the Solver Pro SDK. A *cut* is an automatically generated linear constraint for the problem, in addition to the constraints that you specify. This constraint is constructed so that it "cuts off" some portion of the feasible region of an LP subproblem, without eliminating any possible integer solutions. Cuts add to the work that the LP solver must perform on each subproblem (and hence they do not always improve solution time), but on many integer programming problems, cut generation enables the overall Branch & Bound method to more quickly discover integer solutions, and eliminate subproblems that cannot lead to better solutions than the best one already known.

The default values for the Cut Generation options represent a reasonably good tradeoff for many models, but it may well be worthwhile to experiment with values in these edit boxes to find the best settings for your problem.

Gomory Cuts

Name	MaxGomoryCuts
Default	20
Min	0
Max	2,147,483,647
Type	Integer

The value for this parameter is the maximum number of Gomory cuts that the Solver should generate for a given subproblem. When this maximum is reached, or if there are no further cut opportunities, the SDK proceeds to solve the LP subproblem (with the cuts) via the primal or dual Simplex method.

Gomory cuts are generated by examining the basis inverse at the optimal solution of a previously solved LP relaxation of the problem. This basis inverse is sensitive to rounding error due to the use of finite precision computer arithmetic. Hence, **if you use Gomory cuts, you should take extra care to ensure that your worksheet model is well scaled, and check the Use Automatic Scaling box**. If you have trouble finding the integer optimal solution with the default settings for Gomory cuts, you may want to enter 0 for this parameter, to eliminate Gomory cuts as a possible source of problems due to rounding.

Gomory Passes

Name	GomoryPasses
Default	1
Min	0
Max	2,147,483,647
Type	Integer

The value for this parameter is the number of "passes" the SDK should make over a given subproblem, looking for Gomory cuts. When cuts are generated and added to the model, the new model may

present opportunities to generate further cuts. In fact, it's possible to solve an LP/MIP problem to optimality by generating Gomory cuts in multiple passes, without any branching via Branch & Bound; however, experience has shown that this is usually less efficient than using Branch & Bound. The default value of 1 pass is best for many models, but you may find that increasing this value improves solution time for your model.

Knapsack Cuts

Name	MaxKnapsackCuts
Default	20
Min	0
Max	2,147,483,647
Type	Integer

The value for this parameter is the maximum number of Knapsack cuts that the SDK should generate for a given subproblem. When this maximum is reached, or if there are no further cut opportunities, the SDK proceeds to solve the LP subproblem (with the cuts) via the primal or dual Simplex method. Knapsack cuts, also known as *lifted cover inequalities*, can be generated only for groups of binary integer variables, whereas Gomory cuts can be generated for any integer variables. But when knapsack cuts can be generated, they are often very effective in cutting off portions of the LP feasible region, and improving the speed of the solution process.

Knapsack Passes

Name	KnapsackPasses
Default	1
Min	0
Max	2,147,483,647
Type	Integer

The value for this parameter is the number of "passes" the Solver should make over a given subproblem, looking for Knapsack cuts. As for Gomory cuts, when Knapsack cuts are generated and added to the model, the new model may present opportunities to generate further cuts; but time spent on additional passes could otherwise be spent solving LP subproblems. The default value of 1 pass is best for many models, but you may find that increasing this value improves solution time for your model.

Solve Without Integer Constraints

Name	SolveWithout
Default	0 - Off
Min	0
Max	1 - On
Type	Integer

If you solve your Problem with this parameter set to 1, Solver SDK Platform *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem.

LP/Quadratic Solver Options

In the Solver Platform SDK, if the LP/Quadratic Solver is selected to solve the problem, the parameters for the engine include the parameters in the Common Options section above as well as the parameters described below. Note that the default values for Primal Tolerance and Dual Tolerance

have been chosen very carefully; the LP/Quadratic Solver is designed to solve the vast majority of LP problems ‘out of the box’ with these default tolerances.

Primal Tolerance

Name	PrimalTolerance
Default	1.0e-7
Min	0
Max	1.0
Type	Double

The Primal Tolerance is the maximum amount by which the primal constraints can be violated and still be considered feasible. The default value of 1.0E-7 (0.000001) is suitable for most problems.

Dual Tolerance

Name	DualTolerance
Default	1.0e-7
Min	0
Max	1.0
Type	Double

The Dual Tolerance is the maximum amount by which the dual constraints can be violated and still be considered feasible. The default values of 1.0E-7 (0.000001) is suitable for most problems.

Do Presolve

Name	Presolve
Default	1 – On
Min	0 – Off
Max	1
Type	Integer

When this parameter is set to 1 (which is the default setting), the LP/Quadratic Solver performs a Presolve step before applying the Primal or Dual Simplex method. Presolving often reduces the size of an LP problem by detecting singleton rows and columns, removing fixed variables and redundant constraints, and tightening bounds.

Derivatives for the Quadratic Solver

Name	Derivatives
Default	1 – Forward
Min	1
Max	2 – Central
Type	Integer

When a quadratic programming (QP) problem – one with a quadratic objective and all linear constraints – is solved with the LP/Quadratic Solver, the quadratic Solver extension requires first or

second partial derivatives of the objective function at various points. In the Solver SDK Platform, these derivatives are computed via *finite differencing* and the method used for finite differencing is determined by the setting of the Derivatives parameter. *Forward* differencing uses the point from the previous iteration – where the problem function values are already known – in conjunction with the current point. *Central* differencing relies only on the current point, and perturbs the decision variables in opposite directions from that point. For QP problems, the Central differencing choice yields essentially exact (rather than approximate) derivative values, which can improve solution accuracy and reduce the total number of iterations; however the initial computation of derivatives may take up to twice as long as with Forward differencing.

Solve Without Integer Constraints

Name	SolveWithout
Default	0 - Off
Min	0
Max	1 – On
Type	Integer

If you solve your Problem with this parameter set to 1, Solver SDK Platform *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem.

LP/Quadratic Solver MIP Parameters

This section describes the parameters which control the LP/Quadratic engine when solving a mixed integer problem. This engine contains an extensive set of options to improve performance on problems that contain integers.

Max Subproblems

Name	MaxSubproblems
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Subproblems parameter places a limit on the number of subproblems that may be explored by the Branch & Bound algorithm before the SDK stops with the status result, *Optimize_Status_Bpmips_Limit*. Each subproblem is a "regular" Solver problem with additional bounds on the variables. In a problem with integer constraints, the Max Subproblems limit should be used in preference to the Iterations limit.

Max Feasible (Integer) Solutions

Name	MaxFeasibleSols
	MaxIntegerSols
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Feasible Sols parameter places a limit on the number of feasible integer solutions found by the Branch & Bound algorithm before the SDK stops with the status result, *Optimize_Status_Bpnode_Limit*. Each feasible integer solution satisfies all of the constraints, including the integer constraints; the Solver retains the integer solution with the best objective value so far, called the "incumbent."

It is entirely possible that, in the process of exploring various subproblems with different bounds on the variables, the Branch & Bound algorithm may find the same feasible integer solution (set of values for the decision variables) more than once; the Max Feasible Solutions limit applies to the total number of integer solutions found, not the number of "distinct" integer solutions.

Integer Tolerance

Name	IntTolerance
Default	0
Min	0
Max	1.0
Type	Double

When you solve an integer programming problem, it often happens that the Branch & Bound method will find a good solution fairly quickly, but will require a great deal of computing time to find (or verify that it has found) the optimal integer solution. The Integer Tolerance setting may be used to tell the Solver to stop if the best solution it has found so far is "close enough."

The Branch & Bound process starts by finding the optimal solution without considering the integer constraints (this is called the *relaxation* of the integer programming problem). The objective value of the relaxation forms the initial "best bound" on the objective of the optimal *integer* solution, which can be no better than this. During the optimization process, the Branch & Bound method finds "candidate" integer solutions, and it keeps the best solution so far as the "incumbent." By eliminating alternatives as it proceeds, the B&B method also tightens the "best bound" on how good the integer solution can be.

Each time the Solver finds a new incumbent – an improved all-integer solution – it computes the maximum percentage difference between the objective of this solution and the current best bound on the objective:

$$\frac{\text{Objective of incumbent} - \text{Objective of best bound}}{\text{Objective of best bound}}$$

If the absolute value of this maximum percentage difference is equal to or less than the Integer Tolerance, the Solver will stop and report the current integer solution as the optimal result, with the status, *Optimize_Status_Incumb_Cand*. If you set the Integer Tolerance to zero, the Solver will "prove optimality" by continuing to search until all alternatives have been explored and the optimal integer solution has been found. This could take a great deal of computing time.

Integer Cutoff

Name	IntCutoff
Default	2.0e+30
Min	-2.0e-30
Max	2.0e+30
Type	Double

This option provides another way to save time in the solution of mixed-integer programming problems. If you know the objective value of a feasible integer solution to your problem – possibly from a

previous run of the same or a very similar problem – you can enter this objective value for the Integer Cutoff parameter. This allows the Branch & Bound process to *start* with an "incumbent" objective value (as discussed above under Integer Tolerance) and avoid the work of solving subproblems whose objective can be no better than this value. If you enter a value for this parameter, you must be *sure* that there is an integer solution with an objective value at least this good: A value that is too large (for maximization problems) or too small (for minimization) may cause the SDK to skip solving the subproblem that would yield the optimal integer solution.

Preprocessing

Name	Preprocessing
Default	1
Min	1
Max	3
Type	Integer

1 – Automatic, 2 – None, 3 - Aggressive

When this parameter is set, the Solver performs preprocessing on constraints involving integer variables, to simplify the problem and speed up the solution process. Based on the settings of certain binary integer variables, preprocessing can fix the values of other binary integer variables, tighten the bounds on continuous variables, and in some cases, determine that the subproblem is infeasible, so it is unnecessary to solve it at all.

When the Preprocessing parameter is set to 1, the LP/Quadratic engine will automatically determine the best preprocessing features to use on the problem. If the parameter is set to 2, then no preprocessing will be performed and if the parameter is set to 3, then the LP/Quadratic Solver will apply the most aggressive forms of preprocessing.

Cuts

Name	Cuts
Default	1
Min	1
Max	3
Type	Integer

1 – Automatic, 2 – None, 3 - Aggressive

The LP/Quadratic Solver in the Solver SDK Platform supports a wide range of cuts. A *cut* is an automatically generated linear constraint for the problem, in addition to the constraints that you specify. This constraint is constructed so that it "cuts off" some portion of the feasible region of an LP subproblem, without eliminating any possible integer solutions. Cuts require more work on each subproblem, but they can often lead more quickly to integer solutions and greatly reduce the number of subproblems that must be explored.

The LP/Quadratic engine employs a wide range of cuts including, Clique, Flow Cover, Gomory, Knapsack, Local Tree, Mixed Integer Rounding, Probing, Two Mixed Integer Rounding, Reduce and Split, and Special Ordered Sets.

When the Cuts parameter is set to 1, the LP/Quadratic engine will automatically determine the best cuts to use on the problem. If the parameter is set to 2, then no cuts will be performed and if the parameter is set to 3, then the LP/Quadratic Solver will apply the most aggressive forms of cuts.

Heuristics

Name	Heuristics
Default	1
Min	1
Max	3
Type	Integer

1 – Automatic, 2 – None, 3 – Aggressive

A *heuristic* is a strategy that often – but not always – will find a reasonably good "incumbent" or feasible integer solution early in the search. Heuristics require more work on each subproblem, but they can often lead more quickly to integer solutions and greatly reduce the number of subproblems that must be explored. The LP/Quadratic engine employs several heuristics including: Feasibility Pump, Greedy Cover, Local Search, and Rounding.

When the Heuristics parameter is set to 1, the LP/Quadratic engine will automatically determine the best heuristics to apply to the problem. If the parameter is set to 2, then no heuristics will be used and if the parameter is set to 3, then the LP/Quadratic Solver will use heuristics extensively in trying to find a good initial incumbent.

Thread Mode

Name	Thread Mode
0	Use branching, NumThreads at a time (default).
1	Use deterministic, NumThreads at a time.
2	Use root cuts, NumThreads at a time.
8	Use heuristics, NumThreads at a time.
9	No threads, use one thread at a time.

Determines how the threads are used when a mixed integer model is being solved. If N equals the "Number of Threads", the collection of nodes will be solved using the selected mode, N threads at a time.

LP/Quadratic Solver Stochastic Decomposition Parameters

This section describes the parameters which control the Stochastic Decomposition methods bundled within the Solver SDK Platform and utilized in the LP/Quadratic engine. This engine contains an extensive set of options to improve performance on problems that contain integers. Stochastic Decomposition can be used to solve linear models with recourse variables and uncertainty in the constraints only. (Model must contain at least one constraint that does not include uncertainty.) Typically, Stochastic Decomposition should only be used if the internal model created when Deterministic Equivalent is selected for Solve Uncertain Models is so large that the time and memory required to solve the model is impractical.

Tau

Name	StochTau
Default	2
Min	1

Max	2,147,483,647
Type	Integer

As iterations proceed, the cut which was formed based on the incumbent vector is periodically re-evaluated. If tau iterations have passed since the last update or if the value of the incumbent is less than the objective value at a specific iteration, the incumbent is reformed. This function will solve an additional subproblem (based upon the incumbent and the most recent observation of omega) add the dual solution to the data structures, and re-form the incumbent cut (thus replacing it in the array of cuts).

Tolerance

Name	StochTol
Default	0.001
Min	0
Max	1
Type	Double

The tolerance setting is used to determine the stopping condition for the Stochastic Decomposition method. A larger value will result in the Stochastic Decomposition methods stopping more quickly, while a smaller value will force the Stochastic Decomposition methods to run longer.

Compute Confidence Interval

Name	StochCI
Default	0
Min	0
Max	1
Type	Binary

If this option is turned on, the stochastic decomposition methods will compute a 95% confidence interval for the objective function.

Objective Error

Name	StochErr
Default	0.01
Min	0
Max	1
Type	Double

If Compute Confidence Interval is turned on, then the confidence interval for the objective function will be accurate to within this value, with 95% confidence.

Objective Improvement

Name	StochR
Default	0.2
Min	0
Max	1

Type	Double

This parameter sets the minimum amount of improvement which must be observed in order to update the incumbent.

Compute Recourse Statistics

Name	StochR
Default	1
Min	0
Max	1
Type	Binary

If set to 0, recourse variables for each trial will not be computed. Only “normal” decision variable values will be available after Solver has found a solution. If set to 1, recourse variables will be computed for each trial which can be viewed through the Variables.AllTrials property.

SOCP Barrier Solver Parameters

In the Solver SDK Platform, if the SOCP Barrier Solver is selected as the engine to solve the problem, all parameters described above in the Common Parameters section as well as the parameters described below (Precision, Gap Tolerance, Step Size Factor, Feasibility Tolerance, and the Search Direction option group) will be available to the User.

Gap Tolerance

Name	GapTolerance
Default	1.0e-6
Min	0
Max	1.0
Type	Double

The SOCP Barrier Solver uses a primal-dual method that computes new objective values for the primal problem and the dual problem at each iteration. When the gap or difference between these two objective values is less than the Gap Tolerance, the SOCP Barrier Solver will stop and declare the current solution optimal.

Step Size Factor

Name	StepSizeFactor
Default	0.99
Min	0
Max	1.0
Type	Double

This parameter is the relative size (between 0 and 1) of the step that the SOCP Barrier Solver may take towards the constraint boundary at each iteration.

Feasibility Tolerance

Name	FeasibilityTolerance
Default	1.0e-6

Min	0
Max	1.0
Type	Double

The SOCP Barrier Solver considers a solution feasible if the constraints are satisfied to within this tolerance.

Search Direction

Name	SearchDirection
Default	3
Min	0
Max	3
Type	Integer

The SOCP Barrier Solver offers four options for computing the search direction on each iteration: **0-Power Class**, **1-Power Class with Predictor-Corrector**, **2-Dual Scaling**, and **3- Dual Scaling with Predictor-Corrector**.

Power Class

This option uses the *power class*, which is a subclass of the commutative class of search directions over symmetric cones with the property that the long-step barrier algorithm using this class has polynomial complexity.

Power Class with Predictor-Corrector

This option uses the *power class* as described above, plus a predictor-corrector term.

Dual Scaling

This option uses HKM (Helmberg, Kojima and Monteiro) dual scaling, a Newton direction found from the linearization of a symmetrized version of the optimality conditions.

Dual Scaling with Predictor-Corrector

This option uses HKM dual scaling, plus a predictor-corrector term.

Power Index

Name	PowerIndex
Default	1
Min	0
Max	2,147,483,647
Type	Integer

This parameter is used to select a specific search direction when the Search Direction is computed via the Power Class or Power Class with Predictor-Corrector methods.

SOCP Integer Options

Max Subproblems

Name	MaxSubproblems

Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Subproblems parameter places a limit on the number of subproblems that may be explored by the Branch & Bound algorithm before the SDK stops with the status result, *Optimize_Status_Bpmips_Limit*. Each subproblem is a "regular" Solver problem with additional bounds on the variables. In a problem with integer constraints, the Max Subproblems limit should be used in preference to the Iterations limit.

Max Feasible (Integer) Solutions

Name	MaxFeasibleSols MaxIntegerSols
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Feasible Sols parameter places a limit on the number of feasible integer solutions found by the Branch & Bound algorithm before the SDK stops with the status result, *Optimize_Status_Bpnode_Limit*. Each feasible integer solution satisfies all of the constraints, including the integer constraints; the Solver retains the integer solution with the best objective value so far, called the "incumbent."

It is entirely possible that, in the process of exploring various subproblems with different bounds on the variables, the Branch & Bound algorithm may find the same feasible integer solution (set of values for the decision variables) more than once; the Max Feasible Solutions limit applies to the total number of integer solutions found, not the number of "distinct" integer solutions.

Integer Tolerance

Name	IntTolerance
Default	0
Min	0
Max	1.0
Type	Double

When you solve an integer programming problem, it often happens that the Branch & Bound method will find a good solution fairly quickly, but will require a great deal of computing time to find (or verify that it has found) the optimal integer solution. The Integer Tolerance setting may be used to tell the SDK to stop if the best solution it has found so far is "close enough."

The Branch & Bound process starts by finding the optimal solution without considering the integer constraints (this is called the *relaxation* of the integer programming problem). The objective value of the relaxation forms the initial "best bound" on the objective of the optimal *integer* solution, which can be no better than this. During the optimization process, the Branch & Bound method finds "candidate" integer solutions, and it keeps the best solution so far as the "incumbent." By eliminating alternatives as its proceeds, the B&B method also tightens the "best bound" on how good the integer solution can be.

Each time the Solver finds a new incumbent – an improved all-integer solution – it computes the maximum percentage difference between the objective of this solution and the current best bound on the objective:

$$\frac{\text{Objective of incumbent} - \text{Objective of best bound}}{\text{Objective of best bound}}$$

If the absolute value of this maximum percentage difference is equal to or less than the Integer Tolerance, the Solver will stop and report the current integer solution as the optimal result. In the Solver SDK, the status result will be *Optimize_Status_Incumb_Cand*. If you set the Integer Tolerance to zero, the Solver will continue searching until all alternatives have been explored and the optimal integer solution has been found. This could take a great deal of computing time.

Integer Cutoff

Name	IntCutoff
Default	2e+30
Min	-2e-30
Max	2e+30
Type	Double

This option provides another way to save time in the solution of mixed-integer programming problems. If you know the objective value of a feasible integer solution to your problem – possibly from a previous run of the same or a very similar problem – you can enter this objective value for the Integer Cutoff parameter. This allows the Branch & Bound process to *start* with an "incumbent" objective value (as discussed above under Integer Tolerance) and avoid the work of solving subproblems whose objective can be no better than this value. If you enter a value for this parameter, you must be *sure* that there is an integer solution with an objective value at least this good: A value that is too large (for maximization problems) or too small (for minimization) may cause the SDK to skip solving the subproblem that would yield the optimal integer solution.

In the Solver SDK Pro, if the GRG Nonlinear Solver is selected as the engine to solve the problem the Standard GRG Solver Engine will be used. All parameters described above in the Common Parameters section as well as the parameters described in this section will be available to the User. The Global Optimization options group and the Population Size and Random Seed parameters are described in the next section, "Multistart Search Options."

The default choices for these options are suitable for the vast majority of problems; although it generally won't *hurt* to change these options, you should first consider other alternatives such as improved scaling before attempting to fine-tune them. In some scientific and engineering applications, alternative choices may improve the solution process.

Convergence

Name	Convergence
Default	1.0e-4
Min	0
Max	1
Type	Double

The Standard GRG Solver will stop with the status *Optimize_Status_Converged* when the objective function value is changing very slowly for the last few iterations or trial solutions. More precisely, the GRG Solver stops if the absolute value of the *relative* change in the objective function is less than the value in the Convergence edit box for the last 5 iterations. While the default value of 1.0E-4 (0.0001) is suitable for most problems, it may be too large for some models, causing the GRG Solver to stop

prematurely when this test is satisfied, instead of continuing for more Trial Solutions until the optimality (KKT) conditions are satisfied.

If you are getting this message when you are seeking a locally optimal solution, you can change the parameter setting to a smaller value such as 1.0E-5 (0.00001) or 1.0E-6 (0.000001); but you should also consider why it is that the objective function is changing so slowly. Perhaps you can add constraints or use different starting values for the variables, so that the Solver does not get "trapped" in a region of slow improvement.

Recognize Linear Variables

Name	RecognizeLinear
Default	0 – Do Not Recognize
Min	0
Max	1 – Recognize
Type	Integer

In a nonlinear problem, some of the variables occur linearly in the objective and all of the constraints. Hence the partial derivatives of the problem functions with respect to these variables are constant, and need not be re-computed on each iteration. If you set the parameter Recognize Linear Variables to 1, the Standard GRG Solver will look for variables whose partial derivatives are not changing over several iterations, and then will *assume* that these variables occur linearly, hence that their partial derivatives remain constant. At the solution, the partial derivatives are recomputed and compared to the assumed constant values; if any of these values has changed, the Solver will stop with the status result *Optimize Status Linear Invalid*. If the SDK Pro stops with this status, you should set the Recognize Linear Variables parameter to 0 and re-solve the problem.

Derivatives and Other Nonlinear Options

The default values for the Estimates, Derivatives and Search options can be used for most problems. If you'd like to change these options to improve performance on your model, this section will provide some general background on how they are used by the Standard GRG Solver.

On each major iteration, the GRG Solver requires values for the gradients of the objective and constraints (i.e. the Jacobian matrix). The Derivatives option is concerned with how these partial derivatives are computed.

The GRG (Generalized Reduced Gradient) solution algorithm proceeds by first "reducing" the problem to an unconstrained optimization problem, by solving a set of nonlinear equations for certain variables (the "basic" variables) in terms of others (the "nonbasic" variables). Then a search direction (a vector in n -space, where n is the number of nonbasic variables) is chosen along which an improvement in the objective function will be sought. The Search option is concerned with how this search direction is determined.

Once a search direction is chosen, a one-dimensional "line search" is carried out along that direction, varying a step size, in an effort to improve the reduced objective. The StepSize parameter controls the size of this step. The initial estimates for values of the variables that are being varied have a significant impact on the effectiveness of the search. The Estimates option is concerned with how these estimates are obtained.

Estimates

Name	Estimates
Default	0 – Tangent
Min	0

Max	1 – Quadratic
Type	Integer

This option determines the approach used to obtain initial estimates of the basic variable values at the outset of each one-dimensional search. The Tangent choice uses linear extrapolation from the line tangent to the reduced objective function. The Quadratic choice extrapolates the minimum (or maximum) of a quadratic fitted to the function at its current point. If the current reduced objective is well modeled by a quadratic, then the Quadratic option can save time by choosing a better initial point, which requires fewer subsequent steps in each line search. If you have no special information about the behavior of this function, the Tangent choice is "slower but surer." **Note:** the Quadratic choice here has no bearing on quadratic programming problems.

Derivatives

Name	Derivatives
Default	3
Min	1
Max	4
Type	Integer

1 – Forward, 2 –Central, 3 – Jacobian, 4 – Jacobian with Error Checking

On each major iteration, the GRG Solver requires values for the gradients of the objective and constraints (i.e. the Jacobian matrix). In the Solver SDK, the method used for finite differencing is determined by the Derivatives setting.

Forward differencing (the default choice) uses the point from the previous iteration – where the problem function values are already known – in conjunction with the current point. *Central* differencing relies only on the current point, and perturbs the decision variables in opposite directions from that point. This requires up to twice as much time *on each iteration*, but it may result in a better choice of search direction when the derivatives are rapidly changing, and hence fewer total iterations.

Setting this parameter to 3 tells the SDK that a Jacobian evaluator is being supplied to compute partial derivatives and to call your Jacobian evaluator in place of your function evaluator. A parameter setting of 4 signals to the SDK to call your function evaluator to compute partial derivatives via forward differencing, then call your Jacobian evaluator and compare the results. If the derivatives do not match within a small tolerance, the SDK will call the evaluator, *Eval_Type_GradError* to display the User's error messages.

Search Option

Name	SearchOption
Default	0 – Newton
Min	0
Max	1 – Conjugate
Type	Integer

It would be expensive to determine a search direction using the pure form of Newton's method, by computing the Hessian matrix of *second* partial derivatives of the problem functions. Instead, a direction is chosen through an estimation method. The default choice, Newton, uses a quasi-Newton (or BFGS) method, which maintains an *approximation* to the Hessian matrix; this requires more storage (an amount proportional to the square of the number of currently binding constraints) but performs very well in practice. The alternative choice, Conjugate, uses a conjugate gradient method, which does not require storage for the Hessian matrix and still performs well in most cases. The

choice you make here is not crucial, since the GRG Solver is capable of switching *automatically* between the quasi-Newton and conjugate gradient methods depending on the available storage.

Step Size

Name	StepSize
Default	1.0e-6
Min	1.0e-9
Max	1.0e-4
Type	Double

This parameter determines the "delta" or amount of change used in computing numerical derivatives via finite differencing. Changing the step size to a very small value could result in the GRG engine stopping prematurely. However, a large step size could result in Solver "stepping over" a good solution. **For the vast majority of models, this parameter should be left at the default value.**

Large Scale GRG Nonlinear Solver Parameters

In the Solver SDK Platform, if the GRG Nonlinear Solver is selected as the engine to solve the problem the Large Scale GRG Solver Engine will be used. All parameters described above in the Common Parameters section as well as the parameters described in this section will be available to the User. The Global Optimization options group and the Population Size and Random Seed parameters are described in the next section, "Multistart Search Options."

The default choices for these options are suitable for the vast majority of problems; although it generally won't *hurt* to change these options, you should first consider other alternatives such as improved scaling before attempting to fine-tune them. In some scientific and engineering applications, alternative choices may improve the solution process.

Convergence

Name	Convergence
Default	0.0001
Min	0
Max	1
Type	Double

The LSGRG Solver will stop with the status *Optimize Status Converged* when the objective function value is changing very slowly for the last few iterations or trial solutions. More precisely, the LSGRG Solver stops if the absolute value of the *relative* change in the objective function is less than the value in the Convergence edit box for the last 5 iterations. While the default value of 1.0E-4 (0.0001) is suitable for most problems, it may be too large for some models, causing the LSGRG Solver to stop prematurely when this test is satisfied, instead of continuing for more Trial Solutions until the optimality (KKT) conditions are satisfied.

If you are getting this message when you are seeking a locally optimal solution, you can change the parameter setting to a smaller value such as 1.0E-5 (0.00001) or 1.0E-6 (0.000001); but you should also consider why it is that the objective function is changing so slowly. Perhaps you can add constraints or use different starting values for the variables, so that the Solver does not get "trapped" in a region of slow improvement.

Recognize Linear Variables

Name	RecognizeLinear

Default	0 – Do Not Recognize
Min	0
Max	1 – Recognize
Type	Integer

This option is included for compatibility with the nonlinear GRG Solver in the Solver SDK Pro product, but it has only a limited effect in the LSGRG. It is intended to take advantage of the fact that in many nonlinear problems, some of the variables occur linearly in the objective and all of the constraints. Hence the partial derivatives of the problem functions with respect to these variables are constant, and need not be re-computed on each iteration.

Relax Bounds on Variables

Name	RelaxBounds
Default	0 – Do Not Relax
Min	0
Max	1 – Relax
Type	Integer

By default (and unlike the nonlinear GRG Solver bundled within the Solver SDK Pro), the LSGRG Solver ensures that any trial points evaluated during the solution process will not have values that violate the bounds on the variables you specify, even by a small amount. If your problem functions cannot be evaluated for values outside the variable bounds, this default behavior will ensure that the solution process can continue. However, at times, the LSGRG Solver can make more rapid progress along a given search direction by testing trial points with values slightly outside the bounds on the variables. If you want to permit this to happen, set this option to 1. If you receive the Solver Result, Optimize_Status_Float_Err, as a first step you should set this option back to the default, 0.

Derivatives and Other Nonlinear Options

The default values for the Estimates, Derivatives and Search options can be used for most problems. If you'd like to change these options to improve performance on your model, this section will provide some general background on how they are used by the LSGRG Solver.

On each major iteration, the LSGRG Solver requires values for the gradients of the objective and constraints (i.e. the Jacobian matrix). The Derivatives option is concerned with how these partial derivatives are computed.

The LSGRG (Generalized Reduced Gradient) solution algorithm proceeds by first "reducing" the problem to an unconstrained optimization problem, by solving a set of nonlinear equations for certain variables (the "basic" variables) in terms of others (the "nonbasic" variables). Then a search direction (a vector in n -space, where n is the number of nonbasic variables) is chosen along which an improvement in the objective function will be sought. The Search option is concerned with how this search direction is determined.

Once a search direction is chosen, a one-dimensional "line search" is carried out along that direction, varying a step size in an effort to improve the reduced objective. The Step Size parameter controls the size of this step. The initial estimates for values of the variables that are being varied have a significant impact on the effectiveness of the search. The Estimates option is concerned with how these estimates are obtained.

Estimates

Name	Estimates

Default	0 – Tangent
Min	0
Max	1 – Quadratic
Type	Integer

This option determines the approach used to obtain initial estimates of the basic variable values at the outset of each one-dimensional search. The Tangent choice uses linear extrapolation from the line tangent to the reduced objective function. The Quadratic choice extrapolates the minimum (or maximum) of a quadratic fitted to the function at its current point. If the current reduced objective is well modeled by a quadratic, then the Quadratic option can save time by choosing a better initial point, which requires fewer subsequent steps in each line search. If you have no special information about the behavior of this function, the Tangent choice is "slower but surer." **Note:** the Quadratic choice here has no bearing on quadratic programming problems.

Derivatives

Name	Derivatives
Default	1
Min	1
Max	2
Type	Integer

1 – Forward, 2 –Central

On each major iteration, the LSGRG Solver requires values for the gradients of the objective and constraints (i.e. the Jacobian matrix). In the Large Scale GRG Engine, the method used for finite differencing is determined by the Derivatives setting.

Forward differencing (the default choice) uses the point from the previous iteration – where the problem function values are already known – in conjunction with the current point. *Central* differencing relies only on the current point, and perturbs the decision variables in opposite directions from that point. This requires up to twice as much time *on each iteration*, but it may result in a better choice of search direction when the derivatives are rapidly changing, and hence fewer total iterations.

Supplying Partial Derivatives: Partial derivatives can be supplied via the `Eval_Type_Gradient` evaluator as described in the “Using a Gradient Evaluator to Provide First order Partial Derivatives” section within the **Solver SDK Evaluators** chapter. To supply partial derivatives, both `Eval_Type_Function` and `Eval_Type_Gradients` evaluators must be present in the model. The existence of these two evaluators signals to the SDK that a Jacobian evaluator is being supplied to compute partial derivatives and to call your Jacobian evaluator in place of your function evaluator.

Checking Accuracy of Partial Derivatives: If a third evaluator is present, `Eval_Type_GradErr`, Solver SDK will call your function evaluator to compute partial derivatives via forward differencing, then call your Jacobian evaluator and compare the results. If the derivatives do not match within a small tolerance, the SDK will call the evaluator, `Eval_Type_GradError` to display the User’s error message(s). In order for this feature to be invoked, your model must call three evaluators: `Eval_Type_Function`, `Eval_Type_Gradient` and `Eval_Type_GradError`.

Search Option

Name	SearchOption
Default	0 – Newton
Min	0
Max	1 – Conjugate

Type	Integer

It would be expensive to determine a search direction using the pure form of Newton's method, by computing the Hessian matrix of *second* partial derivatives of the problem functions. Instead, a direction is chosen through an estimation method. The default choice, Newton, uses a quasi-Newton (or BFGS) method, which maintains an *approximation* to the Hessian matrix; this requires more storage (an amount proportional to the square of the number of currently binding constraints) but performs very well in practice. The alternative choice, Conjugate, uses a conjugate gradient method, which does not require storage for the Hessian matrix and still performs well in most cases. The choice you make here is not crucial, since the LSGRG solver is capable of switching *automatically* between the quasi-Newton and conjugate gradient methods depending on the available storage.

Step Size

Name	StepSize
Default	1.0e-6
Min	1.0e-9
Max	1.0e-4
Type	Double

This parameter determines the "delta" or amount of change used in computing numerical derivatives via finite differencing. Changing the step size to a very small value could result in the GRG engine stopping prematurely. However, a large step size could result in Solver "stepping over" a good solution. **For the vast majority of models, this parameter should be left at the default value.**

Multistart Search Parameters

This section discusses the Global Optimization options group and the Population Size and Random Seed parameters that are used by the Large Scale GRG Solver Engine within the Solver SDK Platform and the Standard GRG Solver Engine within the Solver SDK Pro.

These parameters control the multistart methods for global optimization, which will automatically run the GRG Solver (or certain field-installable Solver engines) from a number of starting points in order to seek the globally optimal solution.

Multistart Search

Name	Multistart
Default	0 – Off
Min	0
Max	1 – On
Type	Integer

If this parameter is set to 1, the multistart methods are used to seek a globally optimal solution. If this parameter is set to 0, the other options described in this section are ignored. The multistart methods will generate candidate starting points for the GRG Solver (with randomly selected values between the bounds you specify for the variables), group them into "clusters" using a method called multi-level single linkage, and then run the GRG Solver from a representative point in each cluster. This process continues with successively smaller clusters that are increasingly likely to capture each possible locally optimal solution.

Topographic Search

Name	TopoSearch
Default	0 – Off
Min	0
Max	1 – On
Type	Integer

If this parameter (and the Multistart parameter) are both set to 1, the multistart methods will make use of a "topographic" search method. This method uses the objective value computed for the randomly sampled starting points to compute a "topography" of overall "hills" and "valleys" in the search space, in an effort to find better clusters and start the GRG Solver from an improved point (already in a "hill" or "valley") in each cluster. Computing the topography takes extra time, but on some problems this is more than offset by reduced time taken by the GRG Solver on each subproblem.

Require Bounds on Variables

Name	RequireBounds
Default	1 – On
Min	0 - Off
Max	1
Type	Integer

This parameter is turned on, set to 1, by default, but it comes into play only when the Multistart Search box is checked. The multistart methods generate candidate starting points for the GRG Solver by randomly sampling values between the bounds on the variables that you specify. If you do not specify both upper and lower bounds on each of the decision variables, the multistart methods can still be used, but because the random sample must be drawn from an "infinite" range of values, this is unlikely to effectively cover the possible starting points (and therefore have a good chance of finding all of the locally optimal solutions), unless the GRG Solver is run on a great many subproblems, which will take a very long time.

The tighter the bounds on the variables that you can specify, the better the multistart methods are likely to perform. (This is also true of the Evolutionary Solver.) Hence, this option is turned on by default, so that you will be automatically reminded to include both upper and lower bounds on all of the variables whenever you select Multistart Search. If both the Multistart Search and Require Bounds on Variables parameters are both set to 1, but you have not defined upper and lower bounds on all of the variables, you will receive the Status result, *Optimize_Status_Bounds_Missing*.

If you get this result, you must either add both upper and lower variable bounds (or call *prob.VarDecision.NonNegative()* to add lower bounds on the variables), or else set the Require Bounds parameter to 0 and resolve.

Population Size

Name	PopulationSize
Default	0
Min	0
Max	200
Type	Integer

The multistart methods generate a number of candidate starting points for the GRG Solver equal to the value that you enter for the parameter. This set of starting points is referred to as a "population,"

because it plays a role somewhat similar to the population of candidate solutions maintained by the Evolutionary Solver. The minimum population size is 10 points; if you supply a value less than 10, or do not pass the parameter at all, the multistart methods use a population size of 10 times the number of decision variables in the problem, but no more than 200.

Random Seed

Name	RandomSeed
Default	0
Min	0
Max	2,147,483,647
Type	Integer

The multistart methods use a process of random sampling to generate candidate starting points for the GRG Solver. This process uses a random number generator that is normally "seeded" using the value of the system clock – so the random number sequence (and hence the generated candidate starting points) will be different each time you click Solve. At times, however, you may wish to ensure that the *same* candidate starting points are generated on several successive runs – for example, in order to test different GRG Solver options on each search for a locally optimal solution. To do this, enter an integer value into this box; this value will then be used to "seed" the random number generator each time you click Solve.

Evolutionary Solver Parameters

If the Evolutionary Solver is selected as the engine to solve the problem, all parameters described above in the Common Parameters section as well as the parameters described in this section will be available to the User.

As with the other Solver engines, the Max Time option determines the maximum amount of time the Evolutionary Solver will run before stopping with the status result, *Optimize_Status_Time_Out*. The Iterations option rarely comes into play, because the Evolutionary Solver always uses the Max Subproblems and Max Feasible Solutions parameters, whether or not the problem includes integer constraints. (The count of iterations is reset on each new subproblem, so the Iterations limit normally is not reached.) The Precision option plays the same role as it does in the other Solver engines – governing how close a constraint value must be to its bound to be considered satisfied, and how close to an exact integer value a variable must be to satisfy an integer constraint. It also is used in computing the "penalty" applied to infeasible solutions that are accepted into the population: A smaller Precision value increases this penalty.

Convergence

Name	Convergence
Default	0.0001
Min	0
Max	1
Type	Double

The Evolutionary Solver will stop with the Status result, *Optimize_Status_Converged*, if nearly all members of the current population of solutions have very similar "fitness" values. Since the population may include members representing infeasible solutions, each "fitness" value is a combination of an objective function value and a penalty for infeasibility. Since the population is initialized with trial solutions that are largely chosen at random, the comparison begins after the Solver has found a certain minimum number of improved solutions that were generated by the evolutionary

process. The stopping condition is satisfied if 99% of the population members all have fitness values that are within the Convergence tolerance of each other.

If you believe that the Solver SDK is stopping prematurely with the status *Optimize_Status_Converged*, you can make the Convergence tolerance smaller, but you may also want to increase the Mutation Rate and/or the Population Size, in order to increase the diversity of the population of trial solutions.

Population Size

Name	PopulationSize
Default	0
Min	0
Max	200
Type	Integer

The Evolutionary Solver maintains a population of candidate solutions, rather than a "single best solution" so far, throughout the solution process. This option sets the number of candidate solutions in the population. The minimum population size is 10 members; if you supply a value less than 10 for this option, or not pass anything for the population, the Evolutionary Solver uses a population size of 10 times the number of decision variables in the problem, but no more than 200.

The initial population consists of candidate solutions chosen largely at random, but it always includes at least one instance of the starting values of the variables (adjusted if necessary to satisfy the bounds on the variables), and it may include more than one instance of the starting values, especially if the population is large and the initial values represent a feasible solution.

A larger population size may allow for a more complete exploration of the "search space" of possible solutions, especially if the rate is high enough to create diversity in the population. However, experience with genetic and evolutionary algorithms reported in the research literature suggests that a population need not be very large to be effective – many successful applications have used a population of 70 to 100 members.

Mutation Rate

Name	MutationRate
Default	0.075
Min	0
Max	1.0
Type	Double

The Mutation Rate is the probability that some member of the population will be mutated to create a new trial solution (which becomes a candidate for inclusion in the population, depending on its fitness) during each "generation" or subproblem considered by the evolutionary algorithm. In the Evolutionary Solver, a subproblem consists of a possible mutation step, a crossover step, an optional local search in the vicinity of a newly discovered "best" solution, and a selection step where a relatively "unfit" member of the population is eliminated.

There are many possible ways to mutate a member of the population, and the Evolutionary Solver actually employs five different mutation strategies, including "permutation-preserving" mutation strategies for variables that are members of an "alldifferent" group. The Mutation Rate is effectively subdivided between these strategies, so increasing or decreasing the Mutation Rate affects the probability that each of the strategies will be used during a given "generation" or subproblem.

Random Seed

Name	RandomSeed
Default	0
Min	0
Max	2,147,483,647
Type	Integer

The Evolutionary Solver makes extensive use of random sampling, to generate trial points for the population of candidate solutions, to choose strategies for mutation and crossover on each "generation," and for many other purposes. This process uses a random number generator that is normally "seeded" using the value of the system clock – so the random number sequence (and hence trial points and choices made by the Evolutionary Solver) will be different each time you click Solve. Because of these random choices, the Evolutionary Solver will normally find at least slightly different (and sometimes very different) solutions on each run, even if you haven't changed your model at all. At times, however, you may wish to ensure that exactly the *same* trial points are generated, and the same choices are made on several successive runs. To do this, pass a positive integer value for this parameter; this value will then be used to "seed" the random number generator each time you solve.

Require Bounds on Variables

Name	RequireBounds
Default	1 – On
Min	0 – Off
Max	1
Type	Integer

If the parameter "Require Bounds on Variables" is set to 1, and some of the decision variables do not have upper or lower bounds specified at the time you run the SDK, the Solver will stop immediately with the Status result, *Optimize_Status_Bounds_Missing*. If this parameter is set to 0, the Solver will not require upper and lower bounds on the variables, but will attempt to solve the problem without them. Note that this parameter is turned on by *default*.

Bounds on the variables are especially important to the performance of the Evolutionary Solver. For example, the initial population of candidate solutions is created, in part, by selecting values at random from the ranges determined by each variable's lower and upper bounds. Bounds on the variables are also used in the mutation process – where a change is made to a variable value in some member of the existing population – and in several other ways in the Evolutionary Solver. If you do not specify lower and upper bounds for all of the variables in your problem, the Evolutionary Solver can still proceed, but the almost-infinite range for these variables may significantly slow down the solution process, and make it much harder to find "good" solutions. Hence, it pays for you to determine realistic lower and upper bounds for the variables.

Local Search

Name	LocalSearch
Default	3
Min	0
Max	3
Type	Integer

0-Randomized Local Search, 1- Deterministic Pattern Search, 2-Gradient Local Search, 3-Automatic Choice

This option determines the local search strategy employed by the Evolutionary Solver. As noted under the Mutation rate option, a "generation" or subproblem in the Evolutionary Solver consists of a possible mutation step, a crossover step, an optional local search in the vicinity of a newly discovered "best" solution, and a selection step where a relatively "unfit" member of the population is eliminated. You have a choice of strategies for the local search step. In the Solver SDK, you can use Automatic Choice (the default), which selects an appropriate local search strategy automatically based on characteristics of the problem functions.

Randomized Local Search

This local search strategy generates a small number of new trial points in the vicinity of the just-discovered "best" solution, using a probability distribution for each variable whose parameters are a function of the best and worst members of the current population. (If the generated points do not satisfy all of the constraints, a variety of strategies may be employed to transform them into feasible solutions.) Improved points are accepted into the population.

Deterministic Pattern Search

This local search strategy uses a "pattern search" method to seek improved points in the vicinity of the just-discovered "best" solution. The pattern search method is deterministic – it does not make use of random sampling or choices – but it also does not rely on gradient information, so it is effective for non-smooth functions. It uses a "slow progress" test to decide when to halt the local search. An improved point, if found, is accepted into the population.

Gradient Local Search

This local search strategy makes the assumption that the objective function – even if non-smooth – can be approximated locally by a quadratic model. It uses a classical quasi-Newton method to seek improved points, starting from the just-discovered "best" solution and moving in the direction of the gradient of the objective function. It uses a classical optimality test and a "slow progress" test to decide when to halt the local search. An improved point, if found, is accepted into the population.

Automatic Choice

This option allows the Solver to select the local search strategy automatically in the Solver SDK. The Solver uses diagnostic information from the Polymorphic Spreadsheet Interpreter to select a linear Gradient Local Search strategy if the problem has a mix of non-smooth and linear variables, or a nonlinear Gradient Local Search strategy if the objective function has a mix of non-smooth and smooth nonlinear variables. It also makes limited use of the Randomized Local Search strategy to increase diversity of the points found by the local search step.

Fix Nonsmooth Variables

Name	FixNonsmooth
Default	0 – Off
Min	0
Max	1 – On
Type	Integer

This parameter determines how non-smooth variable occurrences in the problem will be handled during the local search step. If this parameter is set to 1, the non-smooth variables are fixed to their current values (determined by genetic algorithm methods) when a nonlinear Local Gradient or linear Local Gradient search is performed; only the smooth and linear variables are allowed to vary. If this parameter is set to 0, all of the variables are allowed to vary.

Since gradients are undefined for non-smooth variables at certain points, fixing these variables ensures that gradient values used in the local search process will be valid. On the other hand, gradients *are* defined for non-smooth variables at *most* points, and the search methods are often able to proceed in spite of *some* invalid gradient values, so it often makes sense to vary all of the variables during the search. Hence, this parameter is set to 0 by default; you can experiment with its setting on your model.

Filtered Local Search

The Solver SDK applies two tests or "filters" to determine whether to perform a local search each time a new point generated by the genetic algorithm methods is accepted into the population. The "merit filter" requires that the objective value of the new point be better than a certain threshold if it is to be used as a starting point for a local search; the threshold is based on the best objective value found so far, but is adjusted dynamically as the Solver proceeds. The "distance filter" requires that the new point's distance from any known locally optimal point (found on a previous local search) be greater than the distance traveled when that locally optimal point was found.

Thanks to its genetic algorithm methods, improved local search methods, and the distance and merit filters, the Evolutionary Solver in the Solver SDK performs exceedingly well on smooth global optimization problems, and on many non-smooth problems as well.

The local search methods range from relatively "cheap" to "expensive" in terms of the computing time expended in the local search step; they are listed roughly in order of the computational effort they require. On some problems, the extra computational effort will "pay off" in terms of improved solutions, but in other problems, you will be better off using the "cheap" Randomized Local Search method, thereby spending relatively more time on the "global search" carried out by the Evolutionary Solver's mutation and crossover operations.

In addition to the Local Search options, the Evolutionary Solver employs a set of methods, corresponding to the four local search methods, to transform infeasible solutions – generated through mutation and crossover – into feasible solutions in new regions of the search space. These methods, which also vary from "cheap" to "expensive," are selected dynamically (and automatically) via a set of heuristics. For problems in which a significant number of constraints are smooth nonlinear or even linear, these methods can be highly effective. Dealing with constraints is traditionally a weak point of genetic and evolutionary algorithms, but the hybrid Evolutionary Solver in the Solver SDK is unusually strong in its ability to deal with a combination of constraints and non-smooth functions.

Global Search

Name	GlobalSearch
Default	0 – Genetic Algorithm
Min	0
Max	1 – Scatter Search
Type	Binary

If this option is set to 1 (Genetic Algorithm), then the Evolutionary Solver will use methods from the literature on genetic algorithms (its traditional methods) to solve the model. Otherwise, the Evolutionary Solver will use methods from the literature on scatter search. On some models, the scatter search algorithm will result in better answers in less time when compared to the genetic algorithm. However, for other models, the genetic algorithm may be more successful. Since the scatter search algorithm tends to perform best, by a modest margin, on the majority of models, it is the default choice. But we suggest you try both algorithms with your model to see which works better for you.

Model Based Search

Name	ModelBasedSearch
Default	0 – None

Min	0
Max	2
Type	Integer

0-None, 1- CPU Based, 2-GPU Based

This option takes effect only when the Legacy Mode option is set to False. When this option is set to “None”, the new Scatter Search algorithm is used without any Model Based Search. When this option is set to either “CPU Based” or “GPU Based”, an internal model of the problem is created (using the *Radial Basis Functions* method) which closely fits the original problem. The Evolutionary Engine uses this internal model to evaluate many points in parallel (either on the CPU or GPU - depending on the option setting) rather than calling the Interpreter to evaluate each of these points sequentially. Only the most promising of these points are sent to the Interpreter for actual evaluation using the new Scatter Search Algorithm. This new search method typically results in better solutions in less time when compared to using only the Scatter Search algorithm.

Feasibility Pump / FeasibilityPump

Name	FeasibilityPump
Default	1
Min	0 – False/Off
Max	1 – True/On
Type	Integer

The new Feasibility Pump methods for both continuous and integer variables help the Evolutionary Solver find better solutions, faster than ever. This option allows the Evolutionary Solver to search for a feasible solution and add it to the initial population. Feasibility Pump uses a set of fast heuristic and exact methods that seek a feasible solution, but not necessarily a high quality solution, when finding a feasible solution may be as challenging as finding the optimal solution.

Evolutionary Parameters for Integer Problems

Where the other Solver engines use the Branch & Bound method to solve problems with integer constraints, the Evolutionary Solver handles integer constraints on its own, and is subject to the limits set in the following parameters, Max Subproblems, Max Feasible Solutions, Tolerance, Max Time Without Improvement, and Solve Without Integer Constraints. Unlike the other Solver engines, the Evolutionary Solver *always* works on a series of subproblems, even if there are no integer constraints in the model – so these options are always important for the Evolutionary Solver.

Max Subproblems

Name	MaxSubproblems
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Subproblems parameter places a limit on the number of subproblems that may be explored by the evolutionary algorithm before the SDK stops with the Status result, *Optimize_Status_Bpmips_Limit*. In the Evolutionary Solver, a subproblem consists of a possible

mutation step, a crossover step, an optional local search in the vicinity of a newly discovered "best" point, and a selection step where a relatively "unfit" member of the population is eliminated.

Max Feasible Solutions

Name	MaxFeasibleSols
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Feasible Solutions parameter places a limit on the number of feasible solutions found by the evolutionary algorithm before the SDK stops with the status result, *Optimize_Status_Bpnode_Limit*. A feasible solution is any solution that satisfies all of the constraints, including any integer constraints.

Tolerance

Name	IntTolerance
Default	0
Min	0
Max	1.0
Type	Double

This option works in conjunction with the Max Time without Improvement option to limit the time the evolutionary algorithm spends without making any significant progress. If the relative (i.e. percentage) improvement in the best solution's "fitness" is less than the Tolerance parameter's value, the Evolutionary Solver stops as described below. Since the population may include members representing infeasible solutions, the "fitness" value is a combination of an objective function value and a penalty for infeasibility.

Max Time without Improvement

Name	MaxTimeNoImp
Default	30
Min	0
Max	2,147,483,647
Type	Integer

This option works in conjunction with the Tolerance option to limit the time the evolutionary algorithm spends without making any significant progress. If the relative (i.e. percentage) improvement in the best solution's "fitness" is less than the Tolerance parameter value, the SDK stops with the

Status result, *Optimize_Status_No_Remedies* unless the evolutionary algorithm has discovered no feasible solutions at all, in which case the status is *Optimize_Status_Unfeasible*. If you believe that this stopping condition is being met prematurely, you can either make the Tolerance value smaller (or even zero), or increase the number of seconds allowed by the Max Time without Improvement option.

Solve Without Integer Constraints

Name	SolveWithout
Default	0 - Off

Min	0
Max	1 – On
Type	Integer

If you solve your Problem with this parameter set to 1, the Solver *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem, which is often quite useful.

Mixed Integer Problem Parameters

This section describes parameters used when solving integer programming problems for all Solver engines *except* the LP Simplex and the LP/Quadratic Solver. The LP Simplex Solver's integer options are described in the section, *LP Simplex Solver MIP Parameters* and the LP/Quadratic Solver's integer options are described in the section, *LP/Quadratic Solver MIP Parameters* above. This section explains all of the options that pertain to the solution of problems with integer constraints.

Max Subproblems

Name	MaxSubproblems
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Subproblems parameter places a limit on the number of subproblems that may be explored by the Branch & Bound algorithm before the SDK stops with the status result, *Optimize_Status_Bpmips_Limit*. Each subproblem is a "regular" Solver problem with additional bounds on the variables. In a problem with integer constraints, the Max Subproblems limit should be used in preference to the Iterations limit.

Max Feasible (Integer) Solutions

Name	MaxIntegerSols
	MaxFeasibleSols
Default	2,147,483,647
Min	0
Max	2,147,483,647
Type	Integer

The value for the Max Integer Solutions parameter places a limit on the number of "candidate" integer solutions found by the Branch & Bound algorithm before the SDK stops with the status result, *Optimize_Status_Bpnode_Limit*. Each candidate integer solution satisfies all of the constraints, including the integer constraints; the SDK retains the integer solution with the best objective value so far, called the "incumbent."

It is entirely possible that, in the process of exploring various subproblems with different bounds on the variables, the Branch & Bound algorithm may find the same integer solution (set of values for the decision variables) more than once; the Max Integer Solutions limit applies to the total number of integer solutions found, not the number of "distinct" integer solutions.

Integer Tolerance

Name	IntTolerance

Default	0
Min	0
Max	1.0
Type	Double

When you solve an integer programming problem, it often happens that the Branch & Bound method will find a good solution fairly quickly, but will require a great deal of computing time to find (or verify that it has found) the optimal integer solution. The Integer Tolerance setting may be used to tell the SDK to stop if the best solution it has found so far is "close enough."

The Branch & Bound process starts by finding the optimal solution without considering the integer constraints (this is called the *relaxation* of the integer programming problem). The objective value of the relaxation forms the initial "best bound" on the objective of the optimal *integer* solution, which can be no better than this. During the optimization process, the Branch & Bound method finds "candidate" integer solutions, and it keeps the best solution so far as the "incumbent." By eliminating alternatives as it proceeds, the B&B method also tightens the "best bound" on how good the integer solution can be.

Each time the Solver finds a new incumbent – an improved all-integer solution – it computes the maximum percentage difference between the objective of this solution and the current best bound on the objective:

$$\frac{\text{Objective of incumbent} - \text{Objective of best bound}}{\text{Objective of best bound}}$$

If the absolute value of this maximum percentage difference is equal to or less than the Integer Tolerance, the Solver will stop and report the current integer solution as the optimal result. In the Solver SDK, the status result will be *Optimize_Status_Incumb_Cand*. If you set the Integer Tolerance to zero, the Solver will continue searching until all alternatives have been explored and the optimal integer solution has been found. This could take a great deal of computing time.

Integer Cutoff

Name	IntCutoff
Default	2e+30
Min	-2e-30
Max	2e+30
Type	Double

This option provides another way to save time in the solution of mixed-integer programming problems. If you know the objective value of a feasible integer solution to your problem – possibly from a previous run of the same or a very similar problem – you can enter this objective value for the Integer Cutoff parameter. This allows the Branch & Bound process to *start* with an "incumbent" objective value (as discussed above under Integer Tolerance) and avoid the work of solving subproblems whose objective can be no better than this value. If you enter a value for this parameter, you must be *sure* that there is an integer solution with an objective value at least this good: A value that is too large (for maximization problems) or too small (for minimization) may cause the SDK to skip solving the subproblem that would yield the optimal integer solution.

Solve Without Integer Constraints

Name	SolveWithout
Default	0 - Off

Min	0
Max	1 – On
Type	Integer

If you solve your Problem with this parameter set to 1, Solver SDK *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem.

Convergence Monitoring Options

Simulation Convergence Monitoring provides guidance on the number of trials needed in future simulations to ensure stable and reliable results.

Convergence can be applied to any combination of the mean, standard deviation, and specified percentile, for all outputs. Users can set a convergence tolerance, such as 2%, along with a confidence level, like 95%. Solver SDK will report the number of trials required to ensure, with 95% confidence, the monitored output is within 2% of its true value. All outputs must satisfy the convergence.

Otherwise, non-convergence will be returned to `problem.Solver.SimulateStatus`. To turn off convergence monitoring, set all three "Perform Test" options to False.

Convergence Step

Name	ConvStep
Default	100
Min	1
Max	#Trials/2
Type	Double

This option determines how frequently the Risk Solver Engine (within Solver SDK) checks for convergence during a simulation. While it does not affect the stringency of the convergence test, it serves as a trade-off between efficiency and runtime. More frequent convergence checks may result in fewer iterations but longer simulation time, due to the additional overhead of the checks. Conversely, less frequent checks may extend the number of iterations, but reduce the overall runtime for the same reason. Recommended setting: $2 * \text{ConvStep} \leq \text{NumTrials}$.

Convergence Tolerance

Name	ConvTol
Default	0.05
Min	0.01
Max	0.99
Type	Double

Convergence tolerance sets the acceptable change in output statistics (like mean, standard deviation, or a specified percentile) between iterations. Once the simulation is complete, Solver SDK will return the number of trials required for convergence to `problem.Solver.SimulateStatus`.

For example, with a 2% convergence tolerance, the report will indicate how many trials are required for the change in the output (e.g., the mean) to be less than 2% between iterations. This ensures the results are stable and reliable, avoiding unnecessary extra iterations.

Convergence Confidence Level

Name	ConvLevel
Default	0.95
Min	0.05
Max	0.95
Type	Double

The Convergence Confidence Level sets the probability that the simulation results for mean and standard deviation (only) are within a specified tolerance. A higher confidence level requires more iterations to ensure greater accuracy and result stability. Conversely, a lower confidence level may reduce the number of required trials but can lead to less consistent results.

For example, with a 95% confidence level and a 2% tolerance, Solver SDK will report the number of trials required for convergence, with 95% confidence, that the monitored output will be within 2% of its true value. The option is used in computing the mean and standard deviation in the same way.

Convergence at Percentile

Name	ConvAtPerc
Default	0.50
Min	0.01
Max	0.99
Type	Double

Instead of tracking summary statistics like the mean or standard deviation, this option monitors a chosen percentile (e.g., 5th, 50th, or 95th percentile). This option evaluates how much the specified percentile changes between simulation iterations.

For example, with a 2% convergence tolerance and the 90th percentile selected, Solver SDK will calculate and report the number of trials required for the 90th percentile to stabilize with less than a 2% change between iterations. This ensures that percentile-based insights are statistically reliable without requiring unnecessary additional simulation runs.

Perform Test on Mean, Stdev, or Percentiles

If only one statistic (Mean, Stdev or Percentiles) is selected, the simulation will consider convergence based solely on the selected statistic. If all criteria are selected, the simulation will check for convergence across all of them. Convergence is considered achieved when all selected criteria are met. If no criteria are selected, convergence is not requested. **This omission will be returned via `problem.Solver.SimulateStatus` or a message will be returned indicating that no convergence was performed.** To turn off Convergence Monitoring, set all Perform Test options to False. A message will be output to the “status” portion of the RASON output, as shown in the figure below.

TODO – need screenshot

Perform Test on Mean

Name	ConvTestMean
Default	1

Min	0
Max	1
Type	Boolean

This setting monitors the **mean** of the output(s) and checks whether it is changing significantly between iterations. If the change in the mean is below the defined **convergence tolerance** over a certain number of iterations, **Solver SDK will return the number of trials needed for the mean to stabilize TODO**. Enabling this option, ensures that the recommended number of trials required for the mean to reach a stable, reliable value is reported.

Perform Test on Stdev

Name	ConvTestStdev
Default	1
Min	0
Max	1
Type	Boolean

This setting monitors the standard deviation of the output(s) and checks whether it is changing significantly between iterations. If the change in the standard deviation is below the defined **convergence tolerance** over a certain number of iterations, Solver SDK will report the number of trials needed for the standard deviation to stabilize, **in the “status” section of the RASON JSON results**. Enabling this option, **ensures that the “status” portion of the RASON output includes the recommended number of trials required for the standard deviation to reach a stable, reliable value**.

Perform Test on Percentiles

Name	ConvTestPerc
Default	1
Min	0
Max	1
Type	Boolean

This setting monitors a specific percentile of the output(s) and checks whether it is changing significantly between iterations. If the change in the percentile is below the defined convergence tolerance over a certain number of iterations, Solver SDK will report the number of trials needed for the percentile to stabilize, **in the “status” section of the RASON JSON results**. **Enabling this option, ensures that the “status” portion of the RASON JSON results includes the recommended number of trials required for the percentile to reach a stable, reliable value**.

Simulation Convergence Monitoring Results

After a simulation is completed, the “status” portion of the RASON JSON results includes convergence monitoring results, which indicate how many trials are needed to stabilize the simulation output within the specified tolerance and confidence level. If convergence is achieved, the report displays the recommended number of trials required to produce reliable results. If convergence is not achieved, RASON provides a message explaining the reason. Based on this message, you may need to adjust the number of trials in future simulations.

Convergence Messages and Their Meaning

- ***“Simulation convergence has been achieved at <number> trials.”***
The RASON JSON results include the Number of Trials Required for Convergence. Convergence can be achieved with fewer trials than the entered Number of Trials specified within the “modelSettings” portion of the RASON model. To save time and computing power, reduce the number of trials to the recommended value for subsequent simulations.
- ***“Simulation convergence test has been disabled. Set convergence parameters in engine settings.”***
This indicates that convergence monitoring was disabled during the simulation.
 - To disable convergence monitoring, set all *Perform Test* options to False in the “engineSettings” portion of the RASON model.
 - To enable convergence monitoring, set at least one *Perform Test* option to True in the “engineSettings” portion of the RASON model.
- ***“Simulation may not have converged. Increase the number of trials or adjust convergence parameters in engine settings.”***
The number of trials used was insufficient to determine convergence. Increase the number of trials in the Platform tab of the Solver Task Pane and re-run the simulation.

Using Solver SDK Dedicated Server

Introduction

Solver SDK Dedicated Server (also known as *SolverServer*) was designed to meet the growing demand for prescriptive solutions in secure, private and public mission-critical applications. SDK Dedicated Server allows any client machine (which can be a PC, phone, tablet, etc) to run either a simulation or optimization model with the Solver SDK Platform (installed onto a machine operating as the Server) without having the SDK installed on the client machine.

Solver SDK Dedicated Server allows customers a subset of basic RASON REST API calls supporting optimization and simulation, allowing users to call the RASON server themselves, either on-premise or in their own private cloud accounts. RASON is Frontline Systems' high-level modeling language. Since 2015, our RASON Decision Services, Frontline's public cloud server (at Rason.net, developer portal at Rason.com) has supported a wide range of Frontline- and customer-designed applications, with over 360,000 users.

Solver SDK Dedicated Server, easy to install via a Setup program, runs as a service on nearly any Windows Server or Windows 10 computer. It comes with a "server admin" tool that runs on any Windows client PC, making it easy to manage multiple server and client machines, licenses for the RASON API, modeling system and optional Solver Engines, authorized users and security, job priorities and resource usage. Administrators can display active client sessions and review logs of client sessions over time.

Solver SDK Dedicated Server does not currently offer support for many RASON features such as model naming (numeric IDs must be used), model interfaces and arguments, model versioning, multi-stage model workflows, etc. and little support for logging and reporting. Information security is the responsibility of the user. Solver Server runs as a Windows Service, as unmanaged code with full access to the machine.

Installing the Software Via Installation Program

Please see the Installation chapter for instructions on how to install the Solver SDK Platform on the machine that will be acting as the server. To start Solver SDK Dedicated Server automatically, click the "Start the SolverServer service now" checkbox on the last installation dialog. It is also recommended that you click the "Read me..." checkbox to open Readme.doc. This file includes important installation information. Then click **Finish**.

To install, register, and start Solver SDK Dedicated Server using a command prompt

1. Click Start/Programs/Accessories. Right click **Command Prompt** and choose **Run as Administrator** from the menu.
2. Type the following at the prompt to change the directory to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin, then press the Enter key.
`cd C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin`

```

Command Prompt

c:\>cd c:\Program Files\Frontline Systems\Solver SDK Platform\SolverServer\Server\Bin
c:\Program Files\Frontline Systems\Solver SDK Platform\SolverServer\Server\Bin>

```

3. To register and start SolverServer, type: `SolverServer.exe -ir`
Then press the Enter key.

```

Administrator: Command Prompt

c:\Program Files\Frontline Systems\Solver SDK Platform\SolverServer\Server\Binsolverserver.exe
SolverServer.exe -i/i/-install/install installs Frontline SDK Server
SolverServer.exe -ir/i/-install and starts Frontline SDK Server
SolverServer.exe -u/u/-uninstall/uninstall uninstalls Frontline SDK Server
SolverServer.exe -r/r/-run/run-start/start starts Frontline SDK Server
SolverServer.exe -k/k/-kill/kill-stop/stop stops Frontline SDK Server

c:\Program Files\Frontline Systems\Solver SDK Platform\SolverServer\Server\Binsolverserver.exe -ir
Success: Service SolverServer installed
Success: Service SolverServer started

c:\Program Files\Frontline Systems\Solver SDK Platform\SolverServer\Server\Bin>

```

This command installs and starts the Solver SDK Dedicated Server service. By default, the service opens ports 2052 and 2050 where port 2050 is the Administrative Port. (The Windows Firewall may ask to open port 2052. Please allow for local addresses.) Please see below for instructions on how to change port settings in the section below, "Managing Solver Server".

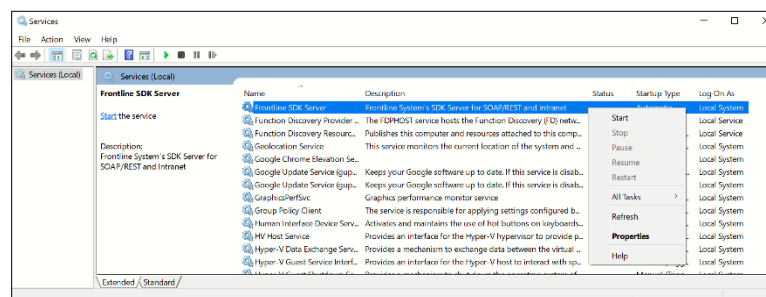
Commands

- i : Installs Solver Server as a service
- ir: Installs and starts Frontline SDK Server
- k : stops Solver Server service
- r : Starts Solver Server service
- k : stops Solver Server service
- u : uninstalls Solver Server from Windows Service Manager

To restart service, SolverServer must first be stopped (SolverServer.exe -k).

For a complete list of command line arguments, please type SolverServer.exe at the command prompt, without any arguments.

SolverServer service can also be managed through Windows Service Manager. The SolverServer service will be listed as "Frontline SDK Server". (Windows Service Manager can be found on the **Control Panel** under **Administrative Tools >> Services**.)



Solver SDK Dedicated Server is now running, click the "X" in the upper left hand corner of the DOS prompt to close.

Using the REST API

This chapter introduces the RASON REST API function calls that may be used to analyze a model, solve an optimization or run a simulation from within your own application.

You can use separate REST API calls to (i) upload your model plus data files, create a **model resource** on the Rason server, (ii) start an optimization or simulation, (iii) check on its status and get progress information, and finally (iv) retrieve results when the solution process is complete. If necessary, you can stop a long run in progress with an API call.

Other REST API calls allow you to *diagnose* a model to determine its type (linear, quadratic, nonlinear, etc.) and size before solving it (mostly useful in model design and development), retrieve a model resource that you've previously created (including the full text of the model), update the model text of an existing model resource, get a list of all the model resources you've created, and delete a model resource.

Note: Solver SDK Dedicated Server supports a limited subset of the REST API endpoints available on the RASON Server and contains no support for many RASON features such as model naming (numeric IDs must be used), model interfaces and arguments, model versioning, multi-stage model workflows, etc. and little support for logging and reporting. **Every aspect of information security for Solver SDK Dedicated Server is the responsibility of the organization. Solver Server runs as a Windows Service, as unmanaged code with full access to the machine.**

This section gives a detailed account of how to use the Rason REST API beginning with how to pass the authorization header (required on each call to the API), then moving on to obtaining and passing the model resource ID, what to expect as a response to a call to a REST API endpoint, unauthorized errors and ending with a description of each REST API endpoint.

Authorization Headers

Every Rason REST API call must include an 'Authorization' header with value 'Certificate' followed by your Rason Server Certificate. Use SAdmin.exe (installed into C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin) to generate your Rason Server Certificate. For instructions on how to generate this certificate, see the chapter "Managing Solver SDK Dedicated Server". An example of an Authorization Header is below.

```
'Authorization': 'Certificate ' + '[YOUR Rason SERVER  
CERTIFICATE] },
```

Model Resources and Location Headers

When you create a model resource using the Rason API endpoint POST `<server_address>/api/model`, the response will include a 'Location' header whose value is the resource URI which includes the model ID.

Note: You will replace `<server_address>` with the actual address of your Rason Server such as `localhost:2053`, if running the server locally, or, if the server is running on another machine, `10.1.1.1:2053`, where "10.1.1.1" is the IP address of the server, or `Frontline:2053`, where "Frontline" is the name of the server. For example, if running the server locally, the Rason API endpoint above would be `localhost:2053/api/model`. If running the server on a remote machine and using the same IP addresses, the Rason API endpoint above would change to: `10.1.1.1:2053/api/model` or `Frontline:2053/api/model` where "Frontline" is the name of the server.

When you make an API call that requires a model id, you simply substitute the string you get from the Location header for *id*. Afterwards, you can call GET

```
<server_address>/api/model/id/optimize or GET
```

```
<server_address>/api/model/id/simulate to place the model in the queue on the Server.  
After the model is placed in the queue, you should periodically check the status of the model, which
```

will either be "complete" or "incomplete". Once the model has finished solving, or the status returned is "complete", then you can call `GET <server_address>/api/model/id/result` to obtain the final solution to the model. Note: If solving a hard, mixed integer optimization model or nonsmooth optimization model, calling `GET <server_address>/api/model/id/result` will return the best solution found so far and the number of subproblems completed.

To delete model resources when you no longer need them, just use the API call `DELETE <server_address>/api/model/id/delete`. Deleting a model resource also removes any data files that were uploaded when it was created.

Responses: Solution Status and Results

When you call `GET <server_address>/api/model/id/status` to check the solution status, the JSON response format is fixed, and easy to examine. For example, if the solution process has completed, you'll get `{ "status": "Complete" }`, you can check for this with code such as `if (response.status == "Complete")`. When you call `GET <server_address>/api/model/id/result` to retrieve results, the response includes a fixed portion describing the solution status, and a variable portion that depends on *what you ask for in the Rason model*.

For example, here is an example response from an optimization model:

```
{
  "status" : {"code":0, "codeText": "Solver found a solution. All
constraints and optimality conditions are satisfied."},
  "variables" : {
    "x" : { "finalValue" : 1.250000 },
    "y" : { "finalValue" : 4.000000 },
    "z" : { "finalValue" : 2.136001 }
  },
  "objective" : {
    "obj" : {"finalValue" : 2.136001 }
  }
}
```

The fixed part of the response includes the "status" property, which tells you the outcome of the optimization. You can test this with JavaScript code such as: `if (response.status.code == 0)`. The variable part of the response includes final values for the three decision variables x, y and z, and the objective being minimized (which is the third variable, z).

To retrieve the final value of the objective in your JavaScript code, you would simply reference `response.objective.z.finalValue`. If the model had asked for the dual value of x by including the property `dualValue: []`, your JavaScript code could obtain this value with a reference to `response.variables.x.dualValue`.

Unauthorized Errors

The Rason REST API will return an appropriate HTTP Status Code to each request.

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.

Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

Synchronous Quick Calls to REST API Endpoints

Each of these calls will allow you to perform a function (analyze, optimize, or simulation) and obtain the results with a single API call. Note that these calls should be limited to solving models that solve quickly, i.e. < 30 seconds.

- POST <server_address>/api/diagnose
- POST <server_address>/api/optimize
- POST <server_address>/api/simulate

REST Endpoint: POST: <server_address>/api/diagnose

A REST API endpoint that accepts a Rason model, runs a model diagnosis and returns a result in JSON format.

Request

- Example:

```
{
  "modelSettings": {
    "transformStochastic": 'robustCounterpart',
    "numTrials": 1000,
    "randomSeed": 1
  },
  "variables": {
    "x": {
      "dimensions": [8],
      "type": "binary",
      "finalValue": []
    }
  },
  "uncertainVariables": {
    "c": {
      "dimensions": [8]
    },
    "d": {
      "dimensions": [8],
      "formula": "PsiBinomial(1, 0.9)"
    },
    "c[1]": {
      "formula": "PsiTriangular(400000, 500000, 900000)"
    },
    "c[2]": {
      "formula": "PsiTriangular(500000, 750000, 1250000)"
    },
    "c[3]": {
      "formula": "PsiTriangular(500000, 1000000, 1500000)"
    },
    "c[4]": {
```

```

        "formula": "PsiTriangular(400000, 600000, 900000)"
    },
    "c[5]": {
        "formula": "PsiTriangular(250000, 500000, 750000)"
    },
    "c[6]": {
        "formula": "PsiTriangular(300000, 500000, 600000)"
    },
    "c[7]": {
        "formula": "PsiTriangular(200000, 450000, 700000)"
    },
    "c[8]": {
        "formula": "PsiTriangular(400000, 500000, 700000)"
    }
},
"formulas": {
    "f": {
        "value": [325000, 450000, 550000, 300000, 150000,
            250000, 150000, 325000]
    },
    "cash": {
        "formula": "sumproduct(c * d - f, x)"
    }
},
"constraints": {
    "invest": {
        "formula": "sumproduct(f, x)",
        "upper": 1500000
    }
},
"objective": {
    "total": {
        "type": "maximize",
        "formula": "cash",
        "chanceType": "ExpVal",
        "finalValue": [] }
    }
}
}

```

- Query parameters: None
- Payload (body): None
- Required Headers
 - Authorization
 - Example: Authorization: certificate {your Rason Server Certificate}

Response

- Content type: JSON
- Response body: Model diagnosis results
- Example:

```

{
    "modeltype": "linear",

```

```

    "variables": "8",
    "functions": "2",
    "bounds": "0",
    "integers": "8"
}

```

REST Endpoint: POST: <server_address>/api/optimize

A REST API endpoint that accepts a Rason model, performs an optimization and returns a result in JSON format.

Request

- Example:

```

{
  "variables": {
    "x": {
      "lower": [0, 0, 0],
      "finalValue": []
    }
  },
  "constraints": {
    "chassisUsed": {
      "formula": "x[1] * 1 + x[2] * 1 + x[3] * 0",
      "upper": 450
    },
    "screensUsed": {
      "formula": "x[1] * 1 + x[2] * 0 + x[3] * 0",
      "upper": 250
    },
    "speakersUsed": {
      "formula": "x[1] * 2 + x[2] * 2 + x[3] * 1",
      "upper": 800
    },
    "powerUsed": {
      "formula": "x[1] * 1 + x[2] * 1 + x[3] * 0",
      "upper": 450
    },
    "electronicsUsed": {
      "formula": "x[1] * 2 + x[2] * 1 + x[3] * 1",
      "upper": 600
    }
  },
  "objective": {
    "obj": {
      "type": "maximize",
      "formula": "x[1] * 75 + x[2] * 50 + x[3] * 35",
      "finalValue": []
    }
  }
}

```

- Query parameters: None
- Payload (body): None
- Required Headers:
 - Authorization

- Example: Authorization: certificate {your Rason Server certificate}

Response

- Content type: JSON
- Response body: Final model result. This result will always include a "code" and "codeText".
- Example:

```
{
  "status": {
    "code": 0,
    "codeText": "Solver found a solution. All constraints and
    optimality conditions are satisfied."
  },
  "variables": {
    "x": {
      "finalValue": [200, 200, 0]
    }
  },
  "objective": {
    "obj": {
      "finalValue": 25000
    }
  }
}
```

REST Endpoint: POST: <server_address>/api/simulate

A REST API endpoint that accepts a Rason model, runs a simulation and returns a result in JSON format.

Request

- Example:

```
{
  "modelSettings": {
    "numSimulations": 1,
    "numTrials": 1000,
    "randomSeed": 1
  },
  "data": {
    "price": {
      "value": 200 },
    "capacity": { "value": 100 },
    "sold": { "value": 110 },
    "refund_no_shows": { "value": 0.5 },
    "refund_overbook": { "value": 1.25 }
  },
  "uncertainVariables": {
    "no_shows": {
      "formula": "PsiLogNormal(0.1*sold, 0.06*sold)",
      "mean": []
    }
  },
  "formulas": {
    "show_ups": {
```

- ```

 "formula": "sold - Round(no_shows, 0)"
 },
 "overbook": {
 "formula": "Max(0, show_ups - capacity)"
 },
 "uncertainFunctions": {
 "revenue": {
 "formula": "price *(sold -
 refund_no_shows*Round(no_shows, 0) -
 refund_overbook*overbook)",
 "mean": [], "percentiles": []
 }
 }
}

```
- Query Parameters: None
  - Payload (body): None
  - Required Headers:
    - Authorization
    - Example: Authorization: certificate {your Rason Server certificate}

## Response

- Content type: JSON
- Response body: Simulation results. This result will always include a "code" and "codeText".
- Example:

```

{
 "status": {
 "code": 0,
 "codeText": "Solver has completed the simulation."
 },
 "uncertainFunctions": {
 "revenue": {
 "mean": 20445.3,
 "percentiles": [
 18599, 19098, 19300, 19500, 19695, 19800, 19800, 19900,
 19950, 19950, 19950, 20000, 20000, 20100, 20100, 20100,
 20100, 20100, 20100, 20100, 20200, 20200, 20250, 20250,
 20250, 20250, 20250, 20250, 20250, 20300, 20300, 20300,
 20400, 20400, 20400, 20400, 20400, 20400, 20400, 20400,
 20400, 20400, 20400, 20500, 20500, 20500, 20526.5,
 20550, 20550, 20550, 20550, 20550, 20550, 20550, 20550,
 20600, 20600, 20600, 20600, 20700, 20700, 20700, 20700,
 20700, 20700, 20700, 20700, 20700, 20700, 20700, 20700,
 20700, 20700, 20800, 20800, 20800, 20800, 20800, 20850,
 20850, 20850, 20850, 20850, 20850, 20850, 20850, 20900,
 20900, 20900, 20900, 20900, 20900, 21000, 21000, 21000,
 21000, 21000, 21000, 21000, 21000]
]
 }
 },
 "uncertainVariables": {
 "no_shows": {

```

```

 "mean": 10.9922
 }
}
}

```

## Asynchronous Rason REST API Endpoints

Frontline Systems' Rason Server offers a second set of asynchronous REST API calls. When using these endpoints, you'll first need to make a call to the REST API endpoint POST

<server\_address>/api/model which will create a "model resource". Then you'll need to start an optimization or simulation via the REST API endpoints GET

<server\_address>/api/model/id/optimize or GET

<server\_address>/api/model/id/simulate. You can check on the model's progress at any time with the REST API endpoint GET <server\_address>/api/model/id/status, and obtain results when completed with the REST API endpoint GET

<server\_address>/api/model/id/result. To stop an optimization or simulation that is in progress, use POST <server\_address>/api/model/id/stop. To delete a model, use DELETE <server\_address>/api/model/id/delete.

- DELETE <server\_address>/api/model/id/delete
- GET <server\_address>/api/model
- GET <server\_address>/api/model/id
- GET <server\_address>/api/model/id/optimize
- GET <server\_address>/api/model/id/result
- GET <server\_address>/api/model/id/result/data
- GET <server\_address>/api/model/id/simulate
- GET <server\_address>/api/model/id/status
- POST <server\_address>/api/model
- POST <server\_address>/api/model/id/stop
- PUT <server\_address>/api/model/id

### ***REST Endpoint: DELETE: <server\_address>/api/model/{id}/delete***

A REST API endpoint that deletes the model resource from the Server.

#### Request

- Example: [<server\\_address>/api/model/{id}/delete](#)
- Query parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}



## Response

- Response Body: None
- Response Header: None

### ***REST Endpoint: GET: <server address>/api/model***

A REST API endpoint that returns all available model resource IDs.

## Request

- Example: [<server address>/api/model](#)
- Query parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

## Response

- Content type: JSON
- Response body: Array of model ID's
- Example:

```
["12345+2015-06-02-14-21-01-517971", "12345+2015-08-27-19-43-55-610106", "12345+2015-08-27-19-45-27-993434"]
```

### ***REST Endpoint: GET: <server address>/api/model/{id}***

A REST API endpoint that retrieves the model associated with the model resource ID.

## Request

- Example: [<server address>/api/model/{id}](#)
- Query parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

## Response

- Content type: JSON
- Response body: Rason model
- Example:

```
{
 "variables": {
 "x": {
 "lower": [0, 0, 0],
```

```

 "finalValue": []
 },
 "constraints": {
 "chassisUsed": {
 "formula": "x[1] * 1 + x[2] * 1 + x[3] * 0",
 "upper": 450
 },
 "screensUsed": {
 "formula": "x[1] * 1 + x[2] * 0 + x[3] * 0",
 "upper": 250
 },
 "speakersUsed": {
 "formula": "x[1] * 2 + x[2] * 2 + x[3] * 1",
 "upper": 800
 },
 "powerUsed": {
 "formula": "x[1] * 1 + x[2] * 1 + x[3] * 0",
 "upper": 450
 },
 "electronicsUsed": {
 "formula": "x[1] * 2 + x[2] * 1 + x[3] * 1",
 "upper": 600
 }
 },
 "objective": {
 "obj": {
 "type": "maximize",
 "formula": "x[1] * 75 + x[2] * 50 + x[3] * 35",
 "finalValue": []
 }
 }
}

```

### ***REST Endpoint: GET: <server address>/api/model/{id}/optimize***

A REST API endpoint that submits the previously submitted model into the queue on the Server.

#### **Request**

- Example: <http://<server address>/api/model/{id}/optimize>
- Query parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

#### **Response**

- Content type: JSON
- Response body: {"Accepted": " + {timeAccepted} + " }
- Example:

```
{"Accepted": "2015-08-28+21-22-59" }
```

***REST Endpoint: GET: <server\_address>/api/model/{id}/result***

A REST API endpoint that retrieves the results from the model previously submitted to GET <server\_address>/api/model/{id}/optimize or GET <server\_address>/api/model/{id}/simulate.

**Request**

- Example: [<server\\_address>/api/model/{id}/result](#)
- Query parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

**Response**

- Content type: JSON
- Response body: Model results for previously submitted model to GET <server\_address>/api/model/id/optimize or GET <server\_address>/api/model/id/simulate.
- Example:

```
{
 "status" : {
 "code" : 0,
 "codeText" : "Solver found a solution. All constraints
and optimality conditions are satisfied."
 },
 "variables" : {
 "d9:f9" : {
 "finalValue" : [[200, 200, 0]],
 "dualValue" : [[0, 0, -2.500000]]
 }
 },
 "objective" : {
 "d18" : {
 "initialValue" : 0 }
 },
 "constraints" : {
 "c11:c15" : {
 "dualValue" : [0, 0, 12.500000, 0, 25] }
 }
}
```

***REST Endpoint: GET: <server\_address>/api/model/{id}/result/{data}***

A REST API endpoint that retrieves the output file containing results from the model previously submitted to GET <server\_address>/api/model/id/optimize or GET

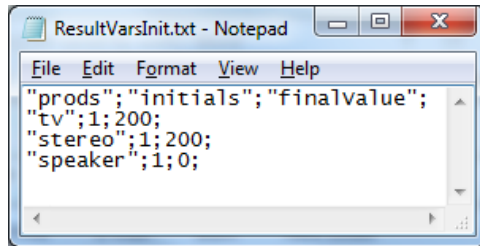
<server\_address>/api/model/id/simulate. You must specify the output file within the dataSources section of your Rason model.

## Request

- Example: [http://<server\\_address>/api/model/{id}/result/{data}](http://<server_address>/api/model/{id}/result/{data})
- Query parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

## Response

- Content type: File Attachment
- Response body: Returns the contents of the file specified within the dataSources section of the Rason model.
- Example: The screenshots below give examples of what data sources may be returned. For a complete list of data sources supported in the Rason modeling language, please see the "Data Sources" section within the *Rason Model Components* chapter of the *Rason Reference Guide*.
  - A CSV file.



- An Excel file.

|   | Q       | R        | S          | T | U     | V          | W | X          |
|---|---------|----------|------------|---|-------|------------|---|------------|
| 1 | prods   | initials | finalValue |   | parts | finalValue |   | finalValue |
| 2 | tv      | 1        | 200        |   | chas  | -50        |   | 25000      |
| 3 | stereo  | 1        | 200        |   | tube  | -50        |   |            |
| 4 | speaker | 1        | 0          |   | cone  | 0          |   |            |
| 5 |         |          |            |   | psup  | -50        |   |            |
| 6 |         |          |            |   | elec  | 0          |   |            |
| 7 |         |          |            |   |       |            |   |            |
| 8 |         |          |            |   |       |            |   |            |

- An Access database file.

| ID    | Profit | Products | prods   | initials | finalValue |
|-------|--------|----------|---------|----------|------------|
| 82    | 75     | tv       | tv      | 1        | 200        |
| 83    | 50     | stereo   | stereo  | 1        | 200        |
| 84    | 35     | speaker  | speaker | 1        | 0          |
| (New) |        |          |         | 0        |            |

### ***REST Endpoint: GET: <server address>/api/model/{id}/simulate***

A REST API endpoint that submits the previously submitted model into the simulation queue.

#### **Request**

- Example: <http://<server address>/api/model/{id}/simulate>
- Query parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

#### **Response**

- Content type: JSON
- Response body: {"Accepted": " + {timeAccepted} + " }
- Example:  

```
{"Accepted": "2015-08-28+21-57-33" }
```

### ***REST Endpoint: GET: <server address>/api/model/{id}/status***

A REST API endpoint that retrieves the status of the model previously submitted to GET <server\_address>/api/model/id/optimize or GET <server\_address>/api/model/id/simulate.

#### **Request**

- Example: <http://<server address>/api/model/{id}/status>
- Query parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

#### **Response**

- Content type: JSON
- Response body: Status of the submitted call to  
GET <server\_address>/api/model/id/simulate or  
GET <server\_address>/api/model/id/optimize.
- Example:  

```
{"status": "Complete" }
```

```
{"status": "Incomplete", "Milliseconds" : -
2147483648, "Iterations" : 0, "Subproblems" : 11, "LocalSolutions"
: 9, "Objective" : 1294839.12895701 }
```

### ***REST Endpoint: POST: <server address>/api/model***

A REST API endpoint that submits a Rason model and returns a model resource URI in the response's 'Location' header.

NOTE: The model resource URI facilitates subsequent calls to the Rason REST API endpoints necessary for manipulating your model (i.e. perform an optimization, run a simulation, check the status of the model, get results, etc.).

#### **Request**

- Example: [<server address>/api/model](#)
- Query Parameters: None
- Payload (body): Rason model
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

#### **Response**

- Response Body: None
- Response Header : Location
- Example:  
Location: [<server address>/api/model/12345+2015-08-27-22-54-09-296799](#)

### ***REST Endpoint: POST: <server address>/api/model/{id}/stop***

A REST API endpoint that aborts the execution of the model previously submitted to GET <server\_address>/api/model/id/optimize or GET <server\_address>/api/model/id/simulate.

#### **Request**

- Example: [<server address>/api/model/{id}/stop](#)
- Query Parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

#### **Response**

- Response Body: None
- Response Header : Location

### ***REST Endpoint: PUT: <server address>/api/model/{id}***

A REST API endpoint that updates the current model on the server, using the model resource ID.

## Request

- Example: <http://<server address>/api/model/{id}>
- Query Parameters: None
- Payload (body): None
- Required Headers:
  - Authorization
  - Example: Authorization: certificate {your Rason Server certificate}

## Response

- Response Body: None
- Response Header : None

---

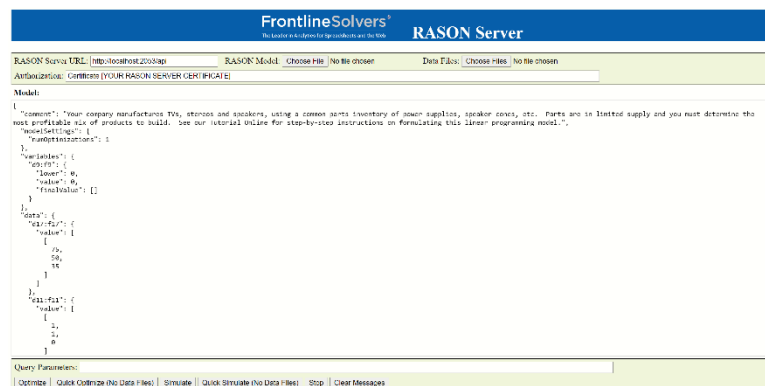
# Solving a RASON Model using Solver SDK Dedicated Server

## Introduction

Frontline Systems ships three examples written in RASON that utilize Solver SDK Dedicated Server: Optimization.html, an optimization example, Simulation.html, a simulation example, and ProjectSelect.html, a simulation optimization example. This chapter will step through each example to give you an in-depth understanding of how to quickly get up and running solving your own Rason optimization or simulation model using Solver SDK Dedicated Server.

## Solving an Optimization

This section will step through an example that uses the Rason Rest API to solve and obtain results from an optimization model. To start, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Client\Examples\REST and open Optimization.html. This example opens a web page. We will begin by looking at the "front end" of the example and later we will open this file using a text editor to view the code and the actual Rason REST API endpoints.



## RASON Server URL

At the top of the webpage, notice the Rason Server URL which is currently pointing to the Rason Server installed on a local machine, "listening" on port 2053. If your installation of Solver SDK Dedicated Server is on your local machine, then this URL will remain. However, if your installation of Solver SDK Dedicated Server is on a remote machine, then you will replace `http://localhost:2053/api` with the actual address of your SDK Server such as: `http://10.1.1.1:2053/api` where "10.1.1.1" is the IP address of the server or `http://Frontline:2053/api` where "Frontline" is the name of the server. Later, we will see exactly how this URL will be used in the actual Rason REST endpoints.

Note: - To use "https" in the URL, the Rason server must be set up with a certificate to authenticate. Users can obtain client authentication certificates from an external certification authority such as VeriSign or create a self-signed certificate.

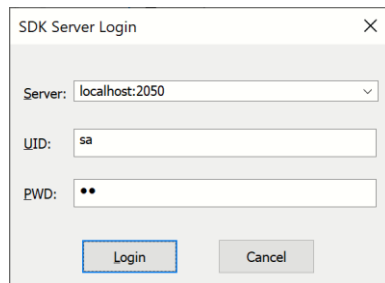
## Authorization

Every Rason REST API call must include an 'Authorization' header with value 'Certificate' followed by your Rason Server Certificate, for example:

```
'Authorization': 'Certificate ' + '[YOUR Rason SERVER CERTIFICATE]'
```

The certificate entered here will be used in the Authorization header for each REST API endpoint.

To generate the certificate, browse to `C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin` and open the `SAdmin.exe` executable file. A login screen will appear.



By default, if the Solver Server is running on the local machine, the server will run at address: `localhost:2053`. This location can be changed in the `[SolverServerN]` section of the `SolverServer.ini` file. For more information on the `SolverServer.ini` file, please see the *Managing Solver SDK Dedicated Server* section that appears later in this chapter.

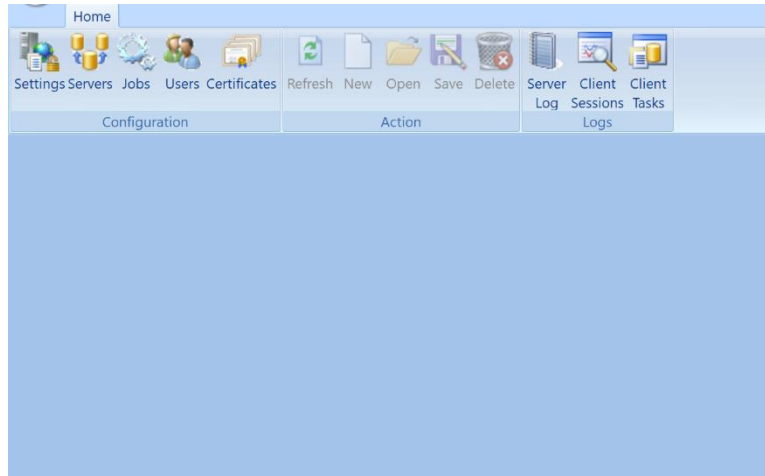
If the server is on a separate machine, then enter the name of the server or the server's IP address. For example: `10.1.1.1:2053` where "10.1.1.1" is the IP address of the server or `Frontline:2050` where "Frontline" is the name of the server. If the server is on a website, such as [www.solver.com](http://www.solver.com), then enter [www.solver.com:2053](http://www.solver.com:2053).

By default, enter "sa" (without the quotes) for UID (User ID). This can be changed to a more meaningful ID within the `Auth.ini` file. See the section *Managing Solver SDK Dedicated Server*, below, for more details.

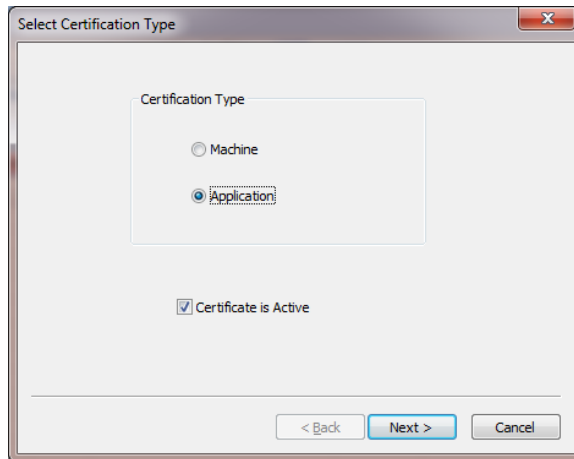
By default, enter "sa" (without the quotes) for your password. This can also be changed to a more meaningful password within `Auth.ini`. See the section *Managing Solver SDK Dedicated Server* for more details.

Click **Login**. The following dialog will open.

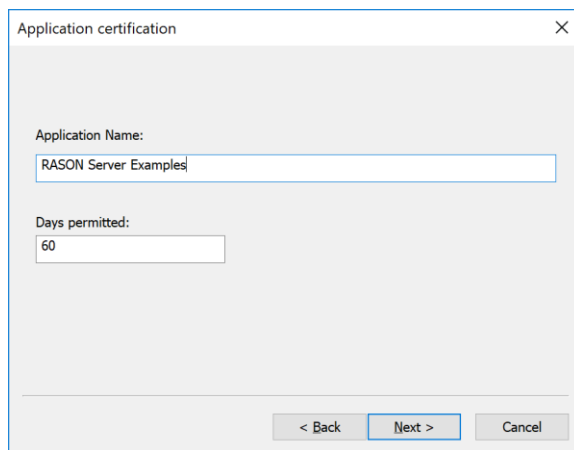




Click **Certificates**, in the Configuration group, then **New** in the Action Group on the Ribbon to open the following dialog.

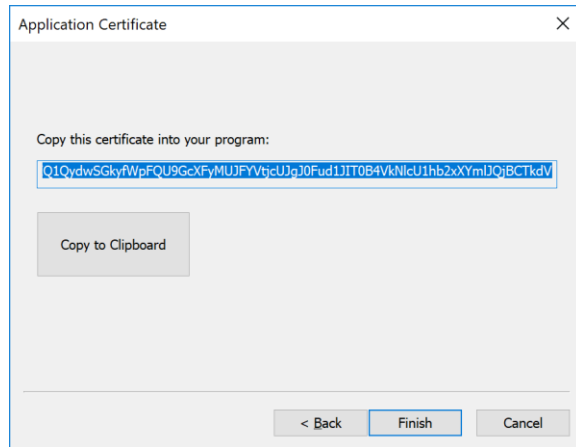


Select either **Machine** or **Application** based on the desired certificate. This example will create an Application Certificate. Confirm that the **Certificate is Active** checkbox is selected. Then click **Next** to display the following dialog.

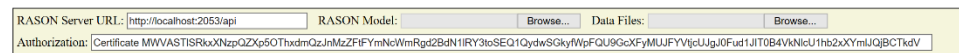


Enter an **Application Name**, such as "Rason Server Examples". The **Days Permitted** field allows the Solver SDK Dedicated Server Administrator to enforce a time limit on the certificate, if desired. Enter "0" for a certificate that never expires.

Click **Next** to generate an application certificate. Click Copy to Clipboard to copy the certificate to the clipboard.



Click **Finish** to close the dialog, then click back to `Optimization.html`, select [YOUR Rason SERVER CERTIFICATE] within the Authorization field and paste in the certificate.



The Rason Model and Data Files fields allow you to browse to and open saved Rason models and/or data files. These two fields will not be used in this example.

Within the Model field, we see the actual Rason optimization model that will be solved via the Rason Server. Scroll down to view the entire model

## Setting Up the Rason Model

When setting up an optimization model in the Rason Modeling language, your model will typically contain these essential segments:

**data:** Where arrays to be used in the calculation of constraints and the objective function will be defined

**variables:** Where the decision variables will be defined.

**objective:** Where the objective function to be maximized or minimized will be defined.

**constraints:** Where the constraints will be defined.

Within this overall framework, you have a great deal of flexibility in choosing how to define and calculate your objective function and constraints. For example, the objective function will ultimately depend on the decision variable cells, but you don't have to calculate the entire function all at once. You can use any number of calculations in the data section to compute intermediate results and use these results to calculate the objective function or the constraints. For a more in-depth discussion of the Rason modeling language, see the Rason User and Reference Guides.

```
{
 "comment": "Your company manufactures TVs, stereos and speakers,
using a common parts inventory of power supplies, speaker cones,
etc. Parts are in limited supply and you must determine the most
profitable mix of products to build. See our Tutorial Online for
step-by-step instructions on formulating this linear programming
model.",
 "modelSettings": {
 "numOptimizations": 1
 }
 "variables": {
```

```

 "d9:f9": {
 "lower": 0,
 "value": 0,
 "finalValue": []
 }
 },
 "data": {
 "d17:f17": {
 "value": [
 [
 75,
 50,
 35
]
]
 },
 "d11:f11": {
 "value": [
 [
 1,
 1,
 0
]
]
 },
 "d12:f12": {
 "value": [
 [
 1,
 0,
 0
]
]
 },
 "d13:f13": {
 "value": [
 [
 2,
 2,
 1
]
]
 },
 "d14:f14": {
 "value": [
 [
 1,
 1,
 0
]
]
 },
 "d15:f15": {
 "value": [
 [
 2,
 1,

```

```

 1
]
}
},
"constraints": {
 "c11": {
 "formula": "sumproduct(d9:f9, d11:f11)",
 "upper": 450
 },
 "c12": {
 "formula": "sumproduct(d9:f9, d12:f12)",
 "upper": 250
 },
 "c13": {
 "formula": "sumproduct(d9:f9, d13:f13)",
 "upper": 800
 },
 "c14": {
 "formula": "sumproduct(d9:f9, d14:f14)",
 "upper": 450
 },
 "c15": {
 "formula": "sumproduct(d9:f9, d15:f15)",
 "upper": 600
 }
},
"objective": {
 "d18": {
 "type": "maximize",
 "formula": "sumproduct(d9:f9, d17:f17)",
 "finalValue": []
 }
}
}

```

Models entered into the Model Editor window may be solved or analyzed using either a ‘Quick Solve’ REST API endpoint or a series of calls to standard REST API endpoints. As discussed in the previous chapter, ‘Quick Solve’ endpoints, of which there are three, are synchronous and should only be used with models that solve fairly quickly. In addition, Quick Solve endpoints do not create a "model resource" which means that a model optimized, simulated or diagnosed with a Quick Solver endpoint cannot be saved, edited, retrieved or solved at a later time. For more information on the three Quick Solver endpoints, see the earlier chapter Using the REST API.

To solve this model using the Quick Optimize endpoint (POST <server\_name>/api/optimize) click the Quick Optimize (No Data Files) command button.

The result, which will always be in JSON, will appear immediately under Messages.

Executing synchronous optimization: POST <https://rason.net/api/optimize>

```

{
 "status": {
 "code": 0,
 "id": "2590+2021-06-08-16-56-10-666576",
 "codeText": "Solver found a solution. All constraints and
 optimality conditions are satisfied."
 },
 "variables": {

```

```

 "d9:f9": {
 "finalValue": [
 [200, 200, 0]
]
 },
 "objective": {
 "d18": {
 "finalValue": 25000
 }
 }
 }
}

```

The result "Solver found a solution. All constraints and optimality conditions are satisfied" means that no other solution exists that is better than the solution found. (For more information on this and all possible Solver Result messages, please see the *Solver Result Messages* chapter that appears earlier in this guide.) To maximize profit, we should make 200 TV sets, 200 stereos, and 0 speakers. The final objective value of 25,000 is the maximum profit that we can attain.

What if you are solving a mixed-integer or global optimization model that might take 30 minutes – or overnight - to run? With a call to the standard REST API endpoint GET

<server\_address>/api/model/id/optimize you can start an optimization, obtain the model's ID by using GET <server\_address>/api/model/id, monitor the progress of Solver using the REST API endpoint GET <server\_address>/api/model/id/status, and obtain results, in JSON, when finished with the REST API endpoint GET <server\_address>/api/model/id/result. These are exactly the steps taken by the example when the "Optimize" button is clicked.

If we were to call these endpoints separately, we would see the intermediate results. For example, a call to GET <your\_server>/api/model/id/optimize returns the result, "Model has been placed in the queue to be optimized." Since we did not specify a specific engine in the Rason model, the LP/Quadratic engine is selected automatically.

A call to GET <your\_server>/api/model/id/status to obtain the status of the solve returns a result such as {"status": "Complete" }

*Status: Complete* means that the optimization has been completed. If the optimization was still in progress an incomplete status with intermediate results would be returned, for example:

```

{"status": "Incomplete", "Milliseconds" : -2147483648, "Iterations" :
0, "Subproblems" : 287, "LocalSolutions" : 8, "Objective" : 3849 }

```

A call to the REST API endpoint GET rason.net/api/model/id/result obtains the result of the optimization as displayed in the Messages field.

## Examining the Code

Let's take a look at the "insides" of Optimization.html. Right click Optimization.html and open with your desired text editor, for example Notepad or WordPad.

At the top of the page, we see the Javascript object rasonApp where various options are set in order to run the example. To change your Rason Server address so that each time Optimization.html is opened your server address appears for URL, simply change the SolverServer URL to point to the correct location. The default location is: <http://localhost:2053/api>.

```
RasonServerOptimization.html - Notepad
File Edit Format View Help
<!DOCTYPE html>
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta charset="utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<script type="text/javascript" src="https://code.jquery.com/jquery-1.9.1.min.js"></script>
<title>Frontline RASON Server</title>
<script type="text/javascript" language="javascript">
// This script demonstrates how you can solve RASON models in two ways. The first way is the 'Quick Solve'
// method and is intended to be used with simple models that solve quickly.
//
// Solving harder models using the RASON REST API is demonstrated below as well. This approach is intended for
// more complex models requiring more significant computational effort and time.
//
var rasonApp = {
 RASONModelID: false,
 solveStopped: false,
 appStatus: "ready",
 startTime: null,
 // the time we wait between result checking calls
 timeoutTime: 1000,
 modelDataFiles: [],
 // URL fragment determining whether or not we are simulating or optimizing the model
 RASONEntry: '/optimize',
 // base URL to the RASON server. this is simply the default and should be changed to the URL of your installed RASON server.
 RASONServer: 'http://localhost:2053/api',
 // The declarative model we will solve with this app. this is a simple default. Using the web page you can
 // browse to, load, and solve, you own models.
 RASONModel: {
 comment: "Your company manufactures TVs, stereos and speakers, using a common parts inventory of power supplies, speaker cones, etc. Parts are in limited quantities.",
 variables: {
 "d9:f9": {
 lower: 0, value: 0, finalValue: []
 }
 },
 data: {
 "d17:f17": {
 value: [[75, 50, 35]]
 },
 "d11:f11": {
 value: [[1, 1, 0]]
 },
 "d12:f12": {
 value: [[1, 0, 0]]
 },
 "d13:f13": {
 value: [[2, 2, 1]]
 },
 "d14:f14": {
 value: [[1, 1, 0]]
 }
 }
 }
}
```

Next, we come to the declaration of the Rason Model within the object, RASONModel. This is simply a default model. Using the web page, you can browse to, load and solve your own Rason models. Afterwards, we come to the object, ajaxOpts. As mentioned above, every Rason REST API call must include an 'Authorization' header with value 'Certificate' followed by your Rason Server Certificate. In order for the example to prefill your certificate in the example, simply copy and paste the certificate used earlier, here. In this example, this certificate will be used in the Authorization header for each REST API endpoint. For example:

```
////////////////////////////////////
ajaxOpts: {
 headers: {
 // Add the 'Authorization' header and certificate to all requests. The value of the certificate
 // can be retrieved from the RASON server administration application
 'Authorization': 'Certificate' + 'fT1SdHdoQEfmYTBSEIxRjJ5RT1HMURYW2k3MEFCQWRcSTg5YUp8ZzpXMFfFnZ01pemUyY1dXQWpIdXBUNXNJT1R3dnN',
 'Content-Type': 'application/json'
 },
 cache: false,
 context: this
},
},
```

The next two functions, quickSolve and quickSimulate, use the two Quick Solve endpoints POST <server\_address>/api/optimize and POST <server\_address>/api/simulate. These two functions are invoked when either **Quick Optimize (No Data Files)** or **Quick Simulate (No Data Files)**, respectively, are clicked.

```

////////////////////////////////////
// The following 3 functions, 'quickSolve', 'quickSimulate' and 'quickStatus', demonstrate how you can quickly solve or
// simulate a model and get the results.
quickSolve: function () {
 this.RASONModel = $('#_userModel').val();
 if(this.RASONModel){
 this.appStatus = "executing";
 this.startTime = new Date();
 updateMessages("Quick Solve started at: " + this.startTime.toString());
 var queryParams = $('#_queryParams').val();
 this.ajaxOpts.method = 'POST';
 this.ajaxOpts.data = this.RASONModel;
 $.ajax(this.RASONServe + '/optimize' + queryParams, this.ajaxOpts).done(this.quickStatus).fail(this.invokeFailed);
 } else {
 alert("No model specified");
 }
},

quickSimulate: function () {
 this.RASONModel = $('#_userModel').val();
 if(this.RASONModel){
 this.appStatus = "executing";
 this.startTime = new Date();
 updateMessages("Quick Simulate started at: " + this.startTime.toString());
 var queryParams = $('#_queryParams').val();
 this.ajaxOpts.method = 'POST';
 this.ajaxOpts.data = this.RASONModel;
 $.ajax(this.RASONServe + '/simulate' + queryParams, this.ajaxOpts).done(this.quickStatus).fail(this.invokeFailed);
 } else {
 alert("No model specified");
 }
},

quickStatus: function (data, textStatus, jqXHR) {
 var resultHtml = JSON.stringify(data, null, 2);
 rasonApp.appStatus = "ready";
 updateMessages("Model finished in: " + (Date.now() - rasonApp.startTime) + " milliseconds \n" + resultHtml);
},

```

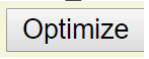
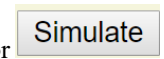
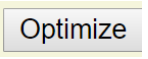
In the next section, a generic error handler is used.

```

////////////////////////////////////
// A generic error handler that cleans up
invokeFailed: function (jqXHR, textStatus, errorThrown) {
 if (!rasonApp.solveStopped) {
 updateMessages("Request failed: " + textStatus + ' ' + jqXHR.status + ': ' + jqXHR.statusText + ' ' + jqXHR.responseText);
 if (rasonApp.RASONModelID) {
 rasonApp.deleteModel();
 // We are done. Clear the data
 rasonApp.ajaxOpts.data = null;
 }
 }
 rasonApp.appStatus = "ready";
},

```

In the next section, six functions are used to optimize or simulate more complex or difficult models requiring more significant computational effort and solution times. Each function utilizes a standard REST API Endpoint such as GET <server\_address>/api/model/id/optimize (as in the

startSolve function) starting when  or  are clicked. When 

is clicked, the solving process begins with the function startSolve. As discussed earlier, when evaluating a model using the standard Rason REST API endpoints, you should follow the steps below

1. First Post the model to the Rason server by calling either GET <server\_address>/api/model/id/optimize (when solving an optimization model) or GET <server\_address>/api/model/id/simulate (when running a simulation).
2. Call GET <server\_address>/api/model/id to obtain the model's ID.
3. Call GET <server\_address>/api/model/id/status to monitor the progress of the solve.
4. Get the final results by calling GET <server\_address>/api/model/id/result.
5. Delete the model ID by calling DELETE <server\_address>/api/model/id/delete.

These steps are followed in the code below.

```

// Entry point - the 'Simulate' button was clicked.
startSimulate: function () {
 this.RASONModel = $('#_userModel').val();
 if(this.RASONModel){
 this.appStatus = "executing";
 this.startTime = new Date();
 updateMessages("Simulate started at: " + this.startTime.toString());
 this.RASONEntry = '/simulate';
 if(this.modelDataFiles.length > 0){
 this.submitModelWithDataFiles();
 } else {
 this.ajaxOpts.method = 'POST';
 this.ajaxOpts.data = this.RASONModel;
 $.ajax(this.RASONServer + '/model', this.ajaxOpts).done(this.getModelId).fail(this.invokeFailed);
 }
 } else {
 alert("No model specified");
 }
},

// Entry point - the 'Optimize' button was clicked.
startSolve: function () {
 this.RASONModel = $('#_userModel').val();
 if(this.RASONModel){
 this.appStatus = "executing";
 this.startTime = new Date();
 updateMessages("Solve started at: " + this.startTime.toString());
 this.RASONEntry = '/optimize';
 if(this.modelDataFiles.length > 0){
 this.submitModelWithDataFiles();
 } else {
 this.ajaxOpts.method = 'POST';
 this.ajaxOpts.data = this.RASONModel;
 $.ajax(this.RASONServer + '/model', this.ajaxOpts).done(this.getModelId).fail(this.invokeFailed);
 }
 } else {
 alert("No model specified");
 }
},

```

See [Optimization.html](#) for contents of full script.

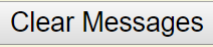
When  is clicked in the example, the endpoint, DELETE <server\_address>/api/model/id/delete is called to delete the model in the stopModel function. Deleting a model resource also removes any data files that were uploaded when it was created. Afterwards, the stopped function displays "Model stopped" in the Messages window.

```

// The following 2 functions demonstrate stopping a model that is currently being solved by the RASON API.
stopModel: function () {
 if (this.RASONModelID) {
 rasonApp.solveStopped = true;
 this.ajaxOpts.method = 'POST';
 // We are cleaning up in this case. Otherwise the 'done' callback could be omitted
 $.ajax(this.RASONModelID + '/stop', this.ajaxOpts).done(this.stopped).fail(this.invokeFailed);
 }
},

stopped: function (data, textStatus, jqXHR) {
 var resultHtml = 'Model stopped.';
 updateMessages(resultHtml);
 rasonApp.deleteModel();
}
};

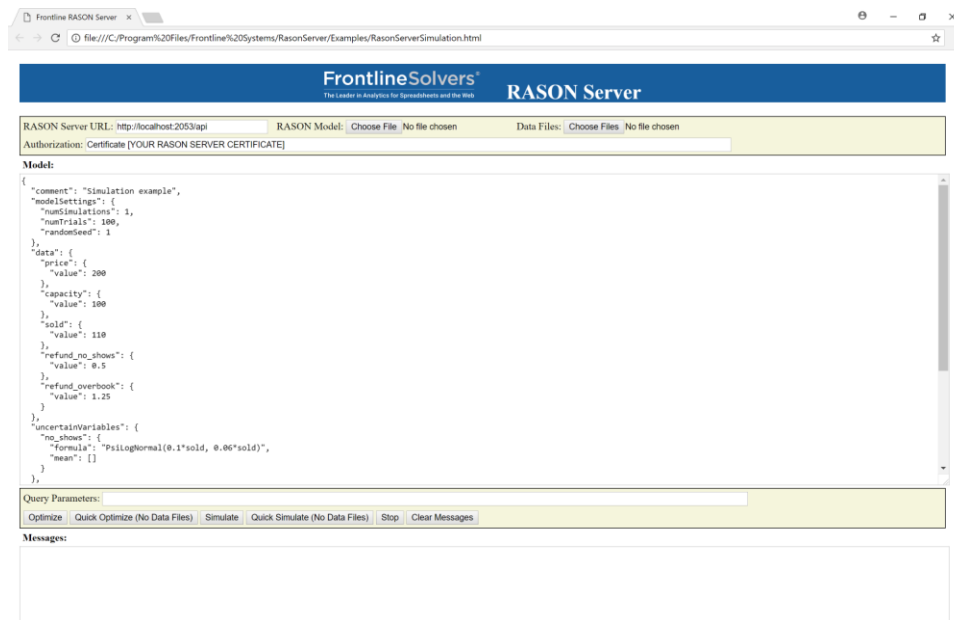
```

Click  to clear any output appearing in the Messages window.

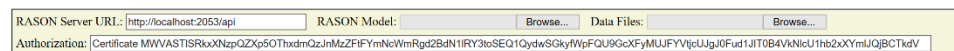
## Simulation Example

This section will step through an example that uses the Rason Rest API to run and obtain results from a simulation model. To start, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Client\Examples\REST and open Simulation.html to open an example web page. First, we will look at the "front end" of the example and later we will open this file using a text editor to view the code and the actual Rason REST API endpoints.





This example will reuse the certificate generated in the example above.



Within the Model field, we see the actual Rason simulation model that will be solved via the Rason Server. Scroll down to view the entire model.

## Setting Up the Rason Model

Building a simulation model using the Rason modeling language is much like building an optimization model. As we learned above, in an optimization model, our Rason model contains components such as variables, data, constraints, and objective. In a simulation model, our Rason model will contain components such as data, uncertainVariables and uncertainFunctions. You will use Psi Distribution functions to model the uncertainty as *inputs* and identify the *outputs* of special interest (such as Total Revenue or Net Profit) to examine or summarize how they behave in light of the uncertainty.

When setting up a simulation model in the RASON Modeling language, your model will typically contain these essential segments:

**data:** Where arrays to be used in the calculation of uncertain function and/or uncertain variables will be defined.

**uncertainVariables:** Where the uncertain variables will be defined.

**uncertainFunctions:** Where the uncertain functions will be defined.

As mentioned above, within this overall framework, you have a great deal of flexibility in choosing how to define and calculate your uncertain variables and uncertain functions. For example, the uncertain function will ultimately depend on the uncertain variables, but you don't have to calculate the entire function all at once. You can use any number of calculations in the data section to compute intermediate results and use these to calculate the uncertain function. For more information on defining a simulation model using the Rason modeling language, see the Rason User Guide.

When creating any RASON model, our model definition begins with a simple open bracket.

```
{
 "comment": "Simulation example",
```

```

"modelSettings": {
 "numSimulations": 1,
 "numTrials": 100,
 "randomSeed": 1
},
"data": {
 "price": {
 "value": 200
 },
 "capacity": {
 "value": 100
 },
 "sold": {
 "value": 110
 },
 "refund_no_shows": {
 "value": 0.5
 },
 "refund_overbook": {
 "value": 1.25
 }
},
"uncertainVariables": {
 "no_shows": {
 "formula": "PsiLogNormal(0.1*sold, 0.06*sold)",
 "mean": []
 }
},
"formulas": {
 "show_ups": {
 "formula": "sold - Round(no_shows, 0)"
 },
 "overbook": {
 "formula": "Max(0, show_ups - capacity)"
 }
},
"uncertainFunctions": {
 "revenue": {
 "formula": "price*(sold - refund_no_shows*Round(no_shows, 0) - refund_overbook*overbook)",
 "percentiles": []
 }
}
}

```

The example above simulates an airline ticket model that oversells airline seats in order to maximize the number of passengers on board each time the airline take flight. If there are more passengers than seats, an "overbooking refund" may be paid to selected passengers at a rate of 125% of the original ticket price. If a passenger is a "no show", a refund of only 50% per ticket is applicable. This number of "no shows" is modeled using the PsiLogNormal distribution. Given this information, the RASON model simulates the total Revenue obtained for the flight based on 100 tickets sold.

## Running a Simulation

As with optimization models, simulation models written in the Rason modeling language calling the Rason Server can be simulated in two different ways: 1. Using the Rason REST Server Quick Call POST <server\_address>/api/simulate or 2. Calling a series of standard REST Server endpoints such

as GET `<server_address>/api/model/id/simulate` to start a simulation, GET `<server_address>/api/model/id` to obtain the model ID, GET `<server_address>/api/model/id/status` to check the progress of the simulation and GET `<server_address>/api/model/id/result` to get the results of the simulation, which will always be valid JSON. Let's run a simulation using both methods using the Simulate and Quick Simulate command buttons in the Simulation example.

Click [Quick Simulate \(No Data Files\)](#) to call the Quick Solve simulation endpoint.

Just like in the Optimization example, this REST API endpoint runs the simulation and immediately returns a Result, in JSON, as shown in the screenshot below.

```

Message:
Quick simulate started at: Tue Mar 20 2018 10:12:20 GMT-0700 (Pacific Daylight Time)
Model finished in: 888 milliseconds.
{
 "status": {
 "code": 0,
 "message": "Solver has completed the simulation."
 },
 "reportsFunctions": {
 "reports": [
 "percentiles": [
 {
 "percentile": 10000,
 "value": 10700,
 "lower": 10300,
 "upper": 11000,
 "mean": 10600,
 "stddev": 2000,
 "min": 10000,
 "max": 12000,
 "median": 10500,
 "q1": 10200,
 "q3": 10800,
 "mode": 10700,
 "range": 2000,
 "skewness": 0.0000000000000001,
 "kurtosis": 0.0000000000000001,
 "entropy": 0.0000000000000001,
 "variance": 4000000,
 "covariance": 0,
 "correlation": 0,
 "rho": 0,
 "tau": 0,
 "alpha": 0,
 "beta": 0,
 "gamma": 0,
 "delta": 0,
 "epsilon": 0,
 "zeta": 0,
 "eta": 0,
 "theta": 0,
 "iota": 0,
 "kappa": 0,
 "lambda": 0,
 "mu": 0,
 "nu": 0,
 "xi": 0,
 "omicron": 0,
 "pi": 0,
 "rho": 0,
 "sigma": 0,
 "tau": 0,
 "upsilon": 0,
 "phi": 0,
 "chi": 0,
 "psi": 0,
 "omega": 0,
 "v": 0,
 "w": 0,
 "x": 0,
 "y": 0,
 "z": 0,
 "a": 0,
 "b": 0,
 "c": 0,
 "d": 0,
 "e": 0,
 "f": 0,
 "g": 0,
 "h": 0,
 "i": 0,
 "j": 0,
 "k": 0,
 "l": 0,
 "m": 0,
 "n": 0,
 "o": 0,
 "p": 0,
 "q": 0,
 "r": 0,
 "s": 0,
 "t": 0,
 "u": 0,
 "v": 0,
 "w": 0,
 "x": 0,
 "y": 0,
 "z": 0,
 "aa": 0,
 "ab": 0,
 "ac": 0,
 "ad": 0,
 "ae": 0,
 "af": 0,
 "ag": 0,
 "ah": 0,
 "ai": 0,
 "aj": 0,
 "ak": 0,
 "al": 0,
 "am": 0,
 "an": 0,
 "ao": 0,
 "ap": 0,
 "aq": 0,
 "ar": 0,
 "as": 0,
 "at": 0,
 "au": 0,
 "av": 0,
 "aw": 0,
 "ax": 0,
 "ay": 0,
 "az": 0,
 "ba": 0,
 "bb": 0,
 "bc": 0,
 "bd": 0,
 "be": 0,
 "bf": 0,
 "bg": 0,
 "bh": 0,
 "bi": 0,
 "bj": 0,
 "bk": 0,
 "bl": 0,
 "bm": 0,
 "bn": 0,
 "bo": 0,
 "bp": 0,
 "bq": 0,
 "br": 0,
 "bs": 0,
 "bt": 0,
 "bu": 0,
 "bv": 0,
 "bw": 0,
 "bx": 0,
 "by": 0,
 "bz": 0,
 "ca": 0,
 "cb": 0,
 "cc": 0,
 "cd": 0,
 "ce": 0,
 "cf": 0,
 "cg": 0,
 "ch": 0,
 "ci": 0,
 "cj": 0,
 "ck": 0,
 "cl": 0,
 "cm": 0,
 "cn": 0,
 "co": 0,
 "cp": 0,
 "cq": 0,
 "cr": 0,
 "cs": 0,
 "ct": 0,
 "cu": 0,
 "cv": 0,
 "cw": 0,
 "cx": 0,
 "cy": 0,
 "cz": 0,
 "da": 0,
 "db": 0,
 "dc": 0,
 "dd": 0,
 "de": 0,
 "df": 0,
 "dg": 0,
 "dh": 0,
 "di": 0,
 "dj": 0,
 "dk": 0,
 "dl": 0,
 "dm": 0,
 "dn": 0,
 "do": 0,
 "dp": 0,
 "dq": 0,
 "dr": 0,
 "ds": 0,
 "dt": 0,
 "du": 0,
 "dv": 0,
 "dw": 0,
 "dx": 0,
 "dy": 0,
 "dz": 0,
 "ea": 0,
 "eb": 0,
 "ec": 0,
 "ed": 0,
 "ee": 0,
 "ef": 0,
 "eg": 0,
 "eh": 0,
 "ei": 0,
 "ej": 0,
 "ek": 0,
 "el": 0,
 "em": 0,
 "en": 0,
 "eo": 0,
 "ep": 0,
 "eq": 0,
 "er": 0,
 "es": 0,
 "et": 0,
 "eu": 0,
 "ev": 0,
 "ew": 0,
 "ex": 0,
 "ey": 0,
 "ez": 0,
 "fa": 0,
 "fb": 0,
 "fc": 0,
 "fd": 0,
 "fe": 0,
 "ff": 0,
 "fg": 0,
 "fh": 0,
 "fi": 0,
 "fj": 0,
 "fk": 0,
 "fl": 0,
 "fm": 0,
 "fn": 0,
 "fo": 0,
 "fp": 0,
 "fq": 0,
 "fr": 0,
 "fs": 0,
 "ft": 0,
 "fu": 0,
 "fv": 0,
 "fw": 0,
 "fx": 0,
 "fy": 0,
 "fz": 0,
 "ga": 0,
 "gb": 0,
 "gc": 0,
 "gd": 0,
 "ge": 0,
 "gf": 0,
 "gg": 0,
 "gh": 0,
 "gi": 0,
 "gj": 0,
 "gk": 0,
 "gl": 0,
 "gm": 0,
 "gn": 0,
 "go": 0,
 "gp": 0,
 "gq": 0,
 "gr": 0,
 "gs": 0,
 "gt": 0,
 "gu": 0,
 "gv": 0,
 "gw": 0,
 "gx": 0,
 "gy": 0,
 "gz": 0,
 "ha": 0,
 "hb": 0,
 "hc": 0,
 "hd": 0,
 "he": 0,
 "hf": 0,
 "hg": 0,
 "hh": 0,
 "hi": 0,
 "hj": 0,
 "hk": 0,
 "hl": 0,
 "hm": 0,
 "hn": 0,
 "ho": 0,
 "hp": 0,
 "hq": 0,
 "hr": 0,
 "hs": 0,
 "ht": 0,
 "hu": 0,
 "hv": 0,
 "hw": 0,
 "hx": 0,
 "hy": 0,
 "hz": 0,
 "ia": 0,
 "ib": 0,
 "ic": 0,
 "id": 0,
 "ie": 0,
 "if": 0,
 "ig": 0,
 "ih": 0,
 "ii": 0,
 "ij": 0,
 "ik": 0,
 "il": 0,
 "im": 0,
 "in": 0,
 "io": 0,
 "ip": 0,
 "iq": 0,
 "ir": 0,
 "is": 0,
 "it": 0,
 "iu": 0,
 "iv": 0,
 "iw": 0,
 "ix": 0,
 "iy": 0,
 "iz": 0,
 "ja": 0,
 "jb": 0,
 "jc": 0,
 "jd": 0,
 "je": 0,
 "jf": 0,
 "jg": 0,
 "jh": 0,
 "ji": 0,
 "jj": 0,
 "jk": 0,
 "jl": 0,
 "jm": 0,
 "jn": 0,
 "jo": 0,
 "jp": 0,
 "jq": 0,
 "jr": 0,
 "js": 0,
 "jt": 0,
 "ju": 0,
 "jv": 0,
 "jw": 0,
 "jx": 0,
 "jy": 0,
 "jz": 0,
 "ka": 0,
 "kb": 0,
 "kc": 0,
 "kd": 0,
 "ke": 0,
 "kf": 0,
 "kg": 0,
 "kh": 0,
 "ki": 0,
 "kj": 0,
 "kk": 0,
 "kl": 0,
 "km": 0,
 "kn": 0,
 "ko": 0,
 "kp": 0,
 "kq": 0,
 "kr": 0,
 "ks": 0,
 "kt": 0,
 "ku": 0,
 "kv": 0,
 "kw": 0,
 "kx": 0,
 "ky": 0,
 "kz": 0,
 "la": 0,
 "lb": 0,
 "lc": 0,
 "ld": 0,
 "le": 0,
 "lf": 0,
 "lg": 0,
 "lh": 0,
 "li": 0,
 "lj": 0,
 "lk": 0,
 "ll": 0,
 "lm": 0,
 "ln": 0,
 "lo": 0,
 "lp": 0,
 "lq": 0,
 "lr": 0,
 "ls": 0,
 "lt": 0,
 "lu": 0,
 "lv": 0,
 "lw": 0,
 "lx": 0,
 "ly": 0,
 "lz": 0,
 "ma": 0,
 "mb": 0,
 "mc": 0,
 "md": 0,
 "me": 0,
 "mf": 0,
 "mg": 0,
 "mh": 0,
 "mi": 0,
 "mj": 0,
 "mk": 0,
 "ml": 0,
 "mm": 0,
 "mn": 0,
 "mo": 0,
 "mp": 0,
 "mq": 0,
 "mr": 0,
 "ms": 0,
 "mt": 0,
 "mu": 0,
 "mv": 0,
 "mw": 0,
 "mx": 0,
 "my": 0,
 "mz": 0,
 "na": 0,
 "nb": 0,
 "nc": 0,
 "nd": 0,
 "ne": 0,
 "nf": 0,
 "ng": 0,
 "nh": 0,
 "ni": 0,
 "nj": 0,
 "nk": 0,
 "nl": 0,
 "nm": 0,
 "nn": 0,
 "no": 0,
 "np": 0,
 "nq": 0,
 "nr": 0,
 "ns": 0,
 "nt": 0,
 "nu": 0,
 "nv": 0,
 "nw": 0,
 "nx": 0,
 "ny": 0,
 "nz": 0,
 "oa": 0,
 "ob": 0,
 "oc": 0,
 "od": 0,
 "oe": 0,
 "of": 0,
 "og": 0,
 "oh": 0,
 "oi": 0,
 "oj": 0,
 "ok": 0,
 "ol": 0,
 "om": 0,
 "on": 0,
 "oo": 0,
 "op": 0,
 "oq": 0,
 "or": 0,
 "os": 0,
 "ot": 0,
 "ou": 0,
 "ov": 0,
 "ow": 0,
 "ox": 0,
 "oy": 0,
 "oz": 0,
 "pa": 0,
 "pb": 0,
 "pc": 0,
 "pd": 0,
 "pe": 0,
 "pf": 0,
 "pg": 0,
 "ph": 0,
 "pi": 0,
 "pj": 0,
 "pk": 0,
 "pl": 0,
 "pm": 0,
 "pn": 0,
 "po": 0,

```

The result "code": 0 means that the simulation completed successfully. Since we asked for the percentile values in the Rason model, the percentiles are listed beneath the result code under `uncertainFunctions` in the result.

Again, as discussed above, the `quickSolve` endpoint should only be used with models that solve very quickly since your application will not be responsive until a result is received. In addition, Quick Solve endpoints do not create a "model resource" which means that a model solved by calling `POST /reason/net/api/simulate` cannot be saved, edited, retrieved or solved at a later time.

To solve the simulation model using the standard Rason REST API endpoints, click **Simulate**. A series of calls will be made to the Rason Rest API:

- GET `<server_address>/api/model/id/simulate` to start a simulation,
- GET `<server_address>/api/model/id` to obtain the model ID,
- GET `<server_address>/api/model/id/status` to check the progress of the simulation,
- and GET `<server_address>/api/model/id/result` to get the results of the simulation, which will always be valid JSON.

```
Messages:
simulate started at: Fri Mar 16 2024 16:57:17 GMT-0700 (Pacific Daylight Time)
workspaceID: http://w.1.1.157/2024/api/node/ac1a1b0payspaysgradiusmo
checking status...
Status: In Queue
Status: Complete
Model finished in: 5194 milliseconds
Result:
{
 "status": {
 "code": 0,
 "codeText": "Solver has completed the simulation."
 },
 "insertIntoFunctions": {
 "parameters": {
 "L0702": 14702,
 "L0802": 15002,
 "L0902": 15002
 }
 }
}
```

Note: Calling `GET <server_address>/api/model/id/status` obtains the status of the simulation. If the simulation is complete, the result returned will be:

```

{"status": "Complete" }

```

*Status:* Complete means that the simulation has been completed. If the simulation was still in progress we would have received an incomplete status with intermediate results, for example: `{"status": "Incomplete", "trials" : 90192 }`

Open `Simulation.html` using a text editor such as WordPad or Notepad. Apart from the Rason model appearing in the Model window, this example includes the same backend code as the previous optimization example.

Since the code in this example is the same as the code in the optimization example, let's jump down to the QuickSimulate function. When **Quick Simulate (No Data Files)** is clicked, the QuickSimulate function is executed. As with the quickSolve function, the quickSimulate function calls the quick solve Rason REST API endpoint POST <server\_address>/api/simulate.

```
quickSimulate: function () {
 this.RASONModel = $('#_userModel').val();
 if(this.RASONModel){
 this.appStatus = "executing";
 this.startTime = new Date();
 updateMessages("Quick Simulate started at: " + this.startTime.toString());
 var queryParams = $('#_queryParams').val();
 this.ajaxOpts.method = 'POST';
 this.ajaxOpts.data = this.RASONModel;
 $.ajax(this.RASONServer + '/simulate' + queryParams, this.ajaxOpts).done(this.quickStatus).fail(this.invokeFailed);
 } else {
 alert("No model specified");
 }
},
```

When **Simulate** is clicked, the solving process begins with the function startSimulate.

1. First, the model is posted to the Rason Server by calling GET <server\_address>/api/model/id/simulate.
2. GET <server\_address>/api/model/id is called to obtain the model's ID.
3. GET <server\_address> api/model/id/status is called to monitor the progress of the solve.
4. Final results are obtained by calling GET <server\_address>/api/model/id/result.
5. The model is deleted by calling DELETE <server\_address>/api/model/id/delete.

```
// Entry point - the 'Simulate' button was clicked.
startSimulate: function () {
 this.RASONModel = $('#_userModel').val();
 if(this.RASONModel){
 this.appStatus = "executing";
 this.startTime = new Date();
 updateMessages("Simulate started at: " + this.startTime.toString());
 this.RASOEntry = '/simulate';
 if(this.modelDataFiles.length > 0){
 this.submitModelWithDataFiles();
 } else {
 this.ajaxOpts.method = 'POST';
 this.ajaxOpts.data = this.RASONModel;
 $.ajax(this.RASONServer + '/model', this.ajaxOpts).done(this.getModelId).fail(this.invokeFailed);
 }
 } else {
 alert("No model specified");
 }
},

submitModelWithDataFiles: function(){
 var formData = new FormData(), me = this, request = new XMLHttpRequest();
 formData.append("RasonModel", this.RASONModel);
 for(var i = 0; i < this.modelDataFiles.length; ++i){
 formData.append(this.modelDataFiles[i].name, this.modelDataFiles[i], this.modelDataFiles[i].name);
 }

 request.onreadystatechange = function(){
 if (this.readyState == 4) {
 if (this.status == 200 || this.status == 201) {
 // model has been posted to the RASON server. enqueue it for processing
 me.getModelId(this.response, this.statusText, this);
 } else {
 me.invokeFailed(this, this.statusText, null);
 }
 }
 };
 request.open("POST", this.RASONServer + '/model');
 request.setRequestHeader('Authorization', this.ajaxOpts.headers['Authorization']);
 request.send(formData);
},
```

```

// Once the model has been posted to the RASON server, get the model ID in order to track
// it's 'solve' progress.
getModelId: function (data, textStatus, jqXHR) {
 rasonApp.RASONModelID = jqXHR.getResponseHeader('location');
 // the model has been posted, there is no longer a need to send this data
 rasonApp.ajaxOpts.data = null;
 if (rasonApp.RASONModelID) {
 updateMessages('RASONModelID: ' + rasonApp.RASONModelID + '\nChecking status...');
 // we have a model ID, move to the next step
 setTimeout(function () {
 rasonApp.ajaxOpts.method = 'GET';
 // **** Note that the model has only been posted at this point. Below is the REST API invocation that begins solving the model. *****
 // Get and add the query parameters to the request. They must be correctly entered in the input text box.
 var queryParams = $('#_queryParams').val();
 $.ajax(rasonApp.RASONModelID + rasonApp.RASONEntry + queryParams, rasonApp.ajaxOpts)
 .done(function(data, textStatus, jqXHR){rasonApp.status(data, textStatus, jqXHR);})
 .fail(function(jqXHR, stat, err){rasonApp.invokeFailed(jqXHR, stat, err);});
 }, rasonApp.timeoutTime);
 } else {
 alert("No model ID");
 }
},
// this function and the one below, 'checkStatus', form a loop that checks the model status on the RASON server.
status: function (data, textStatus, jqXHR) {
 rasonApp.ajaxOpts.method = 'GET';
 $.ajax(rasonApp.RASONModelID + '/status', rasonApp.ajaxOpts).done(rasonApp.checkStatus).fail(rasonApp.invokeFailed);
},
// We are looking for a status of 'Complete'. If the model is not solved (Complete), then we will wait 'timeoutTime'
// seconds and check again. Of course, this is all subject to whether or not the 'Stop' button was clicked.
checkStatus: function (data, textStatus, jqXHR) {
 updateMessages('Status: ' + (data && data.status ? data.status : 'Unknown'));
 if (data && data.status == "Complete") {
 updateMessages("Model finished in: " + (Date.now() - rasonApp.startTime - rasonApp.timeoutTime) + " milliseconds");
 rasonApp.ajaxOpts.method = 'GET';
 // the model is complete. Now get the results
 $.ajax(rasonApp.RASONModelID + '/result', rasonApp.ajaxOpts).done(rasonApp.results).fail(rasonApp.invokeFailed);
 } else if (!rasonApp.solveStopped) {
 setTimeout(function () {
 rasonApp.status();
 }, rasonApp.timeoutTime);
 }
},
// Print the results as a JSON string, showing the structure of the results object returned by the RASON server
results: function (data, textStatus, jqXHR) {
 var resultHTML = 'Result:\n' + JSON.stringify(data, null, 2);
 updateMessages(resultHTML);
 rasonApp.deleteModel();
},

```

## Simulation Optimization Example

This section will step through an example that uses the Rason Rest API to run and obtain results from a simulation optimization model. To start, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Client\Examples\REST and open ProjectSelect.html to open an example web page. First, we will look at the "front end" of the example and later we will open this file using a text editor to view the code and the actual Rason REST API endpoints.

FrontlineSolvers®

The Leader in Analytics for Spreadsheets and the Web

RASON Server

RASON Server URL:

http://localhost:2053/api

RASON Model:

Choose File No file chosen

Data Files:

Choose Files No file chosen

Count:

1

Authorization: Certificate ZVleJxBdHxkDXJM2I4eDM5YXQzUkx\_R1A2aEH0OVhDMnEyPURaWE4lINzaU1vUJEa09hYTZlNUJyYjFBRExhWkBBRUhlcTJ3aFVmdjZJOExYV96YzN2

Model:

```

{
 "comment": "This model has been generated by Psi from an Excel model in the workbook ProjectSelect(SimOpt).xlsx",
 "modelName": "ProjectSelect(SimOpt)",
 "modelDescription": "",
 "modelSettings": {
 "worksheets": [
 "Project_Selection"
],
 "activeSheet": "Project_Selection",
 "simulationOptimization": true,
 "randomSeed": 123,
 "randomGenerator": "Lecuyer",
 "samplingMethod": "Latin"
 },
 "engineSettings": {
 "engine": "Evolutionary",
 "numThreads": 1,
 "scaling": -1
 },
 "variables": {
 "I18:I25": {
 "value": 0,
 "type": "bin",
 "lower": 0,
 "upper": 1,
 "finalValue": []
 }
 },
 "uncertainVariables": {
 "d18": {

```

Query Parameters:

Optimize

Quick Optimize (No File Attachments)

Simulate

Quick Simulate (No File Attachments)

Stop

Clear Messages

This example will reuse the certificate generated in the Optimization.html example above.

Within the Model field, we see the actual Rason data model that will be solved via the Rason Server. Scroll down to view the entire model.

## Setting Up the Rason Model

Building a simulation optimization model using the Rason modeling language is much like building an optimization or simulation model. As we learned above, in an optimization or simulation model, our Rason model contains components such as variables, data, constraints, and objective. In a simulation optimization model, the RASON model will contain variables, constraints, and objective components as well as uncertain variables and uncertain functions components.

The example below uses simulation optimization to select the optimal projects to implement in a year in order to maximize ROI given the uncertainties of project success and cash flow.

This model has been generated using Analytic Solver's, Frontline's addin for Excel, RASON deployment functionality that generates a RASON model from your existing Excel model. For more information on Analytic Solver and this functionality, see the Analytic Solver User Guide.

The beginning of the RASON model simply selects the Evolutionary engine to solve the model and also turns simulation optimization to "true", random generator to "lecuyer" and sampling method to "latin".

```
{
 "comment": "This model has been generated by Psi from an Excel
 model in the workbook ProjectSelect(SimOpt).xlsx",
 "modelName": "ProjectSelect(SimOpt)",
 "modelDescription": "",
 "modelSettings": {
 "worksheets": ["Project_Selection"],
 "activeSheet": "Project_Selection",
 "simulationOptimization": true,
 "randomSeed": 123,
 "randomGenerator": "Lecuyer",
 "samplingMethod": "Latin"
 },
 "engineSettings": {
 "engine": "Evolutionary",
 "numThreads": 1,
 "scaling": -1
 },
}
```

- There are 8 decision variables defined within the "variables" section of the RASON model, I18:I25. These variables are defined as binary variables.

```
"variables": {
 "i18:i25": {
 "value": 1,
 "type": "bin",
 "lower": 0,
 "upper": 1,
 "finalValue": []
 }
},
```

- There are 16 different uncertain variables defined within the "uncertainVariables" section of the RASON model.

Uncertain Variables defined as D18, D19, D20, D21, D22, D23, D24 and D25 define each project's projected cash flow using the PsiTriangular distribution using minimum, maximum and most likely parameters.

Uncertain variables defined as E18, E19, E20, E21, E22, E23, E24 and E25 define the probability of success using the "success" parameter of 90%, defined within the data section for "e15".

```

"uncertainVariables": {
 "d18": {
 "formula": "PsiTriangular(400000,500000,900000)"
 },
 "e18": {
 "formula": "PsiBinomial(1,E15)"
 },
 "d19": {
 "formula": "PsiTriangular(500000,750000,1250000)"
 },
 "e19": {
 "formula": "PsiBinomial(1,E15)"
 },
 "d20": {
 "formula": "PsiTriangular(500000,1000000,1500000)"
 },
 "e20": {
 "formula": "PsiBinomial(1,E15)"
 },
 "d21": {
 "formula": "PsiTriangular(400000,600000,900000)"
 },
 "e21": {
 "formula": "PsiBinomial(1,E15)"
 },
 "d22": {
 "formula": "PsiTriangular(250000,500000,750000)"
 },
 "e22": {
 "formula": "PsiBinomial(1,E15)"
 },
 "d23": {
 "formula": "PsiTriangular(300000,500000,600000)"
 },
 "e23": {
 "formula": "PsiBinomial(1,E15)"
 },
 "d24": {
 "formula": "PsiTriangular(200000,450000,700000)"
 },
 "e24": {
 "formula": "PsiBinomial(1,E15)"
 },
 "d25": {
 "formula": "PsiTriangular(400000,500000,700000)"
 },
 "e25": {
 "formula": "PsiBinomial(1,E15)"
 }
},

```

- The initial investment required for each project is located within the "data" section defined as G18, G19, G20, G21, G22, G23, G24 and G25.

```

"data": {
 "e15": {

```

```

 "value": 0.9
 },
 "g18:g25": {
 "value": [
 325000,
 450000,
 550000,
 300000,
 150000,
 250000,
 150000,
 325000
]
 },
 },
},

```

- Inside of the "formulas" section of the RASON Model, intermediary formulas are calculated.

F18, F19, F20, F21, F22, F23, F24 and F25 calculate the Expected Cash Flow from each project by multiplying the cash flow from each project (uncertain variables in cells D18:D25) with the Chance of Success (uncertain variables in cells E18:E25).

H18, H19, H20, H21, H22, H23, H24 and H25 calculate the Expected Cash Flow – Investment by subtracting the initial investment from the Expected Cash Flow.

```

"formulas": {
 "f18": {
 "formula": "D18*E18"
 },
 "h18": {
 "formula": "F18-G18"
 },
 "f19": {
 "formula": "D19*E19"
 },
 "h19": {
 "formula": "F19-G19"
 },
 "f20": {
 "formula": "D20*E20"
 },
 "h20": {
 "formula": "F20-G20"
 },
 "f21": {
 "formula": "D21*E21"
 },
 "h21": {
 "formula": "F21-G21"
 },
 "f22": {
 "formula": "D22*E22"
 },
 "h22": {
 "formula": "F22-G22"
 },
 "f23": {
 "formula": "D23*E23"
 },

```



```

 },
 "h23": {
 "formula": "F23-G23"
 },
 "f24": {
 "formula": "D24*E24"
 },
 "h24": {
 "formula": "F24-G24"
 },
 "f25": {
 "formula": "D25*E25"
 },
 "h25": {
 "formula": "F25-G25"
 }
 },

```

- There is one uncertain Function which calculates the expected Total Cash Flow from all selected projects.

```

 "uncertainFunctions": {
 "g30": {
 "formula": "SUMPRODUCT(H18:H25,I18:I25)+PsiOutput()",
 "trials": [],
 "mean": []
 }
 },

```

- There is one constraint that ensures that the total budget is not exceeded.

```

 "constraints": {
 "g27": {
 "formula": "sumproduct(G18:G25, I18:I25)",
 "upper": "1500000"
 }
 },

```

- The objective function maximizes the expected total cash flow.

```

 "objective": {
 "g32": {
 "formula": "PsiMean(g30)",
 "type": "max",
 "finalValue": []
 }
 }
 }
}

```

## Running a Simulation Optimization Model

As with optimization and simulation models, simulation optimization models written in the Rason modeling language calling the Rason Server can be solved in two different ways: 1. Using the Rason REST Server Quick Call POST <server\_address>/api/optimize or 2. Calling a series of standard REST Server endpoints such as GET <server\_address>/api/model/id/optimize to start an optimization, GET <server\_address>/api/model/id to obtain the model ID, GET <server\_address>/api/model/id/status to check the progress of the solve and GET <server\_address>/api/model/id/result to get the results, which will always be valid JSON.

Note: Calling POST <server\_address>/api/simulate or GET <server\_address>/api/model/id/simulate will only perform a simulation, not a simulation optimization.

Open ProjectSelect.html using a text editor such as WordPad or Notepad. Apart from the Rason model appearing in the Model window, this example includes the same backend code as the previous optimization and simulation examples.

---

## Managing Solver SDK Dedicated Server

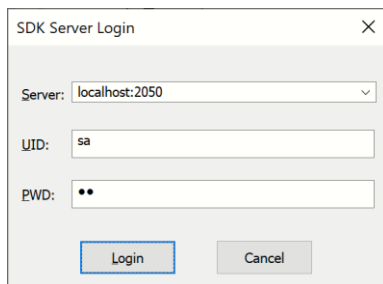
### Introduction

Read the section below to learn how to manage Solver SDK Dedicated Server by using the SAdmin executable installed into C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin. Using this program, you will be able to easily change settings, manage multiple servers, organize the queue of client requests, etc.

### Using Solver SDK Dedicated Server Administrative Program

The program SAdmin.exe is the administrative client for Solver SDK Dedicated Server. It allows users to manage multiple servers, organize the queue of client requests and solutions, etc. for the SDK Server service.

This executable is installed into C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin. Double click SAdmin.exe to start the SDK Server Admin program. The following dialog will appear.



#### Server

- By default, if Solver SDK Dedicated Server is running on the local machine at address: localhost:2053. This location can be changed in the [SolverServerN] section of the SolverServer.ini file. For more information on the SolverServer.ini file, please see the Settings section below.
- If the server is on a separate machine, then enter the name of the server or the server's IP address. For example: 10.1.1.1:2050 where "10.1.1.1" is the IP address of the server or Frontline:2050 where "Frontline" is the name of the server.
- If the server is on a website, such as [www.solver.com](http://www.solver.com), then enter [www.solver.com:2050](http://www.solver.com:2050).
- It is also possible to build Solver SDK Dedicated Server into your web server, please contact our product support department at [support@solver.com](mailto:support@solver.com) for more details.

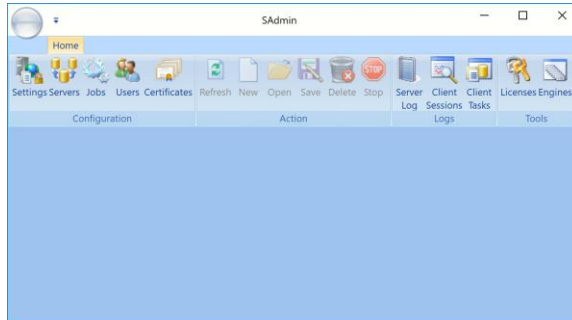
#### UID

- By default, enter "sa" (without the quotes) for your User ID. This can be changed to a more meaningful ID on the Users tab or from within Auth.ini.

## PWD

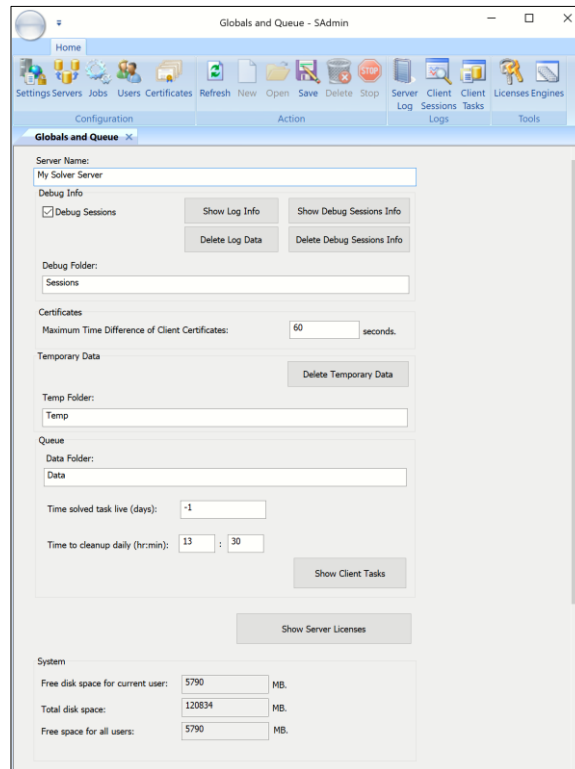
- By default, enter "sa" (without the quotes) for your password. This can also be changed to a more meaningful password on the Users tab or within Auth.ini.

Click **Login**. The following dialog will open.



## Settings

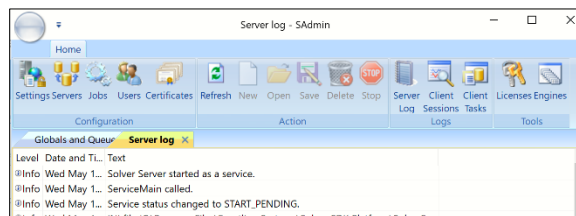
Click **Settings** on the Ribbon to open the Globals and Queue dialog.



On this dialog, you can see/change the global and queue settings.

- **Server Name:** This is the name of your Solver SDK Dedicated Server. By default, the server name is "My Solver Server". This can be changed to something more meaningful to the user.
- **Debug Info:**

- When **Debug Sessions** is checked, Solver SDK Dedicated Server will write out all communications (all calls from clients and all responses from server) to your hard disk in the Sessions folder located at C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin. Selecting this option will reduce the performance of Solver SDK Dedicated Server considerably and will consume considerable disk space. As a result, this option should only be checked when it is required that the Administrator examine user communications.
- To display the contents of the Sessions folder, click **Show Debug Sessions Info**. (Alternatively, the Debug log may also be displayed by clicking the **Client Sessions** icon on the Ribbon.) To change the name of the Sessions folder, simply type the new name under **Debug Folder**.
- Click **Show Log Info** to see a "short form" history of all actions performed by Solver SDK Dedicated Server.



- To delete the log history, click **Delete Log Data**. The log may also be displayed by clicking the **Server Log** icon in the Logs group on the Ribbon. To delete the Debug Sessions Info, click **Delete Debug Sessions Info**.
- **Certificates:**
  - When a certificate is issued, this certificate is valid for the number of seconds entered for Maximum Time Difference of Client Certificates.
- **Temporary Data:**
  - Solver SDK Dedicated Server uses this folder to store temporary data such as session logs.
- **Queue:**
  - The **Data Folder** (located at C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin\Data) holds input/output files from the queue. The order of processing is First in – First out.
  - The settings for the **Time solved task live (days)** and the **Time to cleanup daily (hr:min)** options determine the amount of time (in days) the solution files will remain "live" on Solver SDK Dedicated Server. The default setting of -1 and 13:30 for **Time solved task live** and **Time to cleanup daily**, respectively, indicates that the solution files will be deleted from the Data folder the following day at 1:30 pm (-1 = 1 day & 13:30 = 1:30 pm).
- Click **Show Client Tasks** to view all current tasks.
- Click **Show Server Licenses** to view all server licenses
- Statistics related to disk space can be found under **System**.
- Scroll down to view the current Licenses on the server.

| Licenses on the server:    |     |          |          |         |            |
|----------------------------|-----|----------|----------|---------|------------|
| Product                    | Ver | Days     | Location | Options | Parameters |
| ANALYTICSOLVERPLATFORM-... | 170 | 0 days   | Registry |         | vm=1       |
| SOLVERPLATFORMSDK          | 170 | 0 days   | Registry | I10     |            |
| SOLVERENGINES-TRIAL        | 170 | 0 days   | Registry | I10     |            |
| SOLVERPLATFORMSDK          | 170 | 276 days | File     | I10     | vm=0;      |
| SOLVERSERVERSDK            | 170 | 276 days | File     | I10     | vm=0;      |
| XLMINERSDK                 | 170 | 276 days | File     | I10     | vm=0;      |

To save any changed options, click **Save** in the Action group on the Ribbon. To set an option back to its default (before the option has been saved), click **Refresh**. All options set through this dialog can be viewed in the SolverServer.ini file located at C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin. If desired, this file can be manually edited by opening the file in a text editor.

SolverServer.ini is a text file containing configuration information for Solver SDK Dedicated Server. This file is divided into sections and keys.

[GLOBAL] – This section defines Solver SDK Dedicated Server's global configuration parameters. There is only one key in this section.

[AdminServer1] – This section defines the name, protocol and port for Solver SDK Dedicated Server.

[SolverServerN] – This section defines the Listening task, which is the object used by clients to communicate with Solver SDK Dedicated Server. SDK Server can manage an unlimited amount of client listening tasks. The name of each section must contain "SolverServer" followed by an incrementing integer such as SolverServer1, SolverServer2, SolverServer3, ... The allowed keys in this section are: Name, Protocol, Port, NumClients, and Desc. For a complete description of these options, see the Server section below.

[Queue] – The Queue object is a single object of Solver SDK Dedicated Server service that manages the client tasks posted by remote clients. This section specifies the configuration values of the Queue object and contains the following keys: DataFolder, MaxTimeTaskLive, CleanupTime, and MaxWaitTimeInQueueAfterRestart. **DataFolder** specifies the folder where the client data will be written. **MaxTimeTaskLive** specifies the maximum time a client task can remain live on the server after the model is solved. **CleanupTime** specifies the time Solver SDK Dedicated Server will be "cleaned" and **MaxWaitTimeInQueueAfterRestart** specifies the threshold for client tasks waiting to be processed after Solver SDK Dedicated Server Queue is rebuilt after a server restart. For a complete description of these options, see above.

[JobN] – Jobs are multiple objects that perform the work in solving client tasks. Each Job is an instance of Solver SDK Dedicated Server. "N" must be an incrementing integer such as Job1, Job2, Job3, etc. You may have an unlimited amount of Jobs. Each Job asks Queue for a client task to solve. The allowed keys in this section are: Name, Type, Priority, MaxEvalTime, and MaxEvalCachedPoints. For a description of these options, see the Jobs section below.

```

SolverServer.ini - Notepad
File Edit Format View Help
[GLOBAL]
Name=My RASON Server
DebugSessions=1
DebugFolder=Sessions
TempFolder=Temp

[AdminServer1]
Name=RASON Server Admin
Desc=RASON Server Admin on port 2050
Protocol=SOAP
Port=2050

[SolverServer1]
Name=My RASON (SOAP) Server
Protocol=SOAP
Port=2052
NumClients=5
Desc=RASON Server on port 2052, SOAP protocol
UserDataFolder=Data-2052
Active=1

[SolverServer2]
Name=My RASON (REST) Server
Protocol=REST
Port=2053
NumClients=5
Desc=RASON Server on port 2053, REST protocol
UserDataFolder=Data-2053
Active=1

[Queue]
DataFolder=Data
MaxTimeTaskLive=-1:13:30
CleanupTime=13:30
MaxWaitTimeInQueueAfterRestart=1:0:0

[Job1]
Name=Job1
Type=Local
Priority=0
MaxEvalTime=1200000
MaxEvalCachedPoints=200

[Job2]
Name=Job2
Type=Local
Priority=0
MaxEvalTime=1200000
MaxEvalCachedPoints=200

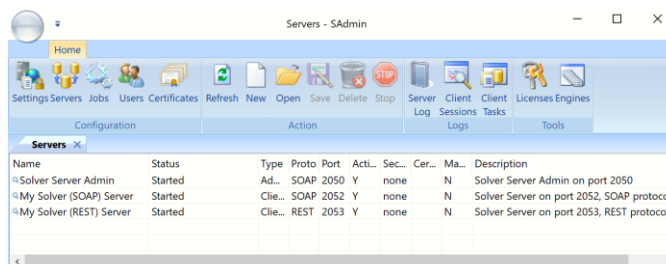
```

## Solver SDK Dedicated Server Workflow

A listening task is utilized when a client instantiates a problem or a Rason file is uploaded to the server. The listening task creates an instance for the client to communicate with the Queue object. When a client task is created, it is placed into the queue and waits until an idle Job accepts it. The order of processing the waiting tasks in the Queue object is "First In, First Out", or FIFO, type stack. The idle job object gets the client task from the Queue, solves the model, and saves the results back to the Queue object. When the client task is solved, the client can access the final results any time in the future. Each task is associated with a unique ID so that the client can use the task ID to obtain a model's final values.

## Servers

Click the **Servers** icon on the Ribbon to open the Servers dialog. This dialog displays all servers, both active and inactive, in Solver SDK Dedicated Server.



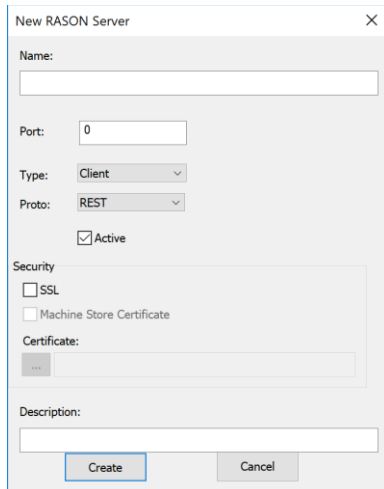
The dialog above shows both the Server machine (Solver Server Admin) and the Client machines (My Solver (REST) Server), their ports, their status (Started or Not Started), the Protocol used (SOAP or REST), whether the Server/Client is Active or not, the Security type, Certificate, and the location of the certificate. If "Y" is listed for MachineStore then the certificate is stored in the Machine Store. Otherwise, the certificate is stored in the User Store.

If you double click either machine (or click Open in the Action group on the Ribbon), the following dialog will appear. Here you can change the **Name**, **Description** and **Status** of the machine. Protocol and Security are disabled.

The Solver SDK Dedicated Server service offers two options for Security: SSL Security or No Security. To use SSL Security, you must have a Certificate authorized by a provider such as Verisign. When SSL is selected, the **Machine Store Certificate** checkbox and the **Certificate** field are enabled. Click the ... button to open the Server Security Certificates dialog and paste in the certificate.

Select **Machine** if the certificate is a Machine certificate and **User** if the certificate was issued to a specific user.

To add a new Server, click the **New** icon in the Action group on the Ribbon to display the following dialog.



The 'New RASON Server' dialog box contains the following fields and controls:

- Name:** A text input field for the server name.
- Port:** A text input field with '0' entered.
- Type:** A dropdown menu with 'Client' selected.
- Proto:** A dropdown menu with 'REST' selected.
- Active:** A checked checkbox.
- Security:** A section containing:
  - ☐ SSL
  - ☐ Machine Store Certificate
  - Certificate:** A text field with a '...' button to its left.
- Description:** A text input field.
- Buttons:** 'Create' and 'Cancel' buttons at the bottom.

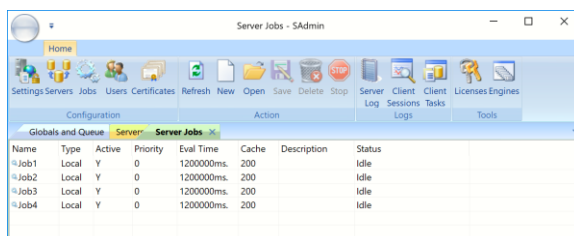
- **Name:** Enter the server name here.
- **Port:** Enter the port number where the server will run here.
- **Type:** Select either Client (for a client machine) or Admin (for a server) from the menu.
- **Proto:** Solver SDK Dedicated Server supports both SOAP and REST protocols.
- **Active:** Uncheck the Active checkbox to deactivate the machine.
- **Security:** As mentioned above, the Solver SDK Dedicated Server service offers two options for Security: SSL Security or No Security. To use SSL security, you must have a Certificate authorized by a provider such as Verisign. When SSL is selected, the **Machine Store Certificate** checkbox and the **Certificate** field are enabled. Click the ... button to open the Server Security Certificates dialog and paste in the certificate.
- **Description:** Enter a meaningful description here.

Click **Create** to create a new Server or **Cancel** to close the dialog without adding a new server.

To delete a server, simply select the server to be deleted and click the **Delete** icon in the Action Group on the Ribbon.

## Jobs

Click the **Jobs** icon on the Ribbon to open the Server Jobs dialog, as shown below.



The 'Server Jobs - SAdmin' dialog box shows a ribbon with 'Home', 'Configuration', and 'Action' tabs. The 'Server Jobs' tab is active, displaying a table of jobs.

| Name  | Type  | Active | Priority | Eval Time  | Cache | Description | Status |
|-------|-------|--------|----------|------------|-------|-------------|--------|
| %Job1 | Local | Y      | 0        | 1200000ms. | 200   |             | Idle   |
| %Job2 | Local | Y      | 0        | 1200000ms. | 200   |             | Idle   |
| %Job3 | Local | Y      | 0        | 1200000ms. | 200   |             | Idle   |
| %Job4 | Local | Y      | 0        | 1200000ms. | 200   |             | Idle   |

This dialog displays each Job and its properties in the queue. Double clicking Job1 will open the following dialog. (Alternatively, select Job1 and click the Open icon within the Action group on the Ribbon.) Here you can change the **Name** of the job, the **Priority**, the **Eval Time**, the number of **Cache Points**, change the status to Inactive and give the Job a **Description**.



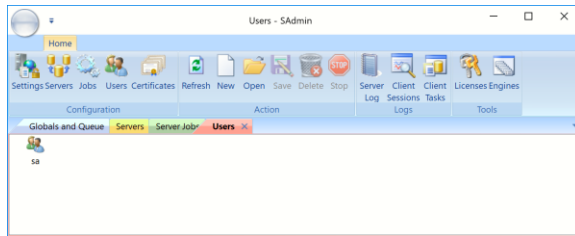
- **Name:** Enter a more meaningful name for the job here.
- **Type:** Currently the default type of Local is the only type permitted. This is subject to change in subsequent releases.
- **Priority:** Enter a value from 0-10 to designate the amount of "focus" time allotted to the Job by the operating system. The operating system will assign more time to a Job when a higher priority value is entered. Keep in mind, however, that when one Job is given a higher priority, the waiting time in the queue could increase dramatically as the "focus" will be given to the Job with the higher priority. Entering a value of 0 for this option is the same as entering a priority level of 4.
- **Eval Time:** This option is only utilized when an evaluator is present, such as `eval_type_function`, `eval_type_iteration`, `eval_type_simulation`, etc. (For a complete list of evaluators, please see the chapter "Solver SDK Evaluators".) When an evaluator is present, Solver SDK Dedicated Server will perturb the variable values and then ask the client machine to calculate the constraints and objective. This option controls the amount of time that a client machine must respond to the server. If this amount of time is exceeded, the server will return with a "user stopped" status. The minimum value for this option is 0 ms, the maximum value is 4,294,967,296 ms, and the default is 20,000 ms.
- **Cache Points:** This option is only utilized when Solver SDK Dedicated Server is used in conjunction with the Solver SDK and an evaluator is present, such as `eval_type_function`, `eval_type_iteration`, `eval_type_simulation` etc., or in conjunction with Analytic Solver and a nonlinear engine such as Standard GRG Nonlinear, Large Scale GRG, KNITRO, etc. This option controls the number of points in the cache allotted to the calculation of the evaluator. The more number of points allotted, the less calls to the client machine will be required. However, the more Cache points allotted for a specific job, the less cache will be available for the remaining jobs. The minimum value for this option is 0, the maximum value is 4,294,967,296, and the default is 200.
- **Active:** A job set to "inactive" (by unchecking the Active checkbox) will remain in the queue but will not be run.
- **Description:** Enter the job description here, if desired.

To create a new Job, click the **New** icon in the Action group on the Ribbon, change the desired option settings, if any, then click **Create** to create the Job or **Cancel** to close the dialog without creating a new Job.

To delete a job, simply select the job to be deleted and click the **Delete** icon in the Action Group on the Ribbon.

## Users

Click the **Users** icon on the Ribbon to open the Users dialog, pictured below.



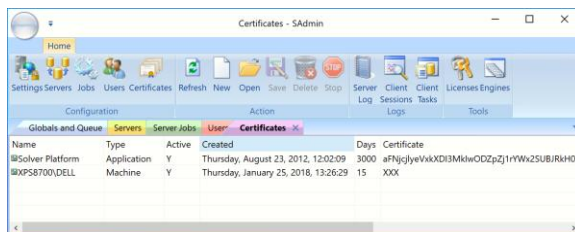
Double click the "sa" user to open the following dialog. Here you can change the **User Name**, **Password**, and change the user to "inactive" by unchecking the **Active** checkbox.

To create a new User, click the **New** icon in the Action group on the Ribbon, enter a User Name and password, then click **OK** to create the User or **Cancel** to close the dialog without creating a new User.

To delete a User, simply select the User to be deleted and click the **Delete** icon in the Action Group on the Ribbon.

## Certificates

Click the **Certificates** icon on the Ribbon to open the Certificates dialog, as shown below.

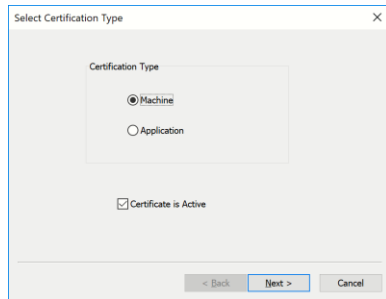


A certificate is a license to call Solver SDK Dedicated Server from a client machine. Both Application and Machine Certificates are supported. An application-based certificate gives permission to the client machine to run a specific application. Ten Application Certificates allowing an application to be accessed for 60 days are available with a trial license for any Solver product. For details on purchasing a Solver SDK Dedicated Server license or additional certificates, please contact our Sales Department at [Sales@solver.com](mailto:Sales@solver.com). You must use Solver SDK Dedicated Server to create the certificate, then include this certificate in your application.

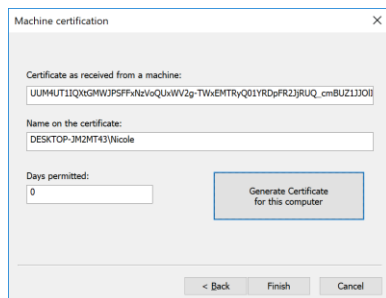
Machine based certificates allow Solver SDK to be run on an Excel workbook.

## Creating a New Certificate

First click the **New** icon in the Action Group on the Ribbon to open the following dialog.



Select either **Machine** or **Application** based on the desired certificate and confirm that the **Certificate is Active** checkbox is selected. Then click **Next** to display the following dialog.



Click **Generate Certificate for this computer** to fill in the **Certificate as received from a machine** and **Name on the certificate** fields. The **Days Permitted** field allows the Solver SDK Dedicated Server Administrator to enforce a time limit on the certificate, if desired. Enter "0" for a certificate that never expires.

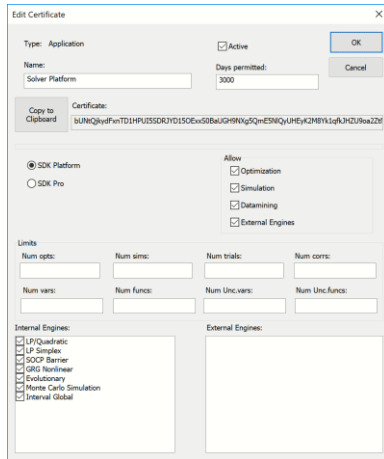
Click **Finish**. The Certificate will appear on the Certificates dialog.

To open the certificate, simply highlight and click the **Open** icon on the Ribbon. To delete the certificate, simply highlight and click the **Delete** icon on the Ribbon.

The certification key string created above must be included in your application. Without this certification key string, Solver SDK Dedicated Server will not function – and all product calls will return without performing any actions. It is important that you keep your certification key string confidential and use it only within one specific application. Frontline Systems will hold users responsible for the use of any additional applications that recognize your specific certification key string.

## Editing an Existing Certificate

To edit an existing certification key string, double click the certificate to open the following dialog. Here you can set the certificate to "Active" or "Inactive" by toggling the **Active** checkbox, change the **Name** of the certificate and the number of **Days Permitted**.

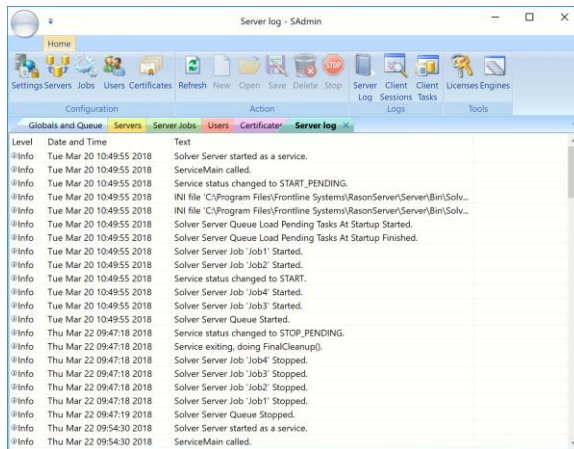


Various options may be set in this dialog such as the number of optimizations (**Num opts**) allowed in an optimization model, number of simulations (**Num sims**) allowed in a simulation model, number of trials (**Num trials**) performed in a simulation model, number of correlations (**Num Corrs**), number of decision variables (**Num vars**), number of functions -- objective and constraints for optimization or stochastic optimization models -- (**Num funcs**), number of uncertain variables (**Num Unc. vars**) and number of uncertain functions (**Num Unc. funcs**) for simulation and stochastic optimization models. Select **SDK Platform** to allow the use of Frontline's Solver SDK Platform or **SDK Pro** to use Frontline's Solver SDK Pro.

Select the names of specific internal and external Solver engines under **Internal Engines** and **External Engines**, respectively, to limit the use of Solver SDK to the engines selected.

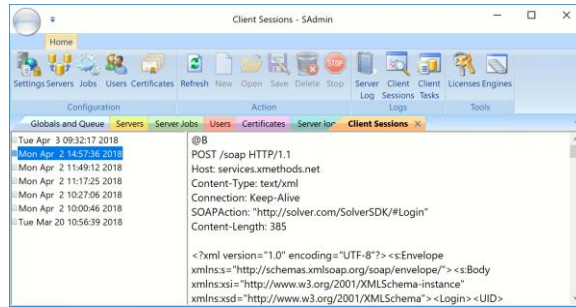
## Server Log

Click **Server Log** to view the contents of SolverServer.log. (SolverServer.log is the log file for Solver SDK Dedicated Server.) New events are appended to the bottom of the file.



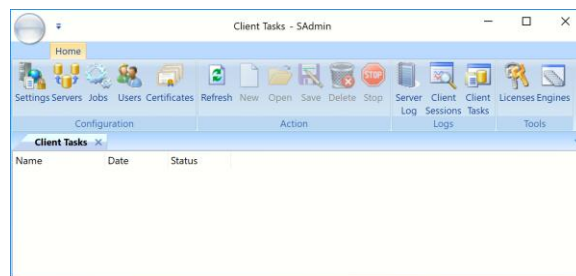
## Client Sessions

Click Client Sessions to view the log file for each client session. Highlight a session on the left of the dialog to view the full session log.



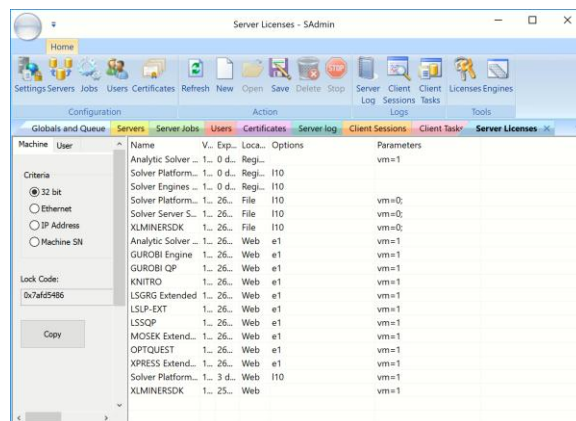
## Client Tasks

Click Client Tasks to see a list of Solver SDK Dedicated Server tasks currently being executed. To delete or stop a task, simply highlight the task and click Delete or Stop.



## Licenses

Click Licenses to view the Solver SDK Dedicated Server License Manager. Licenses running on Solver SDK Dedicated Server are displayed here. Client licenses will not appear here.



Web licenses running on Frontline Systems License Server may also be displayed if the current Solver SDK Dedicated Server User ID matches licenses running on this (Frontline Systems License) server. Click the User tab to view the current user. Click Logout to log out or Login to login with new user credentials.

Machine User

Name:  
Nicole Steidel

Email:  
nicole@solver.com

Phone:  
+1 702-953-9569

Company:  
Frontline systems

Login

Logout

Na  
An  
Sol  
Sol  
Sol  
Sol  
XL  
An  
GU  
GU  
KN  
LSC  
LSL  
LSS  
MC  
OP  
XPI  
Sol  
XL

## Engines

Click Engines to display any internal and external engine licenses available on Solver SDK Dedicated Server. Client licenses will not appear here.

Web licenses running on Frontline Systems License Server may also be displayed if the current SDK Server User ID matches licenses running on this (Frontline Systems License) server. Click Licenses -- Users to view the current user.

| Server Engines - Admin |           |
|------------------------|-----------|
| Configuration          |           |
| Action                 |           |
| Tools                  |           |
| Server Engines         |           |
| Name                   | File      |
| LP/Quadratic           | 261 days. |
| LP Simplex             | 260 days. |
| SQCP Barrier           | 261 days. |
| GRG Nonlinear          | 261 days. |
| Evolutionary           | 261 days. |
| Monte Carlo Simulation | 261 days. |

## Notable Files and Subfolders

This section gives a brief description of the files that appear inside of the SolverServer Bin folder (C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Server\Bin)

1. SolverSDK.wsdl file – This is the Web Service Description file for Solver SDK Dedicated Server SOAP calls.
2. SolverServer.exe – This is the executable of the Solver SDK Dedicated Server Service.
3. SolverServer.ini file - A text formatted file that contains the configuration information of Solver SDK Dedicated Server service. For more information on this file, see the Settings section above.
4. SolverServer.log - A text formatted file containing events written by Solver SDK Dedicated Server service.

Events could be informational, warnings or errors. Each text line in the log file contains the type of the event (I, W, E) followed by ':', the date and time of the event followed by ':' and description of the event.

5. Data subfolder - The subfolder, Data, is the folder where Solver SDK Dedicated Server writes the data for client requests. The folder is defined in the [Queue] section of SolverServer.ini in the DataFolder subkey. SolverServer.log and subfolder Data are created automatically if they do not exist.
6. Auth.ini - A text format file, it contains UID/PWD pairs of administrative users in encrypted form.
7. SAdmin.exe - The server administrator utility.

## Files inside Solver SDK Dedicated Server package CGI folder

1. SolverSOAP.exe – CGI executable that can be incorporated into a Web server; used to redirect SOAP calls to Solver SDK Dedicated Server at a location on the network.
2. SolverSOAP.ini is the configuration file of SolverSOAP.exe.

---

## Environment Variable Example Code

See example code below for creating a new environment object and passing the license string. Example code is provided for COM, Visual Basic, VB.NET, C/C++/C#, Java, and Matlab.

### **COM, Visual Basic, VBA and VB.NET Example Code**

```
Dim env as New Environment 'creates new environment
'Adds new server, passing port settings and the certificate key string
env.Servers.Add(New SolverServer("localhost",2052,"insert certificate string here"))
.....
env = nothing 'destroys environment
```

### **Procedural C Example Code**

```
HEnv* env;
SolverEnvInit(& env, L""); //creates new environment
//Adds new server, passing port settings and certificate key string
SolverEnvSolverServerAdd (env, L"localhost", 2052, L"insert certificate string here");
```

### **C++ Example Code**

```
SolverPlatform.Environment env; //creates new environment
//Adds new server, passing port settings and certificate key string
env.Servers.Add(CSolverServer(L"localhost",2052,L"insert certificate string here"));
```

### **C# Example Code**

```
//creates new environment
SolverPlatform.Environment env = new SolverPlatform.environment();
//Adds new server, passing port settings and certificate key string
env.Servers.Add(new SolverServer("localhost", 2052, "insert certificate string here"));
```

### **Java Example Code**

```
//creates new Environment
SolverPlatform.Environment env = new SolverPlatform.Environment();
//Adds new server, passing port settings and certificate key string
env.Servers().Add(new SolverServer("localhost,2052,"insert certificate string here"));
```

### **Matlab Object Oriented Example Code**

```
env = Environment(); //creates new environment
//Adds new server, passing port settings and certificate key string
Add(env.Servers, SolverServer("localhost", 2052, "insert certificate string here"));
.....
Destroy(env); //destroys environment
```

### **Matlab Procedural Example Code**

```
env = SolverSDK("EnvInit"); //creates new Environment
//Adds new server, passing port settings and certificate key string
SolverSDK("SolverServerAdd", "localhost", 2052, "certificate");
env = SolverSDK("EnvFree"); //destroys Environment
```

If integrating SolverServer into a Web server such as Microsoft IIS or Apache, please replace the port setting of "localhost, 2052" with "Proxy <http://localhost:80/MySoapService/SolverSoap.exe>.

For example:

### **C# Example Code**

```
//Adds new server, passing port settings and certificate key string
env.Servers.Add(new SolverServer(Proxy
"http://localhost:80/MySOAPService/SolverSOAP.exe","insert certificate string here"));
```

---

## **More Information**

SolverServer examples can be found at C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Client\Examples in several different languages. See the section below for information on calling Solver through PHP. If you continue to have any type of question or problem with the SolverServer service or the Solver SDK Platform, please contact product support at [products@support.com](mailto:products@support.com).

---

## **Solving a Linear Model Using PHP**

Users of Solver SDK Platform can also communicate with Solver Server using SOAP in PHP. The PHP programming language contains a utility (class) which allows users to connect to a SOAP server and execute remote procedures on it. The name of this class is SoapClient. For more information on this class, see: <http://php.net/manual/en/class.soapclient.php>.

If using 64 bit SDK, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverServer\Client\Examples\php\Soap to open sixteen example files. See below for a walk through of example 1.

- a. To start, each of Frontline's PHP examples include Utils.php which is a PHP class containing a few general procedures. This file was created exclusively for Frontline's PHP SDK examples.

```
<?php
Include 'Utils.php';
```

- b. Next, we connect to SolverSDK.wsdl which is an XML file containing descriptions of all methods executed by Solver Server.

```
try {
 $a = new SoapClient("http://localhost:2052/SolverSDK.wsdl");
```



- c. Afterwards, call the remote procedure, *doSolve()*, to pass your Solver Certificate, initiate a new Problem, pass the complete model, and finally submit the model into the SolverServer queue to be solved. Once solving is completed, the final result maybe displayed by calling the WriteFinal function.
- d. This example solves a linear programming model with two variables (x1 and x2) and three constraints. Both final variable values must be 0 or larger. The example model being solved can be found below.

Maximize  $2x_1 + 3x_2$

Subject To:

$9x_1 + 6x_2 \leq 54$   
 $6x_1 + 7x_2 \leq 42$   
 $5x_1 + 10x_2 \leq 50$   
 $x_1, x_2$  non-negative

```
$s = $a->doSolve(array(
 "Desc" =>
 array(
 "Certificate" =>
 "aW9gfTlEdlxiezY5aFIzeDl5aVl0XEIwMkMwNTZiNjc7YSZjOUl0QS1OcTYvQGxMaUF
 peXIyWURtNkAhRjtRRERBUkNhSkAhd2JRR1Vlc1ZJOFxpYz9kWFNUSkBDsXl1",
 "Problem" => array(
 "Solver" => array("SolverType" => 2), // Maximize
 "Variables" => array(
 array(
 "Name" => "Vars",
 "Type" => 4, // Decision
 "Size" => 2,
 "NonNegative" => true
)
),
 "Functions" => array(
 array(
 "Name" => "Constrs",
 "Type" => 8, // Function_Type_Constraint
 "Size" => 3,
 "UpperBound" => array(54, 42, 50),
),
 array(
 "Name" => "Obj",
 "Type" => 4, // Function_Type_Objective
 "Size" => 1,
)
),
 "Model" => array(
 "AllLinear" => array(
 array(
 "FunctionType" => 8,
 "VariableType" => 4,
 "Matrix" => array(
 "ArrayOrder" => 1, // Array_Order_ByRow
 "Values" => array(9, 6, 6, 7, 5, 10)
)
)
),
 array(
 "FunctionType" => 4,
```

```

 "VariableType" => 4,
 "Matrix" => array(
 "ArrayOrder" => 1, // Array_Order_ByRow
 "Values" => array(2, 3)
)
)
)
),
),
),
),
);

$s = WaitSolve($a, $s);

WriteFinal("Example 1", $a, $s);
} catch (Exception $e) {
 echo $e->getMessage();
}
?>

```

Let's go through the call to `doSolve()` section by section. In the first section, the call to *doSolve* is initiated and the Solver Certificate is passed. For more information on obtaining a Solver Certificate, please see the Introduction to the chapter above.

```

$s = $a->doSolve(array(
 "Desc" =>
 array(
 "Certificate" =>
 "aW9gfTlEdlxiezY5aFIzeDl5aVl0XEIwMkMwNTZiNjc7YSZjOUI0QS1OcTYvQGxMaUF
 peXIyWURtNkAhRjtRRERBUkNhSkAhd2JRR1Vlc1ZJOXpYz9kWFNUSkBDsXl1",

```

Next, a new Problem is created of type Maximize. Use `SolverType = 1` for a minimization.

```

 "Problem" => array(
 "Solver" => array("SolverType" => 2), // Maximize

```

Next, we specify the variables in the model. A variable block named "Vars" is added to Problem. This variable block contains 2 variables of type 4 (`Variable_Type_Decision`). Use `Type = 2` for uncertain variables and `Type = 8` for recourse variables. To pass a lower bound of 0 on any variable that does not currently have a lower bound, use the "NonNegative" array as shown below.

```

 "Variables" => array(
 array(
 "Name" => "Vars",
 "Type" => 4, // Decision
 "Size" => 2,
 "NonNegative" => true
)
),

```

Next we add four Functions to Problem. The first three functions are constraints. Notice that we name this function block "Constraints", the size of the block is equal to 3 (for 3 constraints) and the upper bounds of the three constraints are given in the "UpperBound" argument. To pass a lower bound for a constraint, use "LowerBound". Constraints may have both upper and lower bounds.

The 2<sup>nd</sup> function block, "Obj" has a size equal to 1 and a Type equal to 4 (`Function_Type_Objective`). Currently, the SDK only supports one objective function per Problem.

```

 "Functions" => array(

```

```

array(
 "Name" => "Constraints",
 "Type" => 8, // Function_Type_Constraint
 "Size" => 3,
 "UpperBound" => array(54, 42, 50),
),
array(
 "Name" => "Obj",
 "Type" => 4, // Function_Type_Objective
 "Size" => 1,
)
),

```

The example code below calls *AllLinear* to pass the constraint and objective coefficient matrices. Note that the first call to *AllLinear* passes the coefficient matrix for `FunctionType = 8` (`Function_Type_Constraint`) and `VariableType = 4` (`Variable_Type_Decision`), or all three constraints. The second call to *AllLinear* passes the coefficient matrix for `FunctionType = 4` (`Function_Type_Objective`) and `VariableType = 4` (`Variable_Type_Decision`), or the objective function. The values are passed "ByRow" meaning that the coefficients are passed row by row. The first constraint has coefficients of 9 and 6; the coefficients for the 2<sup>nd</sup> constraint are 6 and 7; and the coefficients for the 3<sup>rd</sup> constraint are 5 and 10. It's also possible to pass coefficients by column. If we were passing coefficients "ByCol", "-1" must be passed to the `ArrayOrder` array. The `Values` array would change to:

```

"Matrix" => array(
 "ArrayOrder" => -1, // Array_Order_ByCol
 "Values" => array(9, 6, 5, 6, 7, 10)
)

```

Note: This example is passing the coefficient matrix in "dense" mode, meaning that all coefficients, even 0's, must be included in the `Values` array. It is also possible to pass the coefficient matrix in "sparse" model. See below for information on passing a sparse coefficient matrix.

Calling *AllLinear* sets the problem type to "linear" (`Problem_Type_OptLP`). Consequently, the SDK will automatically choose the best linear engine available to solve the problem. If no additional "plug in" engines have been added, then the LP/Quadratic engine will be selected in the Solver SDK Platform and the LP Simplex engine will be selected in the Solver Platform Pro.

```

"Model" => array(
 "AllLinear" => array(
 array(
 "FunctionType" => 8,
 "VariableType" => 4,
 "Matrix" => array(
 "ArrayOrder" => 1, // Array_Order_ByRow
 "Values" => array(9, 6, 6, 7, 5, 10)
)
),
 array(
 "FunctionType" => 4,
 "VariableType" => 4,
 "Matrix" => array(
 "ArrayOrder" => 1, // Array_Order_ByRow
 "Values" => array(2, 3)
)
)
)
)

```

- e. `WaitSolve()` calls the remote Solver Server SOAP procedure `getSolveStatus()` and waits till the server has finished solving. During the solution process, `WaitSolve` provides `SolverServer` with intermediate data, such as function values at a given point.

Once the Problem has been solved, the final result is printed using the `WriteFinal` function.

```

 $s = WaitSolve($a, $s);
 WriteFinal("Example 1", $a, $s);
}

```

Finally, we call our "catch" to catch and return any errors.

```

catch (Exception $e) {
 echo $e->getMessage();
}
?>

```

## Passing a Sparse Coefficient Matrix

When passing in sparse mode, you are not required to pass the 0 coefficients. For example, let's assume that our three constraints were:

$$\begin{aligned}
 9x_1 + 0x_2 &\leq 54 \\
 6x_1 + 7x_2 &\leq 42 \\
 0x_1 + 10x_2 &\leq 50
 \end{aligned}$$

Notice the 0 coefficients for  $x_2$  in the first constraint and  $x_1$  in the third constraint. By using the MajorStarts and MinorIndices arrays, you'll only need to supply the non-zero coefficient values in the Values array.

The MajorStarts array holds the index values for the nonzero coefficient that comes first in each constraint. First, you'll need to assign an index to each coefficient starting with "0" for the coefficient for  $x_1$  in the first row, 9. Moving row-wise, the next coefficient is 0. Since this coefficient is not nonzero, we will skip this one and move to the next, which is the first coefficient for the 2<sup>nd</sup> constraint, 6. We will assign an index = 1 for this coefficient. Moving row-wise again, the next nonzero coefficient is 7. We will assign an index = 2 for this coefficient. Moving to our third constraint, the next coefficient is 0, which we will skip, and move on to 10, assigning an index of 3. Our MajorStarts array will contain the indices 0 (for coefficient  $x_1$  in constraint 1), 1 (for coefficient  $x_1$  in constraint 2), and 3 (for coefficient  $x_2$  in constraint 3). Since we are passing the coefficient matrix, "ByRow", the number of entries in the MajorStarts array will be equal to the number of constraints. (If we were passing "ByCol", the number of entries in the MajorStarts array would be equal to the number of variables.)

The MinorStarts array holds the index values for the nonzero coefficients that come first in each column. Using the indices from the paragraph above, our MinorStarts array would equal: 0 and 2. These are the indices of the coefficients that appear first in each column. Since we are passing the coefficient matrix, "ByRow", the number of entries in the MinorIndices array will be equal to the number of Variables. (If we were passing "ByCol", the number of entries in the this array would equal the number of constraints.)

Finally, we pass the actual coefficient values in the Values array.

```

"Matrix" => array(
 "ArrayOrder" => 1, // Array_Order_ByRow
 "MajorStarts" => array(0, 1, 3)
 "MinorIndices" => array(0, 2)
 "Values" => array(9, 6, 7, 10)
)

```

# Appendix

---

## Setting the Problem Type

### C#

```
problem.ProblemType = Problem_Type.OptNLP;
```

### Procedural API in C

In the Procedural API, *SolverProbTypeSet* also sets the Solver Type (minimize or maximize) as well as the problem type.

```
SolverProbTypeSet(prob, Solver_Type_Minimize, Problem_Type_OptNLP);
```

### COM in VB

```
problem.ProblemType = Problem_Type_OptNLP;
```

[Back](#) to *Setting the Problem Type*

---

## Selecting the Engine

### C#

```
prob.Engine = prob.Engines[Engine.LPName];
prob.Engine = prob.Engines[Engine.GRGName];
prob.Engine = prob.Engines[Engine.LPQPName];
prob.Engine = prob.Engines[Engine.EVOName];
prob.Engine = prob.Engines[Engine.SOCPName];
```

### Procedural API in C

```
SolverProbEngineSet(prob, L"LP Simplex");
```

```
SolverProbEngineSet(prob, L"GRG Nonlinear");
SolverProbEngineSet(prob, L"LP/Quadratic");
SolverProbEngineSet(prob, L"Evolutionary");
SolverProbEngineSet(prob, L"SOCP Barrier");
```

## COM in VB

```
prob.Engine = prob.Engines.Item("LP Simplex")
prob.Engine = prob.Engines.Item("GRG Nonlinear")
prob.Engine = prob.Engines.Item("LP/Quadratic")
prob.Engine = prob.Engines.Item("Evolutionary")
prob.Engine = prob.Engines.Item("SOCP Barrier")
```

[Back](#) to Selecting an Engine.

---

## Selecting a Large Scale Engine

### C#

```
Engine engGurobi = new Engine("Gurobi", "C:\\Program Files\\Frontline
 Systems\\Engines\\gurobieng.dll");
problem.Engine = problem.Engines [problem.Engines.Add(engGurobi)];
```

See the chart below to select any of the remaining seven Large Scale engines.

|        |                                                                                                                                                                                           |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KNITRO | <pre>Engine engKnitro = new Engine("KNITRO", "C:\\Program Files\\Frontline     Systems\\Engines\\knitroeng.dll"); problem.Engine = problem.Engines[problem.Engines.Add(engGurobi)];</pre> |
| LSGRG  | <pre>Engine engLSGRG = new Engine("LSGRG", "C:\\Program Files\\Frontline     Systems\\Engines\\lsgrgeng.dll"); problem.Engine = problem.Engines[problem.Engines.Add(engLSGRG)];</pre>     |
| LSLP   | <pre>Engine engLSLP = new Engine("KNITRO", "C:\\Program Files\\Frontline     Systems\\Engines\\lslpeng.dll"); problem.Engine = problem.Engines[problem.Engines.Add(engLSLP)];</pre>       |
| LSSQP  | <pre>Engine engLSSQP = new Engine("LSSQP", "C:\\Program Files\\Frontline     Systems\\Engines\\lssqpeng.dll"); problem.Engine = problem.Engines[problem.Engines.Add(engLSSQP)];</pre>     |
| MOSEK  | <pre>Engine engMosek = new Engine("MOSEK", "C:\\Program Files\\Frontline     Systems\\Engines\\mosekeng.dll"); problem.Engine = problem.Engines[problem.Engines.Add(engMosek)];</pre>     |

|          |                                                                                                                                                                                                |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| OptQuest | <pre>Engine engOptQuest = new Engine("OptQuest", "C:\\Program Files\\Frontline Systems\\Engines\\optquesteng.dll");  problem.Engine = problem.Engines[problem.Engines.Add(engOptQuest)];</pre> |
| Xpress   | <pre>Engine engXpress = new Engine("Xpress", "C:\\Program Files\\Frontline Systems\\Engines\\xpresseng.dll");  problem.Engine = problem.Engines[problem.Engines.Add(engXpress)];</pre>         |

## Procedural API in C

```
ret = SolverEngineAdd(prob, L"Gurobi", L"C:\\Program
Files\\Frontline Systems\\Engines\\gurobieng.dll");

ret= SolverProbEngineSet(prob, L"Gurobi");
```

See the chart below to select any of the remaining seven Large Scale engines.

|          |                                                                                                                                                                       |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KNITRO   | <pre>ret = SolverEngineAdd(prob, L"KNITRO", L"C:\\Program Files\\Frontline Systems\\Engines\\knitroeng.dll");  ret= SolverProbEngineSet(prob, L"KNITRO");</pre>       |
| LSGRG    | <pre>ret = SolverEngineAdd(prob, L"LSGRG", L"C:\\Program Files\\Frontline Systems\\Engines\\LSGRGeng.dll");  ret= SolverProbEngineSet(prob, L"LSGRG");</pre>          |
| LSLP     | <pre>ret = SolverEngineAdd(prob, L"LSLP", L"C:\\Program Files\\Frontline Systems\\Engines\\lslpeng.dll");  ret= SolverProbEngineSet(prob, L"LSLP");</pre>             |
| LSSQP    | <pre>ret = SolverEngineAdd(prob, L"LSSQP", L"C:\\Program Files\\Frontline Systems\\Engines\\lssqpeng.dll");  ret= SolverProbEngineSet(prob, L"LSSQP");</pre>          |
| MOSEK    | <pre>ret = SolverEngineAdd(prob, L"MOSEK", L"C:\\Program Files\\Frontline Systems\\Engines\\mosekeng.dll");  ret= SolverProbEngineSet(prob, L"MOSEK");</pre>          |
| OptQuest | <pre>ret = SolverEngineAdd(prob, L"OptQuest", L"C:\\Program Files\\Frontline Systems\\Engines\\optquesteng.dll");  ret= SolverProbEngineSet(prob, L"OptQuest");</pre> |

|        |                                                                                                                                                                |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Xpress | <pre>ret = SolverEngineAdd(prob, L"Xpress", L"C:\\Program Files\\Frontline Systems\\Engines\\xpresseng.dll"); ret= SolverProbEngineSet(prob, L"Xpress");</pre> |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|

## COM in VB

Since Visual Basic does not support constructors, we use the .Init property to initialize *engGurobi* to the Gurobi engine located at the path C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Bin\gurobieng.dll. The third and fourth lines of code add the Gurobi engine to the collection and then select the engine to solve the problem, respectively.

```
Dim engGurobi As New Engine
engGurobi.Init "Gurobi", "C:\Program Files\Frontline
Systems\Engines\gurobieng.dll"
prob.Engines.Add engGurobi
prob.Engine = prob.Engines("Gurobi")
```

See the chart below to select any of the remaining seven Large Scale engines.

|        |                                                                                                                                                                                           |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KNITRO | <pre>Dim engKnitro As New Engine engKnitroInit "KNITRO", "C:\Program Files\Frontline Systems\Engines\knitroeng.dll" prob.Engines.Add engKnitro prob.Engine = prob.Engines("KNITRO")</pre> |
| LSGRG  | <pre>Dim engLSGRG As New Engine engLSGRG.Init "LSGRG", "C:\Program Files\Frontline Systems\Engines\lsgrgeng.dll" prob.Engines.Add engLSGRG prob.Engine = prob.Engines("LSGRG")</pre>      |
| LSLP   | <pre>Dim engLSLP As New Engine engLSLP.Init "LSLP", "C:\Program Files\Frontline Systems\Engines\lslpeng.dll" prob.Engines.Add engLSLP prob.Engine = prob.Engines("LSLP")</pre>            |
| LSSQP  | <pre>Dim engLSSQP As New Engine engLSSQP.Init "LSSQP", "C:\Program Files\Frontline Systems\Engines\lssqpeng.dll" prob.Engines.Add engLSSQP prob.Engine = prob.Engines("LSSQP")</pre>      |
| MOSEK  | <pre>Dim engMOSEK As New Engine engMOSEK.Init "MOSEK", "C:\Program Files\Frontline Systems\Engines\mosekeng.dll" prob.Engines.Add engMOSEK</pre>                                          |



|          |                                                                                                                                                                                                                                                        |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|          | <code>prob.Engine = prob.Engines("MOSEK")</code>                                                                                                                                                                                                       |
| OptQuest | <code>Dim engOptQuest As New Engine</code><br><code>engOptQuest.Init "OptQuest", "C:\Program Files\Frontline Systems\Engines\optquesteng.dll"</code><br><code>prob.Engines.Add engGurobi</code><br><code>prob.Engine = prob.Engines("OptQuest")</code> |
| Xpress   | <code>Dim engXpress As New Engine</code><br><code>engXpress.Init "Xpress", "C:\Program Files\Frontline Systems\Engines\Xpresseng.dll"</code><br><code>prob.Engines.Add engXpress</code><br><code>prob.Engine = prob.Engines("Xpress")</code>           |

[Back](#) to Selecting a Large Scale Engine.

---

## Using Discovery Method

### C#

```
ret = prob.Engines.Discover("c:\\program files\\frontline
systems\\Engines");

//Replace "Gurobi Solver" in the line of code below with the desired
engine name: "KNITRO Solver", "Large Scale GRG Solver", "Large
Scale LP Solver", "Large Scale SQP Solver", "Mosek Solver",
"OptQuest Solver", or "Xpress Solver"

prob.Engine = prob.Engines("Gurobi Solver");
```

### Procedural API in C

```
ret = SolverEngineDiscover(prob, L"C:\\Program Files\\Frontline
Systems\\Engines");

//Replace "Gurobi Solver" in the line of code below with the desired
engine name: "KNITRO Solver", "Large Scale GRG Solver", "Large
Scale LP Solver", "Large Scale SQP Solver", "Mosek Solver",
"OptQuest Solver", or "Xpress Solver"

SolverProbEngineSet(prob, L"Gurobi Solver");
```

### COM in VB

```
ret = prob.Engines.Discover("C:\Program Files\Frontline
Systems\Engines")

'Replace "Gurobi Solver" in the line of code below with the desired
engine name: "KNITRO Solver", "Large Scale GRG Solver", "Large
Scale LP Solver", "Large Scale SQP Solver", "Mosek Solver",
"OptQuest Solver", or "Xpress Solver"

prob.Engine = prob.Engines("Gurobi Solver")
```

[Back](#) to Using Discovery Method.

---

# Using an Evaluator to Check the Progress of an Optimization

## C#

```
problem.Evaluators[Eval_Type.Iteration].OnEvaluate += new
 EvaluateEventHandler(EvalIteration_Evaluate);

private Engine_Action EvalIteration_OnEvaluate(Evaluator evaluator)
{
 MessageBox.Show("Callback Evaluator Called. Iteration = " +
 evaluator.Problem.Engine.Stat.Iterations + ", Objective = " +
 evaluator.Problem.FcnObjective.Value[0]);

 MessageBox.Show("varvalue " +
 evaluator.Problem.VarDecision.Value[0]);

 return Engine_Action.Continue;
}
```

[Back](#) to *Using an Evaluator to Check the Progress of an Optimization*

## Procedural API in C

```
SolverEvaluatorAdd(prob, EvalIteration, NULL, Eval_Type_Iteration);

_Engine_Action EvalIteration(HProblem prob, void* refuser,
 _Eval_Type evaltype)
{
 double obj[1];
 double x[2];

 ret = SolverEngStat(prob, L"LP/Quadratic", NULL, &piterations,
 NULL, NULL, NULL, NULL, NULL);
 printf("Iterations = %g\n", piterations);
 SolverFcnValueGet(prob, L"Objective", obj);
 SolverVarValue(prob, L"Vars", x);
 cout << "CallBack: Obj[0] = " << obj[0] << endl;
 cout << "X[0] = " << x[0] << " << x[0] << endl;
 cout << "X[1] = " << x[1] << " << x[1]<< endl;
 return Engine_Action_Continue;
}
```

*SolverEngStat()* "gets" several types of engine statistics.

```
SolverEngStat (HProblem prob, LPCWSTR enginename, /*[out]*/
 int*pmilliseconds, /*[out]*/ int* piterations, /*[out]*/ int*
 psubproblems, /*[out]*/ int* plocalsols, /*[out]*/ int*
```

```

 pfuncevals, /*[out]*/ int* pjacevals, /*[out]*/ int*
 phessevals);

```

pmilliseconds    Time in milliseconds.

piterations       Number of iterations in an optimization problem and number of trials in a simulation problem.

psubproblems     Number of subproblems in an optimization problem.

plocalsols        Number of local solutions in a nonlinear optimization problem.

pfuncevals        Number of calls to the evaluator of type Eval\_Type\_Function.

pjacevals         Number of calls to the evaluator of type Eval\_Type\_Gradient.

phessevals        Number of calls to the evaluator of type Eval\_Type\_Hessian.

[Back](#) to *Using an Evaluator to Check the Progress of an Optimization*

## COM in VB

```

Set EvalIteration = New SolverPlatform.evaluator
prob.Evaluators.Item(Eval_Type_Iteration) = EvalIteration

Private Function EvalIteration_Evaluate(ByVal evaluator As
SolverPlatform.IEvaluator) As SolverPlatform.Engine_Action
 MsgBox ("Callback Evaluator Called. Iteration = " & _
 evaluator.Problem.Engine.Stat.Iterations)
 MsgBox ("Objective = " &
evaluator.Problem.FcnObjective.Value(0))
 EvalIteration_Evaluate = Engine_Action_Continue
End Function

```

[Back](#) to *Using an Evaluator to Check the Progress of an Optimization*

---

# Using a Function Evaluator to Compute the Objective and Constraints

## C#

```

problem.Evaluators[Eval_Type.Function].OnEvaluate += new
EvaluateEventHandler(evalNLP_Evaluate);

Engine_Action evalNLP_OnEvaluate(Evaluator evaluator)
{
 Problem p = evaluator.Problem;
 double[] pVar = p.VarDecision.Value.Array;
 double[] pFcn = p.FcnConstraint.Value.Array;
 pFcn[0] = pVar[0] + pVar[1];
}

```

```

 pFcn[1] = pVar[0] * pVar[1];
 p.FcnObjective.Value[0] = pVar[0] * pVar[0] + pVar[1] * pVar[1];
 p.FcnConstraint.Value.Array = pFcn;
 return Engine_Action.Continue;
}

```

[Back](#) to *Using an Function Evaluator to Compute the Objective and Constraints*

## Procedural API in C

```

SolverEvaluatorAdd(prob, evalNLP, NULL, Eval_Type_Function);

_Engine_Action evalNLP(HProblem prob, void* refuser, _Eval_Type
 evaltype)
{
 double var[2];
 double fcn[2];
 double obj[1];
 SolverVarValue(prob, L"Vars", var);
 fcn[0] = var[0] + var[1];
 fcn[1] = var[0] * var[1];
 obj[0] = var[0]*var[0] + var[1]*var[1];
 SolverFcnValueSet(prob, L"Constraints", fcn);
 SolverFcnValueSet(prob, L"Objective", obj);
 return Engine_Action_Continue;
}

```

[Back](#) to *Using an Function Evaluator to Compute the Objective and Constraints*

## COM in VB

```

Set evalNLP = New SolverPlatform.evaluator
prob.Evaluators.Item(Eval_Type_Function) = evalNLP

Private Function evalNLP_Evaluate(ByVal Evaluator As
SolverPlatform.IEvaluator) As Engine_Action
 Dim p As Problem
 p = Evaluator.Problem
 Dim pVar() As Double
 pVar = p.VarDecision.Value.Array
 Dim Fcn(1) As Double
 pFcn(0) = Var(0) + Var(1)
 pFcn(1) = Var(0) * Var(1)

```

```

Dim obj As Double
p.FcnObjective.Value(0) = pVar(0) * pVar(0) + pVar(1) * pVar(1)
p.FcnConstraint.Value.Array = pFcn
NLPEvaluator_Evaluate = Engine_Action_Continue
End Function

```

[Back](#) to *Using an Function Evaluator to Compute the Objective and Constraints*

---

## Using a Gradient Evaluator to Provide First Order Partial Derivatives

### C#

```

problem.Evaluators[Eval_Type.Gradient].OnEvaluate += new
EvaluateEventHandler(evalJac_Evaluate);

Engine_Action evalJac_OnEvaluate(Evaluator evaluator)
{
 DoubleMatrix objGrad =
 evaluator.Problem.Model.AllGradValue[Function_Type.Objective];
 objGrad[0, 0] = 2.0 * evaluator.Problem.VarDecision.Value[0];
 objGrad[0, 1] = 2.0 * evaluator.Problem.VarDecision.Value[1];
 DoubleMatrix matGrad =
 evaluator.Problem.Model.AllGradValue[Function_Type.Constraint]
;
 matGrad[0, 0] = 1.0;
 matGrad[0, 1] = 1.0;
 matGrad[1, 0] = evaluator.Problem.VarDecision.Value[1];
 matGrad[1, 1] = evaluator.Problem.VarDecision.Value[0];
 return Engine_Action.Continue;
}

```

[Back](#) to *Using a Gradient Evaluator to Provide First Order Partial Derivatives*.

### Procedural API in C

```

SolverEvaluatorAdd(prob, evalJac, NULL, Eval_Type_Gradient);

_Engine_Action evalJac(HProblem prob, void* refuser, _Eval_Type
 evaltype)
{
 double Var[2];
 double objgrad[2];

```

```

double matval[4];
SolverVarValue(prob, L"Vars", Var);
objgrad[0] = 2.0 * Var[0];
objgrad[1] = 2.0 * Var[1];
//notice that the matval array is passed "column-wise"
//matval[0] and matval[2] are the partial derivatives for the
//first constraint, x + y
//matval[1] and matval[3] are the partial derivatives for the
//second constraint, x * y
matval[0] = 1.0;
matval[1] = Var[1];
matval[2] = 1.0;
matval[3] = Var[0];
SolverModAllGradientSet(prob, Function_Type_Objective,
 Variable_Type_Decision, objgrad);
SolverModAllGradientSet(prob, Function_Type_Constraint,
 Variable_Type_Decision, matval);
return Engine_Action_Continue;
}

```

[Back](#) to *Using a Gradient Evaluator to Provide First Order Partial Derivatives.*

## COM in VB

```

Set evalJac = New SolverPlatform.evaluator
prob.Evaluators.Item(Eval_Type_Gradient) = evalJac

Private Function evalJac_Evaluate(ByVal Evaluator As
 SolverPlatform.IEvaluator) As Engine_Action
Dim Var() As Double
Var = evaluator.Problem.VarDecision.Value.Array
With evaluator.Problem.Model.AllGradValue(Function_Type_Objective,
 Variable_Type_Decision)
 .Value(0, 0) = 2# * Var(0)
 .Value(0, 1) = 2# * Var(1)
End With
With evaluator.Problem.Model.AllGradValue(Function_Type_Constraint,
 Variable_Type_Decision)
 .Value(0, 0) = 1#
 .Value(0, 1) = 1#
 .Value(1, 0) = Var(1)
 .Value(1, 1) = Var(0)
End With

```

End With

evalJac\_Evaluate = Engine\_Action\_Continue

End Function

Back to *Using a Gradient Evaluator to Provide First Order Partial Derivatives*.

---

## Using a Hessian Evaluator to Provide Second Order Partial Derivatives

### Solving a Quadratic Problem with a Hessian Evaluator

#### C#

```
int nvars = 5, ncons = 2;
double[] objcoeff = { 0.0, 0.0, 0.0, 0.0, 0.0 };
double[] ub_constraints = { 1.0, Constants.PINF };
double[] lb_constraints = { 1.0, 0.085 };
double[] concoeff = { 1.0, 1.0, 1.0, 1.0, 1.0, 0.086, 0.071, 0.095,
 0.107, 0.069 };
double[] lb = { 0.0, 0.0, 0.0, 0.0, 0.0 };
double[] ub = { 1.0, 1.0, 1.0, 1.0, 1.0 };
try
{
 using (Problem problem = new Problem(Solver_Type.Minimize, nvars,
 ncons))
 {
 problem.FcnConstraint.UpperBound.Array = ub_constraints;
 problem.FcnConstraint.LowerBound.Array = lb_constraints;
 problem.VarDecision.UpperBound.Array = ub;
 problem.VarDecision.LowerBound.Array = lb;
 problem.Model.AllLinear[Function_Type.Objective] = new
 DoubleMatrix(Array_Order.ByCol, 1, nvars, objcoeff);
 problem.Model.AllLinear[Function_Type.Constraint] = new
 DoubleMatrix(Array_Order.ByRow, ncons, nvars, concoeff);
 problem.ProblemType = Problem_Type.OptQP;
 problem.Evaluators[Eval_Type.Hessian].OnEvaluate += new
 EvaluateEventHandler(Hessevaluator_Evaluate);
 problem.Solver.Optimize();
 }
}
//See the chapter, Calling the Solver SDK through C# for example
code //obtaining the optimization status and results.
```

```

}

Engine_Action evalHess_OnEvaluate(Evaluator evaluator)
{
 Problem prob = evaluator.Problem;
 double[] qmatval =
 {
 0.000204, 0.000424, 0.000170, 0.000448, -0.000014,
 0.000424, 0.012329, 0.001785, 0.001633, -0.000539,
 0.000170, 0.001785, 0.000365, 0.000425, -0.000075,
 0.000448, 0.001633, 0.000425, 0.005141, 0.000237,
 -0.000014, -0.000539, -0.000075, 0.000237, 0.000509
 };
 if (prob.Solver.FunctionType == Function_Type.Objective)
 prob.Model.FcnHessValue[Function_Type.Objective,0] =
 new DoubleMatrix(Array_Order.ByCol, 5, 5, qmatval); ;
 return Engine_Action.Continue;
}

```

[Back](#) to *Solving a Quadratic Problem with a Hessian Evaluator*.

## Procedural API in C

```

_Optimize_Status status;
double x[5];
HProblem prob;
int nvars = 5, ncons = 2;
double objcoeff[] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
double ub_constraints[] = { 1.0, 1E+30 };
double lb_constraints[] = { 1.0, 0.085 };
double concoeff[] = { 1.0, 1.0, 1.0, 1.0, 1.0, 0.086, 0.071, 0.095,
0.107, 0.069 };
double ub[] = { 1.0, 1.0, 1.0, 1.0, 1.0 };
double lb[] = { 0.0, 0.0, 1.0, 1.0, 1.0 };
SolverProblem(&prob);
SolverProbTypeSet(prob, Solver_Type_Minimize, Problem_Type_OptQP);
SolverVariableAdd(prob, L"Vars", Variable_Type_Decision, nvars);
SolverVarUpperBoundSet(prob, L"Vars", ub);
SolverVarLowerBoundSet(prob, L"Vars", lb);
SolverFunctionAdd(prob, L"Objective", Function_Type_Objective, 1);

```



```

SolverFunctionAdd(prob, L"Constraints", Function_Type_Constraint,
 ncons);
SolverFcnLowerBoundSet(prob, L"Constraints", lb_constraints);
SolverFcnUpperBoundSet(prob, L"Constraints", ub_constraints);
SolverModArrayOrder(prob, Array_Order_ByRow);
SolverModAllLinearSet(prob, Function_Type_Constraint,
 Variable_Type_Decision, concoeff);
SolverEvaluatorAdd(prob, evalHess, NULL, Eval_Type_Hessian);
SolverOptimize(prob, &status, &objval);

//See the chapter, Calling the Solver SDK through the //Procedural
API to see example code obtaining the optimization //status and
results.
}

Engine_Action evalHess(HProblem prob, void* refuser, _Eval_Type
 evaltype)
{
 _Function_Type fcnType;
 double qmatval[] =
 { 0.000204, 0.000424, 0.000170, 0.000448, -0.000014,
 0.000424, 0.012329, 0.001785, 0.001633, -0.000539,
 0.000170, 0.001785, 0.000365, 0.000425, -0.000075,
 0.000448, 0.001633, 0.000425, 0.005141, 0.000237,
 -0.000014, -0.000539, -0.000075, 0.000237, 0.000509 };
 SolverFunctionType(prob, &fcnType);
 if (fcnType == Function_Type_Objective)
 SolverModFcnHessianSet(prob, fcnType, Variable_Type_Decision,
 0, qmatval);
 return Engine_Action_Continue;
}

```

[Back](#) to *Solving a Quadratic Problem with a Hessian Evaluator.*

## COM in VB

```

Dim objcoeff As SolverPlatform.DoubleMatrix
Set objcoeff = New SolverPlatform.DoubleMatrix
concoeff.Create Rows:=1, Columns:=5
objcoeff.Value(0, 0) = 0
objcoeff.Value(0, 1) = 0
objcoeff.Value(0, 2) = 0
objcoeff.Value(0, 3) = 0

```

```

objcoeff.Value(0, 4) = 0
Dim concoeff As SolverPlatform.DoubleMatrix
Set concoeff = New SolverPlatform.DoubleMatrix
concoeff.Create Rows:=2, Columns:=5
concoeff.Value(0, 0) = 1#
concoeff.Value(0, 1) = 1#
concoeff.Value(0, 2) = 1#
concoeff.Value(0, 3) = 1#
concoeff.Value(0, 4) = 1#
concoeff.Value(1, 0) = 0.086
concoeff.Value(1, 1) = 0.071
concoeff.Value(1, 2) = 0.095
concoeff.Value(1, 3) = 0.107
concoeff.Value(1, 4) = 0.069
Dim ub_constraints(1) As Double
ub_constraints(0) = 1#
ub_constraints(1) = 1E+30
Dim lb_constraints(1) As Double
lb_constraints(0) = 1#
lb_constraints(1) = 0.085
Dim ub(4) As Double
ub(0) = 1#
ub(1) = 1#
ub(2) = 1#
ub(3) = 1#
ub(4) = 1#
Dim lb(4) As Double
lb(0) = 0
lb(1) = 0
lb(2) = 0
lb(3) = 0
lb(4) = 0
Dim prob As SolverPlatform.Problem
Set prob = New SolverPlatform.Problem
prob.FcnConstraint.Size = 2
prob.VarDecision.Size = 5
prob.FcnConstraint.UpperBound.Array = ub_constraints
prob.FcnConstraint.LowerBound.Array = lb_constraints

```

```

prob.VarDecision.UpperBound.Array = ub
prob.VarDecision.LowerBound.Array = lb
prob.Model.AllLinear(Function_Type_Objective,
 Variable_Type_Decision) = objcoeff
prob.Model.AllLinear(Function_Type_Constraint,
 Variable_Type_Decision) = concoeff
Set HessEvaluator = New SolverPlatform.evaluator
prob.Evaluators.Item(Eval_Type_Hessian) = evalHess
prob.ProblemType = Problem_Type_OptQP
prob.Solver.SolverType = Solver_Type_Minimize
prob.Solver.Optimize
'See the chapter, Calling the Solver SDK through COM to see example
'code obtaining the optimization status and results.
End Sub

```

```

Private Function evalHess_Evaluate(ByVal evaluator As
SolverPlatform.IEvaluator) As Engine_Action
Dim QMatVal As SolverPlatform.DoubleMatrix
Set QMatVal = New SolverPlatform.DoubleMatrix
QMatVal.Create Rows:=5, Columns:=5
QMatVal.Value(0, 0) = 0.000204
QMatVal.Value(1, 0) = 0.000424
QMatVal.Value(2, 0) = 0.00017
QMatVal.Value(3, 0) = 0.000448
QMatVal.Value(4, 0) = -0.000014
QMatVal.Value(0, 1) = 0.000424
QMatVal.Value(1, 1) = 0.012329
QMatVal.Value(2, 1) = 0.001785
QMatVal.Value(3, 1) = 0.001633
QMatVal.Value(4, 1) = -0.000539
QMatVal.Value(0, 2) = 0.00017
QMatVal.Value(1, 2) = 0.001785
QMatVal.Value(2, 2) = 0.000365
QMatVal.Value(3, 2) = 0.000425
QMatVal.Value(4, 2) = -0.000075
QMatVal.Value(0, 3) = 0.000448
QMatVal.Value(1, 3) = 0.001633
QMatVal.Value(2, 3) = 0.000425
QMatVal.Value(3, 3) = 0.005141

```

```

QMatVal.Value(4, 3) = 0.000237
QMatVal.Value(0, 4) = -0.000014
QMatVal.Value(1, 4) = -0.000539
QMatVal.Value(2, 4) = -0.000075
QMatVal.Value(3, 4) = 0.000237
QMatVal.Value(4, 4) = 0.000509
evaluator.Problem.Model.FcnHessValue(Function_Type_Objective, 0) =
QMatVal
evalHess_Evaluate = Engine_Action_Continue
End Function

```

[Back](#) to Solving a Quadratic Problem with a Hessian Evaluator.

## Solving a Quadratically Constrained Problem with a Hessian Evaluator

### C#

```

int nvars = 5, ncons = 2;
double[] objcoeff = { 0.086, 0.0714, 0.095, 0.107, 0.069 };
double[] ub_constraints = { 1.0, 0.01 };
double[] lb_constraints = { 1.0, Constants.MINF };
double[] concoeff = { 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0,
0.0 };
double[] lb = { 0.0, 0.0, 0.0, 0.0, 0.0 };
double[] ub = { 1.0, 1.0, 1.0, 1.0, 1.0 };
Depend_Type[] concoefftype =
{
 Depend_Type.Linear, Depend_Type.Linear,
 Depend_Type.Linear, Depend_Type.Linear,
 Depend_Type.Linear, Depend_Type.Quadratic,
 Depend_Type.Quadratic, Depend_Type.Quadratic,
 Depend_Type.Quadratic, Depend_Type.Quadratic
};
try
{
 using (Problem problem = new Problem(Solver_Type.Minimize, nvars,
ncons))
 {
 problem.FcnConstraint.UpperBound.Array = ub_constraints;
 problem.FcnConstraint.LowerBound.Array = lb_constraints;
 }
}

```

```

 problem.VarDecision.UpperBound.Array = ub;
 problem.VarDecision.LowerBound.Array = lb;
 problem.Model.AllLinear[Function_Type.Objective] = new
 DoubleMatrix(Array_Order.ByCol, 1, nvars, objcoeff);
 problem.Model.AllLinear[Function_Type.Constraint] = new
 DoubleMatrix(Array_Order.ByRow, ncons, nvars, concoeff);
 problem.Model.AllGradDepend[Function_Type.Constraint] = new
 DependMatrix(Array_Order.ByRow, ncons, nvars, concoefftype);
 problem.ProblemType = Problem_Type.OptQCP;
 problem.Evaluators[Eval_Type.Hessian].OnEvaluate += new
 EvaluateEventHandler(evalHess_Evaluate);
 problem.Solver.Optimize();

//See the chapter, Calling the Solver SDK through C# for example
code //obtaining the optimization status and results.
}

```

```

Engine_Action evalHess_OnEvaluate(Evaluator evaluator)
{
 Problem prob = evaluator.Problem;
 double[] qmatval =
 {
 0.000204, 0.000424, 0.000170, 0.000448, -0.000014,
 0.000424, 0.012329, 0.001785, 0.001633, -0.000539,
 0.000170, 0.001785, 0.000365, 0.000425, -0.000075,
 0.000448, 0.001633, 0.000425, 0.005141, 0.000237,
 -0.000014, -0.000539, -0.000075, 0.000237, 0.000509
 };
 if (prob.Solver.FunctionType == Function_Type.Constraint)
 prob.Model.FcnHessValue[Function_Type.Constraint,0] =
 new DoubleMatrix(Array_Order.ByCol, 5, 5, qmatval); ;
 return Engine_Action.Continue;
}

```

Back to *Solving a Quadratically Constrained Problem with a Hessian Evaluator.*

## Procedural API in C

```

_Optimize_Status status;
double x[5];
HProblem prob;
int nvars = 5, ncons = 2;
double objcoeff[] = { 0.086, 0.0714, 0.095, 0.107, 0.069 };

```

```
double ub_constraints[] = { 1.0, 0.01 };
double lb_constraints[] = {1.0, -1E+30};
double concoeff[] = { 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0,
 0.0 };
double ub[] = { 1.0, 1.0, 1.0, 1.0, 1.0 };
double lb[] = { 0.0, 0.0, 1.0, 1.0, 1.0 };
```

In the Procedural API, we must pass two additional arrays in order to call AllGradDependSet to pass the diagnostic information for the constraints (i.e. whether the variables appear linearly or quadratically), the matbeg array and the matind array. We are passing the concoefftype array inside AllGradDependSet method "row-wise". The matbeg array contains the indices for the elements in the concoefftype array that begin each column. For example the first element in the matbeg array (0) is the index of the first element in the concoefftype array that begins the x1 column. The second element in the matbeg array (1) is the index of the second element in the concoefftype array that begins the x2 column. The size of the matbeg array will always be nvars (number of variables). The matind array holds the the row number of the corresponding coefficient in the concoefftype array. For example, the first five elements in the matind array belong to the first constraint ( $x_1 + x_2 + x_3 + x_4 + x_5$ ) and the last five elements belong to the 2<sup>nd</sup> constraint (portfolio return).

```
int matbeg[] = { 0, 1, 2, 3, 4 };
int matind[] = { 0, 0, 0, 0, 0, 1, 1, 1, 1, 1 };
_Depend_Type concoefftype[] =
{
 Depend_Type_Linear, Depend_Type_Linear, Depend_Type_Linear,
 Depend_Type_Linear, Depend_Type_Linear, Depend_Type_Quadratic,
 Depend_Type_Quadratic, Depend_Type_Quadratic,
 Depend_Type_Quadratic, Depend_Type_Quadratic
};

SolverProblem(&prob);

SolverProbTypeSet(prob, Solver_Type_Minimize, Problem_Type_OptQCP);
SolverVariableAdd(prob, L"Vars", Variable_Type_Decision, nvars);
SolverFcnLowerBoundSet(prob, L"Constraints", lb_constraints);
SolverFcnUpperBoundSet(prob, L"Constraints", ub_constraints);
SolverVarUpperBoundSet(prob, L"Vars", ub);
SolverVarLowerBoundSet(prob, L"Vars", lb);
SolverFunctionAdd(prob, L"Objective", Function_Type_Objective, 1);
SolverFunctionAdd(prob, L"Constraints", Function_Type_Constraint,
 ncons);
SolverModArrayOrder(prob, Array_Order_ByRow);
SolverModAllLinearSet(prob, Function_Type_Objective,
 Variable_Type_Decision, objcoeff);
```



[Back](#) to *Solving a Quadratically Constrained Problem with a Hessian Evaluator*.

## COM in VB

```
Dim objcoeff As SolverPlatform.DoubleMatrix
Set objcoeff = New SolverPlatform.DoubleMatrix
objcoeff.Create Rows:=1, Columns:=5
objcoeff.Value(0, 0) = 0.086
objcoeff.Value(0, 1) = 0.0714
objcoeff.Value(0, 2) = 0.095
objcoeff.Value(0, 3) = 0.107
objcoeff.Value(0, 4) = 0.069
Dim concoeff As SolverPlatform.DoubleMatrix
Set concoeff = New SolverPlatform.DoubleMatrix
concoeff.Create Rows:=2, Columns:=5
concoeff.Value(0, 0) = 1#
concoeff.Value(0, 1) = 1#
concoeff.Value(0, 2) = 1#
concoeff.Value(0, 3) = 1#
concoeff.Value(0, 4) = 1#
concoeff.Value(1, 0) = 0.0
concoeff.Value(1, 1) = 0.0
concoeff.Value(1, 2) = 0.0
concoeff.Value(1, 3) = 0.0
concoeff.Value(1, 4) = 0.0
Dim ub_constraints(1) As Double
ub_constraints(0) = 1#
ub_constraints(1) = 0.01
Dim lb_constraints(1) As Double
lb_constraints(0) = 1#
lb_constraints(1) = -1E+30
Dim ub(4) As Double
ub(0) = 1#
ub(1) = 1#
ub(2) = 1#
ub(3) = 1#
ub(4) = 1#
Dim lb(4) As Double
lb(0) = 0
```



```

lb(1) = 0
lb(2) = 0
lb(3) = 0
lb(4) = 0
Dim prob As SolverPlatform.Problem
Set prob = New SolverPlatform.Problem
prob.FcnConstraint.Size = 2
prob.VarDecision.Size = 5
prob.FcnConstraint.UpperBound.Array = ub_constraints
prob.FcnConstraint.LowerBound.Array = lb_constraints
prob.VarDecision.UpperBound.Array = ub
prob.VarDecision.LowerBound.Array = lb
prob.Model.AllLinear(Function_Type_Objective,
 Variable_Type_Decision) = objcoeff
prob.Model.AllLinear(Function_Type_Constraint,
 Variable_Type_Decision) = concoeff
Set HessEvaluator = New SolverPlatform.evaluator
prob.Evaluators.Item(Eval_Type_Hessian) = evalHess
prob.ProblemType = Problem_Type_OptQCP
prob.Solver.SolverType = Solver_Type_Minimize
prob.Solver.Optimize
'See the chapter, Calling the Solver SDK through COM to see example
'code obtaining the optimization status and results.
End Sub

Private Function evalHess_Evaluate(ByVal evaluator As
SolverPlatform.IEvaluator) As Engine_Action
Dim i as Integer
'create an empty array to hold the quadratic terms of the first
'constraint, which are none
Dim QMatVal0 As SolverPlatform.DoubleMatrix
Set QMatVal0 = New SolverPlatform.DoubleMatrix
QMatVal0.Create Rows:=5, Columns:=5
'create a 2nd QmatVal array to hold the qmatrix for the 2nd
'constraint.
Dim QMatVal1 As SolverPlatform.DoubleMatrix
Set QMatVal1 = New SolverPlatform.DoubleMatrix
QMatVal1.Create Rows:=5, Columns:=5
QMatVal1.Value(0, 0) = 0.000204
QMatVal1.Value(1, 0) = 0.000424

```

```

QMatVal1.Value(2, 0) = 0.00017
QMatVal1.Value(3, 0) = 0.000448
QMatVal1.Value(4, 0) = -0.000014
QMatVal1.Value(0, 1) = 0.000424
QMatVal1.Value(1, 1) = 0.012329
QMatVal1.Value(2, 1) = 0.001785
QMatVal1.Value(3, 1) = 0.001633
QMatVal1.Value(4, 1) = -0.000539
QMatVal1.Value(0, 2) = 0.00017
QMatVal1.Value(1, 2) = 0.001785
QMatVal1.Value(2, 2) = 0.000365
QMatVal1.Value(3, 2) = 0.000425
QMatVal1.Value(4, 2) = -0.000075
QMatVal1.Value(0, 3) = 0.000448
QMatVal1.Value(1, 3) = 0.001633
QMatVal1.Value(2, 3) = 0.000425
QMatVal1.Value(3, 3) = 0.005141
QMatVal1.Value(4, 3) = 0.000237
QMatVal1.Value(0, 4) = -0.000014
QMatVal1.Value(1, 4) = -0.000539
QMatVal1.Value(2, 4) = -0.000075
QMatVal1.Value(3, 4) = 0.000237
QMatVal1.Value(4, 4) = 0.000509

'pass the hessian matrix for the first constraint (x1+x2+x3+x4+x5)
evaluator.Problem.Model.FcnHessValue(Function_Type_Constraint, 0) =
QMatVal0

'pass the hessian matrix for the second constraint (Portfolio
Return)
evaluator.Problem.Model.FcnHessValue(Function_Type_Constraint, 1) =
QMatVal1

evalHess_Evaluate = Engine_Action_Continue

End Function

```

[Back](#) to Solving a *Quadratically Constrained Problem with a Hessian Evaluator*.

## Solving a Nonlinear Problem with a Hessian Evaluator

### C#

```

problem.Evaluators[Eval_Type.Hessian].OnEvaluate += new
EvaluateEventHandler(evalHess_Evaluate);

```

```

Engine_Action evalHess_OnEvaluate(Evaluator evaluator)
{
 Problem prob = evaluator.Problem;
 double [] objHess = { 2, 0, 0, 2}; //elements passed ByRow
 double [] matHess = { 0, 1, 1, 0}; //elements passed ByRow
 if (prob.Solver.FunctionType == Function_Type.Objective)
 prob.Model.FcnHessValue[Function_Type.Objective, 0] = new
 DoubleMatrix(Array_Order.ByRow, 2, 2, objHess);
 if (prob.Solver.FunctionType == Function_Type.Constraint)
 {
 prob.Model.FcnHessValue[Function_Type.Constraint, 0] = new
 DoubleMatrix (2, 2, (double)0);
 prob.Model.FcnHessValue[Function_Type.Constraint, 1] = new
 DoubleMatrix (Array_Order.ByRow, 2, 2, matHess);
 }
 return Engine_Action.Continue;
}

```

Back to *Solving a Nonlinear Problem with a Hessian Evaluator*

## Procedural API in C

```

SolverEvaluatorAdd(prob, evalHess, NULL, Eval_Type_Hessian);

_Engine_Action evalHess(HProblem prob, void* refuser, _Eval_Type
 evaltype)
{
 _Function_Type fcnType;
 double objHess[] = { 2, 0, 0, 2 };
 double matHess0[] = { 0, 0, 0, 0};
 double matHess1[] = {0, 1, 1, 0};
 SolverFunctionType(prob, &fcnType);
 if (fcnType == Function_Type_Objective)
 SolverModFcnHessianSet (prob, fcnType, Variable_Type_Decision, 0,
 objHess);
 If (fcnType == Function_Type_Constraint)
 {
 //passes matHess0 for the first constraint that has 0 quadratic
 terms
 SolverModFcnHessianSet (prob, fcnType, Variable_Type_Decision, 0,
 matHess0);
 //passes matHess1 for the second constraint
 }
}

```

```

SolverModFcnHessianSet(prob, fcnType, Variable_Type_Decision, 1,
 matHess1);
}
return Engine_Action_Continue;
}

```

[Back](#) to Solving a Nonlinear Problem with a Hessian Evaluator

## COM in VB

```

Set evalJac = New SolverPlatform.evaluator
prob.Evaluators.Item(Eval_Type_Hessian) = evalJac

Private Function evalJac_Evaluate(ByVal Evaluator As
 SolverPlatform.IEvaluator) As Engine_Action
Dim objHess As SolverPlatform.DoubleMatrix
Set objHess = New SolverPlatform.DoubleMatrix
objHess.Create Rows:=2, Columns:=2
objHess.Value(0, 0) = 2
objHess.Value(0, 1) = 0
objHess.Value(1, 0) = 0
objHess.Value(1, 1) = 2
Dim QMatVal0 As SolverPlatform.DoubleMatrix
Set QMatVal0 = New SolverPlatform.DoubleMatrix
QMatVal0.Create Rows:=2, Columns:=2
Dim QMatVal1 As SolverPlatform.DoubleMatrix
Set QMatVal1 = New SolverPlatform.DoubleMatrix
QMatVal1.Create Rows:=2, Columns:=2
QMatVal1.Value(0, 0) = 0
QMatVal1.Value(0, 1) = 1
QMatVal1.Value(1, 0) = 1
QMatVal1.Value(1, 1) = 0
evaluator.Problem.Model.FcnHessValue(Function_Type_Objective, 0) =
objHess
evaluator.Problem.Model.FcnHessValue(Function_Type_Constraint, 0) =
matHess0
evaluator.Problem.Model.FcnHessValue(Function_Type_Constraint, 1) =
matHess1
evalJac_Evaluate = Engine_Action_Continue
End Function

```

[Back](#) to Solving a Nonlinear Problem with a Hessian Evaluator

---

## Using an Evaluator to Check the Progress of a Simulation Run

### C#

```
problem.Evaluators[Eval_Type.Simulation].OnEvaluate += new
 EvaluateEventHandler(EvalSimulation_Evaluate);

private Engine_Action EvalSimulation_OnEvaluate(Evaluator evaluator)
{
 MessageBox.Show("Simulation Index = " +
 evaluator.Problem.Solver.SimulationIndex);
 return Engine_Action.Continue;
}
```

[Back](#) to *Using an Evaluator to Check the Progress of a Simulation Run*.

### Procedural API in C

```
SolverEvaluatorAdd(prob, EvalSimulation, NULL,
 Eval_Type_Simulation);

_Engine_Action EvalSimulation(HProblem prob, void* refuser,
 _Eval_Type evaltype)
{
 int simindex = 0;
 SolverSimIndexGet(prob, &simindex);
 printf("Simulation # %d", simindex);
 return Engine_Action_Continue;
}
```

[Back](#) to *Using an Evaluator to Check the Progress of a Simulation Run*.

### COM in VB

```
Set EvalSimulation = New SolverPlatform.evaluator
prob.Evaluators.Item(Eval_Type_Simulation) = EvalSimulation

Private Function EvalSimulation_Evaluate(ByVal evaluator As
SolverPlatform.IEvaluator) As SolverPlatform.Engine_Action
 MsgBox ("Simulation Index = " & _
 evaluator.Problem.Solver.SimulationIndex)
 EvalTrial_Evaluate = Engine_Action_Continue
End Function
```

[Back](#) to *Using an Evaluator to Check the Progress of a Simulation Run.*

---

## Using an Evaluator to Compute Uncertain Functions in a Simulation Problem

### C#

```
problem.Evaluators[Eval_Type.Sample].OnEvaluate += new
EvaluateEventHandler(evalSim_Evaluate);
```

```
Engine_Action evalSim_OnEvaluate(Evaluator evaluator)
{
 Problem p = evaluator.Problem;
 double[] pVar = p.VarUncertain.Value.Array;
 double[] pFcn = p.FcnUncertain.Value.Array;
 pFcn[0] = pVar[0] + pVar[1];
 pFcn[1] = pVar[0] * pVar[1];
 p.FcnConstraint.Value.Array = pFcn;
 return Engine_Action.Continue;
}
```

[Back](#) to *Using an Evaluator to Compute Uncertain Functions in a Simulation Problem.*

### Procedural API in C

```
SolverEvaluatorAdd(prob, evalSim, NULL, Eval_Type_Sample);

_Engine_Action evalSim(HProblem prob, void* refuser, _Eval_Type
 evaltype)
{
 double var[2];
 double fcn[2];
 SolverVarValue(prob, L"Vars", var);
 fcn[0] = var[0] + var[1];
 fcn[1] = var[0] * var[1];
 SolverFcnValueSet(prob, L"Functions", fcn);
 return Engine_Action.Continue;
}
```

[Back](#) to *Using an Evaluator to Compute Uncertain Functions in a Simulation Problem.*

## COM in VB

```
Set evalSim = New SolverPlatform.evaluator
prob.Evaluators.Item(Eval_Type_Sample) = evalSim

Private Function evalSim_Evaluate(ByVal Evaluator As
SolverPlatform.IEvaluator) As Engine_Action
 Dim p As Problem
 p = Evaluator.Problem
 Dim pVar() As Double
 pVar = p.VarUncertain.Value.Array
 Dim Fcn(1) As Double
 pFcn(0) = Var(0) + Var(1)
 pFcn(1) = Var(0) * Var(1)
 p.FcnUncertain.Value.Array = pFcn
 evalSim_Evaluate = Engine_Action_Continue
End Function
```

[Back](#) to *Using an Evaluator to Compute Uncertain Functions in a Simulation Problem*

---

## Stochastic Information Packets

### C# - Creating a SIP from a Data Source

```
using (Problem prob = new Problem(Solver_Type.Simulate, 4, 1))
{
 DoubleMatrix mySipValues = new DoubleMatrix(4, 1000);
 mySipValues.NZBgn();
 mySipValues[0,0] = 9;
 mySipValues[0,1] = 10;
 mySipValues[0,2] = 6;
 mySipValues[0,3] = 8;
 mySipValues[0,4] = 11;
 mySipValues[1,0] = 1;
 mySipValues[1,1] = 2;
 mySipValues[1,2] = 1;
 mySipValues[1,3] = 1;
 mySipValues[1,4] = 1;
 mySipValues[2,0] = 3;
 mySipValues[2,1] = 2;
 mySipValues[2,2] = 5;
```

```

mySipValues[2,3] = 7;
mySipValues[2,4] = 3;
mySipValues[3,0] = 6;
mySipValues[3,1] = 6;
mySipValues[3,2] = 6;
mySipValues[3,3] = 6;
mySipValues[3,4] = 6;
mySipValues.NZEnd();
prob.Variables[0].AllTrials = mySipValues;

```

Alternatively, you could also use the PsiSip Distribution object to pass these same values as in the example below.

```

prob.Model.VarDistribution[0]=CDistribution("PsiSip(9,10,6,8,11)");
prob.Model.VarDistribution[1] = CDistribution("PsiSip(1,2,1,1,1)");
prob.Model.VarDistribution[2] = CDistribution("PsiSip(3,2,5,7,3)");
prob.Model.VarDistribution[3] = CDistribution("PsiSip(6,6,6,6,6)");

```

One more way to create and pass a SIP is by creating a new Distribution. In the example code below, we first create four double arrays (*SipValues1*, *SipValues2*, *SipValues3*, *SipValues4*) of size 5, create 4 distributions (*dist1*, *dist2*, *dist3*, *dist4*) of type *Distribution\_Type\_Sip*, set the number of parameters (sampled values) to 5, and then pass the sampled values to the four arrays, *SipValues1*, *SipValues2*, *SipValues3*, and *SipValues4*.

```

double SipValues1[5] = { 9, 10, 6, 8, 11 };
double SipValues2[5] = { 1, 2, 1, 1, 1 };
double SipValues3[5] = { 3, 2, 5, 7, 3 };
double SipValues4[5] = { 6, 6, 6, 6, 6 };
CDistribution dist1;
dist1.DistType = Distribution_Type_Sip;
dist1.NumParams = 5;
dist1.Params = SipValues1;
CDistribution dist2;
dist2.DistType = Distribution_Type_Sip;
dist2.NumParams = 5;
dist2.Params = SipValues2;
CDistribution dist3;
dist3.DistType = Distribution_Type_Sip;
dist3.NumParams = 5;
dist3.Params = SipValues3;
CDistribution dist4;
dist4.DistType = Distribution_Type_Sip;
dist4.NumParams = 5;

```



```
dist4.Params = SipValues4;
```

```
 prob.Evaluators[Eval_Type.Sample].OnEvaluate += new
 EvaluateEventHandler(Simulation_OnEvaluate);

 prob.Solver.Simulate();

//See the chapter, Calling the Solver SDK through C# to see example
//code obtaining the simulation status and results.
}
```

```
private Engine_Action Simulation_OnEvaluate(Evaluator evaluator)
{
 Problem p = evaluator.Problem;
 double[] pVar = p.VarUncertain.Value.Array;
 double[] pFcn = p.FcnUncertain.Value.Array;
 double bars, beans, nuggets, eclairs
 pFcn[0]= pVar[0] + pVar[1];
 pFcn[1] = pVar[0] * pVar[1];
 p.FcnUncertain.Value.Array = pFcn;
 return Engine_Action.Continue;
}
```

## C# - Creating a SIP inside the Evaluator

```
using (Problem prob = new Problem(Solver_Type.Simulate, 0, 1))
{
 prob.Solver.NumTrials = 5;
 prob.Evaluators[Eval_Type.Sample].OnEvaluate += new
 EvaluateEventHandler(SipSimulation_OnEvaluate);
 prob.Solver.Simulate();

//See the chapter, Calling the Solver SDK through C# to see
//example code obtaining the simulation status and results.
}
```

```
//Global Variables
```

```
int i;
double[] barsData = new double [5] {9,10,6,8,11};
double[] beansData = new double [5]{1,2,1,1,1};
double[] nuggetsData = new double [5]{3,2,5,1,3};
double[] eclairsData = new double [5]{6,6,6,6,6};
```

```
Engine_Action SipSimulation_OnEvaluate(Evaluator evaluator)
{
```

```

 Problem p = evaluator.Problem;
 double[] pFcn = p.FcnUncertain.Value.Array;
 double bars, beans, nuggets, eclairs;
 bars = barsData[i] * (2.5 - 1.5);
 beans = beansData[i] * (1.75 - 1.05);
 nuggets = nuggetsData[i] * (3.5 - 2.25);
 eclairs = eclairsData[i] * (4 - 2.95);
 pFcn[0] = bars + beans + nuggets + eclairs;
 i++;
 return Engine_Action.Continue;
 }

```

## C# - Creating a SIP from a PsiDistribution

```

using (Problem prob = new Problem(Solver_Type.Simulate, 4, 1))
{
 prob.Model.VarDistribution[0] = new
Distribution("PsiNormal(0,2)");
 prob.Model.VarDistribution[1] = new
Distribution("PsiBeta(1.5,3)");
 prob.Model.VarDistribution[2] = new Distribution("PsiGamma(4,1)");
 prob.Model.VarDistribution[3] = new
 Distribution("PsiTriangGen(1,2,4,0.1,0.9)");
 prob.Model.VarDistribution[0].ControlParams[1] = 0;
 prob.Model.VarDistribution[0].ControlParams[2] = 20;
 prob.Model.VarDistribution[0].ControlParams[1] = 0;
 prob.Model.VarDistribution[0].ControlParams[2] = 20;
 prob.Model.VarDistribution[0].ControlParams[1] = 0;
 prob.Model.VarDistribution[0].ControlParams[2] = 20;
 prob.Model.VarDistribution[0].ControlParams[1] = 0;
 prob.Model.VarDistribution[0].ControlParams[2] = 20;
 double[] SipValues1 = new double[5];
 double[] SipValues2 = new double[5];
 double[] SipValues3 = new double[5];
 double[] SipValues4 = new double[5];
 Distribution dist1 = new Distribution();
 dist1.DistType = Distribution_Type.Sip;
 dist1.NumParams = 5;
 dist1.Params.Array = SipValues1;
 Distribution dist2 = new Distribution();

```

```

dist2.DistType = Distribution_Type.Sip;
dist2.NumParams = 5;
dist2.Params.Array = SipValues2;
Distribution dist3 = new Distribution();
dist3.DistType = Distribution_Type.Sip;
dist3.NumParams = 5;
dist3.Params.Array = SipValues3;
Distribution dist4 = new Distribution();
dist4.DistType = Distribution_Type.Sip;
dist4.NumParams = 5;
dist4.Params.Array = SipValues4;
prob.Solver.NumTrials = 5;
prob.Evaluators[Eval_Type.Sample].OnEvaluate +=new
 EvaluateEventHandler(SipSimulation_OnEvaluate);
prob.Solver.Simulate();
//See the chapter, Calling the Solver SDK through C# to see example
//code obtaining the simulation status and results.
}

```

```

Engine_Action SipSimulation_OnEvaluate(Evaluator evaluator)
{
 Problem p = evaluator.Problem;
 double[] pFcn = p.FcnUncertain.Value.Array;
 double bars, beans, nuggets, eclairs;
 bars = barsData[i] * (2.5 - 1.5);
 beans = beansData[i] * (1.75 - 1.05);
 nuggets = nuggetsData[i] * (3.5 - 2.25);
 eclairs = eclairsData[i] * (4 - 2.95);
 pFcn[0] = bars + beans + nuggets + eclairs;
 return Engine_Action_Continue;
}

```

## Procedural API - Creating a SIP from a Data Source

```

void sipexample1()
{
 _Simulate_Status status;
 HProblem prob;
 double alltrials[] = {9,10,6,8,11,1,2,1,1,1,3,2,5,7,3,6,6,6,6,6};
 SolverProblem(&prob);
}

```

```

SolverNumTrialsSet(prob,5);
SolverProbTypeSet(prob, Solver_Type_Simulate, Problem_Type_NA);
SolverVariableAdd(prob, L"Vars", Variable_Type_Uncertain, 4);
SolverFunctionAdd(prob, L"GrossProfit", Function_Type_Uncertain,
 1);
SolverVarAllTrialsSet (prob, L"AllTrials", Array_Order_ByRow,
 alltrials);
SolverEvaluatorAdd(prob, EvalSip, NULL, Eval_Type_Sample);
SolverSimulate(prob, & status);
//See the chapter, Calling the Solver SDK through the
//Procedural API to see example code obtaining the simulation status
//and results.
}

```

Alternatively, you could also use the PsiSip Distribution object to pass these same values as in the example above.

```

void sipexample1()
{
 _Simulate_Status status;
 HProblem prob;
 SolverProblem(&prob);
 SolverNumTrialsSet(prob,5);
 SolverProbTypeSet(prob, Solver_Type_Simulate, Problem_Type_NA);
 SolverVariableAdd(prob, L"Vars", Variable_Type_Uncertain, 4);
 SolverFunctionAdd(prob, L"GrossProfit", Function_Type_Uncertain, 1);
 prob.Model.VarDistribution[0] =
 CDistribution("PsiSip(9,10,6,8,11)");
 prob.Model.VarDistribution[1] = CDistribution("PsiSip(1,2,1,1,1)");
 prob.Model.VarDistribution[2] = CDistribution("PsiSip(3,2,5,7,3)");
 prob.Model.VarDistribution[3] = CDistribution("PsiSip(6,6,6,6,6)");
 SolverEvaluatorAdd(prob, EvalSip, NULL, Eval_Type_Sample);
 SolverSimulate(prob, & status);
 ...
}

```

```

Engine_Action EvalSip(HProblem prob, void* refuser, _Eval_Type
evaltype)
{
 double pVar[4];
 double pFcn[1];
 double bars, beans, nuggets, eclairs;

```

```

SolverVarValue(prob, L"Vars", pVar);
bars = pVar[0] * (2.5 - 1.5);
beans = pVar[1] * (1.75 - 1.05);
nuggets = pVar[2] * (3.5 - 2.25);
eclairs = pVar[3] * (4 - 2.95);
pFcn[0] = bars + beans + nuggets + eclairs;
SolverFcnValueSet(prob, L"Funcs", pFcn);
return Engine_Action_Continue;
}

```

## Procedural API – Creating a Sip inside the Evaluator

```

void sipexample()
{
 _Simulate_Status status;
 HProblem prob;
 SolverProblem(&prob);
 SolverNumTrialsSet(prob, 5);
 SolverProbTypeSet(prob, Solver_Type_Simulate, Problem_Type_NA);
 SolverFunctionAdd(prob, L"GrossProfit", Function_Type_Uncertain,
 1);
 SolverEvaluatorAdd(prob, EvalSip, NULL, Eval_Type_Sample);
 SolverSimulate(prob, &status);
 //See the chapter, Calling the Solver SDK through the Procedural API
 //to see example code obtaining the simulation status //and results.
}

```

```

_Engine_Action EvalSip(HProblem prob, void* refuser, _Eval_Type
evaltype)
{
 double pFcn[1];
 double bars, beans, nuggets, eclairs;
 bars = barsData[i] * (2.5 - 1.5);
 beans = beansData[i] * (1.75 - 1.05);
 nuggets = nuggetsData[i] * (3.5 - 2.25);
 eclairs = eclairsData[i] * (4 - 2.95);
 pFcn[0] = bars + beans + nuggets + eclairs;
 SolverFcnValueSet(prob, L"Funcs", pFcn);
 i++;
 return Engine_Action_Continue;
}

```

```
}
```

## Procedural API - Creating a SIP from a PsiDistribution

```
//In the Procedural API, in order to create a SIP from a
//PsiDistribution, you must create two problems and call
//prob.Solver.Simulate twice. Once to run a simulation to sample
the //distributions and secondly to call the evaluator.
```

```
void sipexample()
{
 _Simulate_Status status1, status2;
 HProblem prob1;
 HProblem prob2;
 double alltrials[4000];
 double cparam[7];
 double varBarsTrials[5];
 double varBeansTrials[5];
 double varNuggetsTrials[5];
 double varEclairsTrials[5];
 cparam[0] = 0; //shift
 cparam[1] = 0; // lower cut;
 cparam[2] = 20; // upper cut
 cparam[3] = 0; // optional seed
 cparam[4] = -1e+30; // lower censor value
 cparam[5] = +1e+30; // upper censor value
 cparam[6] = 0; // basecase
 //create the first problem, prob1, with 4 uncertain variables
 SolverProblem(&prob1);
 SolverNumTrialsSet(prob1,5);
 SolverProbTypeSet(prob1, Solver_Type_Simulate, Problem_Type_NA);
 SolverVariableAdd(prob1, L"Vars", Variable_Type_Uncertain, 4);
 //set the distributions for the four uncertain variables
 SolverModDistInit(prob1, Dist_Var, 0, L"PsiNormal(0,1)");
 SolverModDistInit(prob1, Dist_Var, 1, L"PsiBeta(1.5,3)");
 SolverModDistInit(prob1, Dist_Var, 2, L"PsiGamma(4,1)");
 SolverModDistInit(prob1, Dist_Var, 3,
 L"PsiTriangGen(1,2,4,0.1,0.9)");
 //set the upper and lower bounds for each distribution
 SolverModDistControlParamsSet(prob1, Dist_Var, 0, cparam);
 SolverModDistControlParamsSet(prob1, Dist_Var, 1, cparam);
```

```

 SolverModDistControlParamsSet(prob1, Dist_Var, 2, cparam);
 SolverModDistControlParamsSet(prob1, Dist_Var, 3, cparam);
//run the first simulation to sample the values from the
//distributions

 SolverSimulate(prob1, &status1);
//creat the 2nd problem, prob2, with four uncertain variables and on
//uncertain function

 SolverProblem(&prob2);
 SolverProbTypeSet(prob2, Solver_Type_Simulate, Problem_Type_NA);
 SolverVariableAdd(prob2, L"VarBars", Variable_Type_Uncertain, 1);
 SolverVariableAdd(prob2, L"VarBeanss", Variable_Type_Uncertain,
1);
 SolverVariableAdd(prob2, L"VarNuggets", Variable_Type_Uncertain,
 1);
 SolverVariableAdd(prob2, L"VarEclairs", Variable_Type_Uncertain,
 1);
 SolverFunctionAdd(prob2, L"GrossProfit", Function_Type_Uncertain,
 1);

 SolverNumTrialsSet(prob2,5);
//creat the evaluator to compute the uncertain functions

 SolverEvaluatorAdd(prob2, EvalSip, NULL, Eval_Type_Sample);
//assign the sample trials from the uncertain variables to the
//alltrials array.

 SolverVarAllTrialsGet(prob1, L"Vars",
Array_Order_ByRow,alltrials);

 varBarsTrials[0] = alltrials[0];
 varBarsTrials[0] = alltrials[1];
 varBarsTrials[0] = alltrials[2];
 varBarsTrials[0] = alltrials[3];
 varBarsTrials[0] = alltrials[4];
 varBeansTrials[1] = alltrials[5];
 varBeansTrials[1] = alltrials[6];
 varBeansTrials[1] = alltrials[7];
 varBeansTrials[1] = alltrials[8];
 varBeansTrials[1] = alltrials[9];
 varNuggetsTrials[2] = alltrials[10];
 varNuggetsTrials[2] = alltrials[11];
 varNuggetsTrials[2] = alltrials[12];
 varNuggetsTrials[2] = alltrials[13];
 varNuggetsTrials[2] = alltrials[14];

```

```

varEclairsTrials[3] = alltrials[15];
varEclairsTrials[3] = alltrials[16];
varEclairsTrials[3] = alltrials[17];
varEclairsTrials[3] = alltrials[18];
varEclairsTrials[3] = alltrials[19];
//Use SolverVarAllTrialsSet to pass the sample trial values to the
//SDK
SolverVarAllTrialsSet (prob2, L"VarBars",Array_Order_ByRow ,
 varBarsTrials);
SolverVarAllTrialsSet (prob2, L"VarBeans",Array_Order_ByRow ,
 varBeansTrials);
SolverVarAllTrialsSet (prob2, L"VarNuggets",Array_Order_ByRow ,
 varNuggetsTrials);
SolverVarAllTrialsSet (prob2, L"VarEclairs",Array_Order_ByRow ,
 varEclairsTrials);
//Run the 2nd simulation. This time the sampled values from the
//uncertain variables in prob1 will be passed to the evaluator in
//prob2.
SolverSimulate(prob2,&status2);
//See the chapter, Calling the Solver SDK through the Procedural API
//to see example code obtaining the simulation status and results.
}

```

```

_Engine_Action EvalSip(HProblem prob, void* refuser, _Eval_Type
evaltype)

```

```

{
 double pVar[4];
 double pFcn[1];
 double bars, beans, nuggets, eclairs;
 SolverVarValue(prob, L"Vars", pVar);
 bars = pVar[0] * (2.5 - 1.5);
 beans = pVar[1] * (1.75 - 1.05);
 nuggets = pVar[2] * (3.5 - 2.25);
 eclairs = pVar[3] * (4 - 2.95);
 pFcn[0] = bars + beans + nuggets + eclairs;
 SolverFcnValueSet(prob, L"Funcs", pFcn);
 return Engine_Action_Continue;
}

```

## COM - Creating a SIP from a Data Source

```

Private Sub btnSimulation_Click()
 Dim prob As New SolverPlatform.Problem

```



```

prob.Solver.SolverType = Solver_Type_Simulate
prob.VarUncertain.Size = 4
prob.FcnUncertain.Size = 1
prob.Solver.NumTrials = 5
Dim mySipValues As New DoubleMatrix
With mySipValues
 .Create Rows:=4, Columns:=prob.Solver.NumTrials
 .Value(0, 0) = 9
 .Value(0, 1) = 10
 .Value(0, 2) = 6
 .Value(0, 3) = 8
 .Value(1, 0) = 1
 .Value(1, 1) = 2
 .Value(1, 2) = 1
 .Value(1, 3) = 1
 .Value(2, 0) = 3
 .Value(2, 1) = 2
 .Value(2, 2) = 5
 .Value(2, 3) = 7
 .Value(3, 0) = 6
 .Value(3, 1) = 6
 .Value(3, 2) = 6
 .Value(3, 3) = 6
End With
prob.Variables(0).AllTrials = mySipValues

```

Alternatively, you could use the PsiSIP Distribution object to pass these same values as in the example below.

```

Dim Sip1 As New Distribution
Sip1.Init "PsiSip(9,10,6,8,11)"
prob.Model.VarDistribution(4) = Sip1
Dim Sip2 As New Distribution
Sip2.Init "PsiSip(1,2,1,1,1)"
prob.Model.VarDistribution(4) = Sip2
Dim Sip3 As New Distribution
Sip3.Init "PsiSip(3,2,5,7,3)"
prob.Model.VarDistribution(4) = Sip3
Dim Sip4 As New Distribution
Sip4.Init "PsiSip(6,6,6,6,6)"

```

```

prob.Model.VarDistribution(4) = Sip4

Set SIPEvaluator = New Evaluator
prob.Evaluators(Eval_Type_Sample) = SIPEvaluator
prob.Solver.Simulate

'See the chapter, Calling the Solver SDK through COM to see example
'code obtaining the simulation status and results.

End Sub

Private Function SIPEvaluator_Evaluate(ByVal Evaluator As
SolverPlatform.IEvaluator) As SolverPlatform.Engine_Action

 Dim pVar() As Double
 pVar = Evaluator.Problem.VarUncertain.Value.Array
 Dim pFcn() As Double
 pFcn = Evaluator.Problem.VarUncertain.Value.Array
 Dim bars As Double
 Dim beans As Double
 Dim nuggets As Double
 Dim eclairs As Double
 bars = pVar(0) * (2.5 - 1.5)
 beans = pVar(1) * (1.75 - 1.05)
 nuggets = pVar(2) * (3.5 - 2.25)
 eclairs = pVar(3) * (4 - 2.95)
 pFcn(0) = bars + beans + nuggets + eclairs
 SIPEvaluator_Evaluate = Engine_Action_Continue

End Function

```

## COM - Creating a SIP from a PsiDistribution

```

Private Sub btnSimulation_Click()

 Dim prob As New SolverPlatform.Problem
 prob.Solver.SolverType = Solver_Type_Simulate
 prob.VarUncertain.Size = 4
 prob.FcnUncertain.Size = 1
 prob.Solver.NumTrials = 5
 Dim distNormal As New Distribution
 distNormal.Init "PsiNormal(0,1)"
 prob.Model.VarDistribution(0) = distNormal
 Dim distBeta As New Distribution
 distBeta.Init "PsiBeta(1,1)"
 prob.Model.VarDistribution(1) = distBeta

```

```

Dim distGamma As New Distribution
distGamma.Init "PsiGamma(4,1)"
prob.Model.VarDistribution(2) = distGamma
Dim distTriangGen As New Distribution
distTriangGen.Init "PsiTriangGen(1,2,4,0.1,0.9)"
prob.Model.VarDistribution(3) = distTriangGen
prob.Model.VarDistribution(0).ControlParams(0) = 0
prob.Model.VarDistribution(0).ControlParams(1) = 20
prob.Model.VarDistribution(1).ControlParams(0) = 0
prob.Model.VarDistribution(1).ControlParams(1) = 20
prob.Model.VarDistribution(2).ControlParams(0) = 0
prob.Model.VarDistribution(2).ControlParams(1) = 20
prob.Model.VarDistribution(3).ControlParams(0) = 0
prob.Model.VarDistribution(3).ControlParams(1) = 20
Dim SipValues1(4) As Double
Dim SipValues2(4) As Double
Dim SipValues3(4) As Double
Dim SipValues4(4) As Double
Dim dist1 As New Distribution
dist1.DistType = Distribution_Type_Sip
dist1.NumParams = 5
dist1.Params.Array = SipValues1
Dim dist2 As New Distribution
dist2.DistType = Distribution_Type_Sip
dist2.NumParams = 5
dist2.Params.Array = SipValues2
Dim dist3 As New Distribution
dist3.DistType = Distribution_Type_Sip
dist3.NumParams = 5
dist3.Params.Array = SipValues3
Dim dist4 As New Distribution
dist4.DistType = Distribution_Type_Sip
dist4.NumParams = 5
dist4.Params.Array = SipValues4
Set SIPEvaluator = New Evaluator
prob.Evaluators(Eval_Type_Sample) = SIPEvaluator
prob.Solver.Simulate

```

'See the chapter, **Calling the Solver SDK through COM** to see example  
'code obtaining the simulation status and results.

End Sub

```
Private Function SIPEvaluator_Evaluate(ByVal Evaluator As
SolverPlatform.IEvaluator) As SolverPlatform.Engine_Action
```

```
 Dim pVar() As Double
```

```
 pVar = Evaluator.Problem.VarUncertain.Value.Array
```

```
 Dim pFcn() As Double
```

```
 pFcn = Evaluator.Problem.VarUncertain.Value.Array
```

```
 Dim bars As Double
```

```
 Dim beans As Double
```

```
 Dim nuggets As Double
```

```
 Dim eclairs As Double
```

```
 bars = pVar(0) * (2.5 - 1.5)
```

```
 beans = pVar(1) * (1.75 - 1.05)
```

```
 nuggets = pVar(2) * (3.5 - 2.25)
```

```
 eclairs = pVar(3) * (4 - 2.95)
```

```
 pFcn(0) = bars + beans + nuggets + eclairs
```

```
 SIPEvaluator_Evaluate = Engine_Action_Continue
```

```
End Function
```

[Back](#) to *Stochastic Information Packets*

---

## GetCorrelation

### C#

```
//Create an array for correlation coefficients for the first
variable //and function blocks. Each block contains 3 elements.
```

```
double[] coeff = new double[3];
```

```
coeff = prob.Variables[0].GetCorrelation(0);
```

```
coeff = prob.Functions[0].GetCorrelation(0);
```

[Back](#) to GetCorrelation

### Procedural API

```
//Create an array for correlation coefficients of variable block
// "Vars" and function block "Funcs" which contain 3 vars and 3
functions, respectively.
```

```
int index = 0; //first variable
```

```
SolverVarGetCorrelation (prob, L"Vars", varindex, coeff);
```

```
SolverFcnGetCorrelation (prob, L"Funcs", fcnindex, coeff);
```

[Back](#) to GetCorrelation

## COM in VB

```
`Create an array for correlation coefficients
Dim coeff
coeff = prob.Variables(0).GetCorrelation(0)
coeff = prob.Functions(0).GetCorrelation(0)
```

[Back](#) to GetCorrelation

---

# GetFrequency

## C#

```
double[] binbounds = new double[]{ -150, -100, -50, 0, 50, 100,
150};

DoubleMatrix vargetfreq =
prob.VarUncertain.GetFrequency(Frequency_Type.Cumulative, binbounds)
DoubleMatrix funcgetfreq =
prob.FcnUncertain.GetFrequency(Frequency_Type.Cumulative,
binbounds);
```

## Procedural API

```
double binbounds[] = { -150, -100, -50, 0, 50, 100, 150 };
double binfrequency[8];

SolverVarGetFrequency (prob, L"Vars[0]", Frequency_Type_Cumulative,
7, binbounds, binfrequency);
SolverFcnGetFrequency (prob, L"Funcs[0]", Frequency_Type_Cumulative,
7, binbounds, binFrequency);
```

## COM in VB

```
Dim binbounds(7) As Double
binbounds (0) = -150
binbounds (1) = -100
binbounds (2) = -50
binbounds (3) = 0
binbounds (4) = 50
binbounds (5) = 100
binbounds (6) = 150

Dim vargetfreq As New DoubleMatrix
Dim funcgetfreq as New DoubleMatrix
```

```

vargetfreq =
prob.VarUncertain.GetFrequency(Frequency_Type.Cumulative, binbounds)
fcnggetfreq =
prob.FcnUncertain.GetFrequency(Frequency_Type.Cumulative, binbounds)
Back to GetFrequency

```

---

## Calling IISFind

### C#

```

if (status == Optimize_Status.Unfeasible)
{
 problem.Solver.IISFind();
 for (int i = 0; i < problem.Solver.OptIIS.NumBounds; i++)
 MessageBox.Show("Bound[" + problem.Solver.OptIIS.BoundIndex[i] + "],
 status is " + problem.Solver.OptIIS.BoundStatus[i] + "\r\n");
 for (int i = 0; i < problem.Solver.OptIIS.NumConstraints; i++)
 MessageBox.Show("Constraint[" +
 problem.Solver.OptIIS.ConstraintIndex[i] + "], status is " +
 problem.Solver.OptIIS.ConstraintStatus[i] + "\r\n");
}

```

[Back to Optimization Result Codes](#)

### Procedural API

```

int *iisconindex = NULL;
int *iisbndindex = NULL;
int iisnumcons, iisnumbnds;
_IIS_Status *iisconstatus = NULL;
_IIS_Status *iisbndstatus = NULL;
int i;
if (status == Optimize_Status_Unfeasible)
{
 SolverOptIISFind(prob, &iisnumcons, &iisnumbnds);
 iisconindex = (int*)malloc(iisnumcons*sizeof(int));
 iisbndindex = (int*)malloc(iisnumbnds*sizeof(int));
 iisconstatus =
 (_IIS_Status*)malloc(iisnumcons*sizeof(_IIS_Status));
 iisbndstatus =
 (_IIS_Status*)malloc(iisnumbnds*sizeof(_IIS_Status));
}

```

```

SolverOptIIS(prob, iisconindex, iisconstatus, iisbndindex,
iisbndstatus);
for(i = 0; i <iisnumbnds; i ++)
 printf("Bound[%d] status is %d\n", iisbndindex[i],
iisbndstatus[i]);
for(i = 0; i < iisnumcons; i ++)
 printf("Constraint[%d] status is %d\n", iisconindex[i],
iisconstatus[i]);
free(iisconindex);
free(iisbndindex);
free(iisconstatus);
free(iisbndstatus);
}

```

Back to [Optimization Result Codes](#)

## COM

```

If status = Optimize_Status_Unfeasible Then
 prob.Solver.IISFind
 For i = 0 To prob.Solver.OptIIS.NumBounds - 1
 MsgBox("Bound[" & prob.Solver.OptIIS.BoundIndex(i) & _
 "], status is " & prob.Solver.OptIIS.BoundStatus(i))
 Next
 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

```

Back to [Optimization Result Codes](#)

# Index

|                                                               |                              |                                                          |                        |
|---------------------------------------------------------------|------------------------------|----------------------------------------------------------|------------------------|
| advanced basis .....                                          | 270                          | discontinuous function.....                              | 261                    |
| alldifferent constraint 256, 274, 275, 277, 291, 296, 298     |                              | distance filter .....                                    | 293                    |
| automatic differentiation.....                                | 284, 287                     | diversity of population.....                             | 262                    |
| basic variable .....                                          | 283, 286                     | dual feasible .....                                      | 270                    |
| Bayesian test .....                                           | 256, 260                     | Dual Simplex method.....                                 | 269                    |
| best bound.....                                               | 297                          | DualTolerance SDK parameter .....                        | 281                    |
| BFGS method (Search).....                                     | 284, 287, 288                | Error in model.....                                      | 250, 255               |
| binding constraints .....                                     | 284, 287, 288                | error value.....                                         | 254                    |
| biological organisms .....                                    | 23                           | Estimates option.....                                    | 283, 286               |
| Bounds Improvement option.....                                | 271                          | evaluation license .....                                 | 250, 251               |
| bounds on the variables ... 256, 257, 258, 288, 289, 291, 292 |                              | evolutionary algorithm .....                             | 23, 261                |
| Branch & Bound .....                                          | 255, 261, 269, 270, 271, 297 | Evolutionary Solver... 252, 253, 254, 255, 256, 261, 262 |                        |
| Branch & Bound method.....                                    | 268, 276                     | Evolutionary Solver Stopping Conditions .....            | 261                    |
| Bypass Solver Reports option .....                            | 274                          | Feasibility Report.....                                  | 253                    |
| candidate solutions .....                                     | 290                          | field-installable Solver engines .....                   | 250                    |
| Central choice (Derivatives) .....                            | 275, 284, 287                | Filtered Local Search.....                               | 293                    |
| clique.....                                                   | 270                          | finite differencing.....                                 | 275, 284, 287          |
| clustering .....                                              | 288                          | finite precision .....                                   | 259, 265, 266          |
| Conjugate choice (Search).....                                | 284, 287, 288                | fitness .....                                            | 23, 252, 262, 290, 295 |
| conjugate gradient method.....                                | 284, 287, 288                | Fix Nonsmooth Variables .....                            | 293                    |
| constraint handling methods .....                             | 294                          | Forward choice (Derivatives).....                        | 275, 284, 287          |
| constraint left hand side.....                                | 265                          | Generalized Reduced Gradient .....                       | 283, 286               |
| constraint right hand side.....                               | 265                          | genetic algorithm.....                                   | 22, 261                |
| Convergence option .....                                      | 252, 260, 262, 282, 285, 290 | global optimization.....                                 | 256, 260               |
| convex problem.....                                           | 259                          | globally optimal solution .....                          | 256, 260, 262          |
| correlation.....                                              | 236                          | Gomory Cuts.....                                         | 272                    |
| correlation coefficient.....                                  | 236, 237                     | Gomory Passes.....                                       | 273                    |
| crossover.....                                                | 23, 291, 292, 295            | good solution .....                                      | 261                    |
| custom result codes .....                                     | 250                          | GRG Nonlinear Solver .....                               | 261                    |
| cut generation.....                                           | 272                          | GRG Solver stopping conditions.....                      | 259, 282, 285          |
| cycling.....                                                  | 260                          | Help                                                     |                        |
| Dealing with poor scaling.....                                | 259                          | Solver.....                                              | 260                    |
| degeneracy.....                                               | 252                          | Hessian matrix .....                                     | 284, 287               |
| degenerate problem .....                                      | 260                          | heuristic stopping rules.....                            | 262                    |
| dependent variables.....                                      | 236                          | hybrid Evolutionary Solver.....                          | 294                    |
| Derivatives.....                                              | 284, 287                     | IF function .....                                        | 261                    |
| for Quadratic Solver .....                                    | 275                          | incumbent .....                                          | 268, 297, 298          |
| Derivatives option .....                                      | 294                          | independent variables .....                              | 236                    |
| <b>Derivatives SDK parameter</b> .....                        | 292, 294                     | inducing correlation.....                                | 236                    |
| Deterministic Pattern Search.....                             | 292                          | integer constraint.....                                  | 265                    |
|                                                               |                              | Integer Cutoff option .....                              | 269, 277, 297          |



|                                                                |                         |                                                     |                                   |
|----------------------------------------------------------------|-------------------------|-----------------------------------------------------|-----------------------------------|
| Integer Options dialog .....                                   | 275, 278, 296           | non-smooth function.....                            | 261                               |
| integer programming .....                                      | 297                     | non-smooth problems .....                           | 261                               |
| Integer Tolerance option.....                                  | 255, 268, 276, 297      | optimal solution.....                               | 251, 297                          |
| Interpreter .....                                              | 284, 287                | Optimality conditions .....                         | 259                               |
| iterations .....                                               | 282, 285                | Optimality Fixing option .....                      | 271                               |
| Iterations option .....                                        | 252, 265                | Options                                             |                                   |
| Jacobian matrix .....                                          | 283, 284, 286, 287      | Solver.....                                         | 258, 267, 268, 274, 280, 282, 285 |
| Knapsack Cuts .....                                            | 273                     | OptQuest Solver .....                               | 254                               |
| Knapsack Passes.....                                           | 273                     | orders of magnitude .....                           | 258, 266                          |
| Kuhn-Tucker conditions .....                                   | 259                     | Other Nonlinear Options.....                        | 283, 286                          |
| left hand side                                                 |                         | partial derivative.....                             | 283, 284, 286, 287                |
| constraints .....                                              | 265                     | Pearson product moment correlation coefficient..... | 236                               |
| license record .....                                           | 250, 251                | Pivot Tolerance .....                               | 267                               |
| Limit Options dialog.....                                      | 262, 290                | poorly scaled models .....                          | 252, 258, 259, 260, 266           |
| limitations                                                    |                         | population of solutions .....                       | 23                                |
| non-smooth optimization .....                                  | 261                     | Population Size option.....                         | 252, 263, 289, 290                |
| smooth nonlinear optimization .....                            | 259                     | Precision and Integer Constraints .....             | 265                               |
| linear function .....                                          | 253                     | Precision and Regular Constraints.....              | 265                               |
| Linear Local Gradient Search .....                             | 293                     | Precision option.....                               | 259, 265, 290                     |
| linear programming                                             |                         | Premium Solver Platform .....                       | 256, 260, 269, 270                |
| solution .....                                                 | 251                     | primal feasible.....                                | 270                               |
| linearity test .....                                           | 261                     | PrimalTolerance SDK parameter.....                  | 281                               |
| local search .....                                             | 295                     | Probing .....                                       | 270                               |
| Local Search options .....                                     | 292                     | Probing/Feasibility option.....                     | 270                               |
| locally optimal solution .....                                 | 256, 259, 260, 263      | problem size .....                                  | 254                               |
| Locally Versus Globally Optimal Solutions .....                | 259                     | Problems with Poorly Scaled Models.....             | 258                               |
| loss of diversity .....                                        | 262                     | pseudocost branching .....                          | 271, 272                          |
| LP/Quadratic Solver .....                                      | 267, 268, 274           | PSI Distribution functions.....                     | 178, 237                          |
| Max Feasible Sols option.....                                  | 252, 255, 262, 290, 295 | PSI Property functions.....                         | 237, 245                          |
| Max Feasible Solutions option.....                             | 276                     | <i>PsiAbsDev</i> .....                              | 239                               |
| Max Integer Sols option.....                                   | 255, 268, 297           | PsiBernoulli .....                                  | 214                               |
| Max Subproblems option. 252, 255, 262, 268, 275, 290, 295, 296 |                         | PsiBeta.....                                        | 179                               |
| Max Time option.....                                           | 255, 264, 266, 290      | PsiBetaGen .....                                    | 181                               |
| Max Time w/o Improvement option.....                           | 252, 262, 296           | PsiBetaSubj.....                                    | 182                               |
| memory.....                                                    | 250, 255                | PsiBinomial.....                                    | 215                               |
| virtual.....                                                   | 250, 255                | <i>PsiBVAR</i> .....                                | 239                               |
| merit filter .....                                             | 293                     | PsiCauchy .....                                     | 183                               |
| mixed-integer programming .....                                | 268, 276, 296, 297      | PsiChiSquare.....                                   | 184                               |
| solution .....                                                 | 251                     | <i>PsiCITrials</i> .....                            | 240                               |
| Monte Carlo                                                    |                         | <i>PsiCorrelation</i> .....                         | 237                               |
| simulation.....                                                | 178, 236                | PsiCount .....                                      | 242                               |
| multistart methods.....                                        | 256, 259, 260, 288, 289 | PsiCumul .....                                      | 224                               |
| mutation.....                                                  | 23, 291, 292, 295       | <i>PsiCVAR</i> .....                                | 240                               |
| Mutation Rate option.....                                      | 252, 263, 291           | PsiDiscrete .....                                   | 225                               |
| natural evolution.....                                         | 23                      | PsiDisUniform .....                                 | 226                               |
| natural selection .....                                        | 23                      | PsiErf.....                                         | 185                               |
| Newton choice (Search).....                                    | 284, 287, 288           | PsiErlang.....                                      | 186                               |
| no feasible solution.....                                      | 253                     | PsiExponential .....                                | 187                               |
| nonbasic variable.....                                         | 283, 286                | <i>PsiFrequency</i> .....                           | 238                               |
| non-convex problem.....                                        | 259                     | PsiGamma.....                                       | 188                               |
| nonlinear function .....                                       | 253                     | PsiGeneral .....                                    | 227                               |
| Nonlinear Gradient Search.....                                 | 293                     | PsiGeometric.....                                   | 216                               |
| nonlinear optimization                                         |                         | PsiHistogram.....                                   | 228                               |
| solution .....                                                 | 251, 252                | PsiHyperGeo .....                                   | 218                               |
| nonlinear problems.....                                        | 259                     | PsiIntUniform .....                                 | 219                               |
|                                                                |                         | PsiInvNormal .....                                  | 189                               |

|                                         |                    |                                                     |                              |
|-----------------------------------------|--------------------|-----------------------------------------------------|------------------------------|
| <i>PsiKurtosis</i> .....                | 240                | Require Bounds on Variables option.....             | 256, 289, 292                |
| <i>PsiLaplace</i> .....                 | 191                | result codes                                        |                              |
| <i>PsiLogarithmic</i> .....             | 220                | custom.....                                         | 250                          |
| <i>PsiLogistic</i> .....                | 191                | right hand side                                     |                              |
| <i>PsiLogLogistic</i> .....             | 192                | constraints .....                                   | 265                          |
| <i>PsiLogNorm2</i> .....                | 195                | satisfied                                           |                              |
| <i>PsiLogNormal</i> .....               | 194                | constraints .....                                   | 265                          |
| <i>PsiMax</i> .....                     | 241                | Savage, Sam.....                                    | 229                          |
| <i>PsiMaxExtreme</i> .....              | 196                | scaling .....                                       | 258, 260, 265, 266           |
| <i>PsiMean</i> .....                    | 241                | Scaling SDK parameter .....                         | 265                          |
| <i>PsiMeanCI</i> .....                  | 241                | Search option .....                                 | 283, 284, 286, 287           |
| <i>PsiMin</i> .....                     | 241                | second order cone programming .....                 | 251                          |
| <i>PsiMinExtreme</i> .....              | 196                | selection process .....                             | 23, 291, 292, 295            |
| <i>PsiMode</i> .....                    | 242                | Set Cell values do not converge .....               | 252                          |
| <i>PsiMyerson</i> .....                 | 197                | sexual reproduction .....                           | 23                           |
| <i>PsiNegBinomial</i> .....             | 221                | Show Trial Solution dialog .....                    | 252                          |
| <i>PsiNormal</i> .....                  | 200, 246           | Simplex method .....                                | 267                          |
| <i>PsiPareto</i> .....                  | 201                | Simplex Solver Options.....                         | 267, 268, 274                |
| <i>PsiPareto2</i> .....                 | 202                | SIP .....                                           | 229                          |
| <i>PsiPearson5</i> .....                | 202                | SLURP .....                                         | 229, 236                     |
| <i>PsiPearson6</i> .....                | 204                | SOC Barrier Solver .....                            | 280, 282, 285                |
| <i>PsiPercentile</i> .....              | 242                | SOC Barrier Solver Options .....                    | 280, 282, 285                |
| <i>PsiPert</i> .....                    | 205                | Solve Without Integer Constraints option .....      | 274, 275, 277, 296, 298      |
| <i>PsiPoisson</i> .....                 | 222                | Solver cannot improve the current solution .....    | 252, 262                     |
| <i>PsiRayleigh</i> .....                | 206                | Solver converged in probability .....               | 256                          |
| <i>PsiSeed</i> .....                    | 245                | Solver could not find a feasible solution.....      | 253                          |
| <i>PsiSemiDev</i> .....                 | 242                | Solver encountered an error computing derivatives . | 256                          |
| <i>PsiSemiVar</i> .....                 | 243                | Solver encountered an error value .....             | 254                          |
| <i>PsiSkewness</i> .....                | 243                | Solver engine size limits.....                      | 254                          |
| <i>PsiStdDev</i> .....                  | 243                | Solver has converged to the current solution ....   | 251, 262                     |
| <i>PsiStdDevCI</i> .....                | 244                | Solver Options dialog.....                          | 259, 267, 274, 280, 282, 285 |
| <i>PsiStudent</i> .....                 | 207                | Solver Result Message .....                         | 250                          |
| <i>PsiTarget</i> .....                  | 244                | SOS constraint .....                                | 270                          |
| <i>PsiTriangGen</i> .....               | 208                | Spearman rank order correlation coefficient.....    | 237                          |
| <i>PsiTriangular</i> .....              | 210                | Special Ordered Set.....                            | 270                          |
| <i>PsiTruncate</i> .....                | 245, 246           | Stochastic Information Packet.....                  | 229                          |
| <i>PsiUniform</i> .....                 | 212                | Stochastic Libraries .....                          | 229                          |
| <i>PsiVariance</i> .....                | 244                | Stopping conditions                                 |                              |
| <i>PsiWeibull</i> .....                 | 213                | GRG Solver.....                                     | 259, 282, 285                |
| Quadratic choice (Estimates) .....      | 283, 286           | <b>StrongBranching SDK parameter</b> .....          | 277                          |
| quadratic objective .....               | 275                | subproblem                                          |                              |
| quadratic programming                   |                    | Branch & Bound.....                                 | 269, 277, 297                |
| solution .....                          | 251                | swapping.....                                       | 250, 255                     |
| Quadratic Solver .....                  | 275                | Tangent choice (Estimates).....                     | 283, 286                     |
| quasi-Newton method.....                | 284, 287, 288      | Tolerance option .....                              | 252, 262, 295                |
| random number generator.....            | 289, 291           | Too many adjustable cells.....                      | 254                          |
| Random Seed option .....                | 289, 291           | Too many constraints .....                          | 254                          |
| Randomized Local Search .....           | 292                | Too many integer adjustable cells .....             | 254                          |
| Recognize Linear Variables option ..... | 283                | topographic search .....                            | 288                          |
| Reduced Cost Tolerance.....             | 267                | Topographic Search option.....                      | 288                          |
| Reduced Costs .....                     | 267                | trial license.....                                  | 250, 251                     |
| redundant constraints.....              | 252                | Use Automatic Scaling option .....                  | 258, 260, 265, 266           |
| relaxation                              |                    | Use Dual Simplex for Subproblems option .....       | 269                          |
| of integer problem .....                | 268, 276, 296, 297 | Variable Reordering option.....                     | 271                          |
| repair of a solution .....              | 23                 |                                                     |                              |

