

# Premium Solver Platform

## User Guide

Examples.xlsx - Microsoft Excel

Home Insert Page Layout Formulas Data Review View Add-Ins

Function Wizard Recently Used Financial Date & Time Logical Text Name Manager Name a Range Use in Formula Create from Selection Formula Auditing Calculation

Portfolio\_Variance  $=\text{QUADPRODUCT}(\text{Allocations}, \text{B8:F8}, \text{Stock\_Covariances})$

1 **Portfolio Optimization - Markowitz Method**

2 This Solver model uses the QUADPRODUCT function at cell I14 to compute the portfolio variance.

3 It can be solved for the minimum variance using either the GRG nonlinear solver or the Quadratic Solver.

	Stock 1	Stock 2	Stock 3	Stock 4	Stock 5	Total
Portfolio %	20.00%	20.00%	20.00%	20.00%	20.00%	100.00%
Expected Return	7.00%	8.00%	9.50%	6.50%	14.00%	
Linear QP Terms	0	0	0	0	0	

Variance/Covariance Matrix

	Stock 1	Stock 2	Stock 3	Stock 4	Stock 5	
Stock 1	2.50%	0.10%	0.25%	-0.50%	0.25%	
Stock 2	0.10%	2.00%	-0.10%	1.20%	-0.85%	
Stock 3	0.25%	-0.10%	2.00%	0.65%	0.75%	Variance 0.60%
Stock 4	-0.50%	1.20%	0.65%	2.00%	0.50%	Std. Dev. 7.75%
Stock 5	0.25%	-0.85%	0.75%	0.50%	2.00%	Return 9.00%

Ready


Version 7.0

For Use With Excel 2000-2007


# *Premium Solver Platform*

# User Guide

**Solver Parameters V7.0**

Set Cell:  

Equal To: ☐ Max ☒ Min ☐ Value Of:

By Changing Variable Cells:  

Subject to the Constraints:

Portfolio\_Return >= 0.095  
Total\_Portfolio = 1

Standard LP/Quadratic

Standard GRG Nonlinear  
Standard LP/Quadratic  
Standard Evolutionary  
Standard Interval Global  
Standard SOCP Barrier  
KNITRO Solver  
Large-Scale GRG Solver  
Large-Scale LP Solver  
Large-Scale SQP Solver  
MOSEK Solver Engine  
OptQuest Solver  
XPRESS Solver Engine

**Solver Model**

Original Transformed Diagnosis Options

QP Convex	Variables	Functions	NonZeroes
All	<input type="text" value="5"/>	<input type="text" value="3"/>	<input type="text" value="15"/>
Smooth	<input type="text" value="5"/>	<input type="text" value="3"/>	<input type="text" value="15"/>
Quadratic	<input type="text" value="5"/>	<input type="text" value="1"/>	<input type="text" value="5"/>
Linear	<input type="text" value="0"/>	<input type="text" value="2"/>	<input type="text" value="10"/>
Bounds	<input type="text" value="10"/>	Sparsity %	<input type="text" value="100"/>
Integers	<input type="text" value="0"/>	Total Cells	<input type="text" value="43"/>

Solve With ☒ PSI Interpreter ☐ Excel Interpreter

Check For  
☐ Gradients  
☐ Structure  
☒ Convexity  
☐ Automatic



## Copyright

Software copyright 1991-2006 by Frontline Systems, Inc.

Portions copyright 1989 by Optimal Methods, Inc. ; portions copyright 2002 by Masakazu Muramatsu.

User Guide copyright 2006 by Frontline Systems, Inc.

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 on the following pages.

## Trademarks

Premium Solver, Premium Solver Platform, Risk Solver Engine, and Solver Platform SDK are trademarks of Frontline Systems, Inc.

Windows and Excel are trademarks of Microsoft Corporation.

Lotus and 1-2-3 are trademarks of IBM Corporation.

KNITRO is a trademark of Ziena Optimization, Inc.

MOSEK is a trademark of MOSEK ApS.

OptQuest is a trademark of OptTek Systems, Inc.

Xpress<sup>MP</sup> is a trademark of Dash Optimization, Inc.

## How to Order

Contact Frontline Systems, Inc., P.O. Box 4288, Incline Village, NV 89450.

Tel (775) 831-0300 Fax (775) 831-0314 Toll-Free (888) 831-0333

Email [info@solver.com](mailto:info@solver.com) Web <http://www.solver.com>

---

## **SOFTWARE LICENSE AND LIMITED WARRANTY**

This is an agreement between Frontline Systems, Inc. ("Frontline") and the person or organization acquiring a license ("Licensee") to use the computer program products described in this User Guide (the "Software"), in exchange for Licensee's payment to Frontline. Licensee may designate the individual(s) who will use the Software from time to time, in accordance with the terms of this agreement. Unless replaced by a separate written agreement signed by an officer of Frontline, this agreement, including the Software License, Limited Warranty, and U.S. Government Restricted Rights sections below, shall govern Licensee's use of the Software; by accepting delivery of the Software or allowing Use of the Software, Licensee accepts all terms and conditions of this agreement, and agrees that this agreement supersedes the terms and conditions of any purchase order issued in connection with the license purchase.

"Use" of the Software means the use of any of its functions to define, analyze, solve (optimize, simulate, etc.) and/or obtain results for a single user-defined model. Use with multiple models at the same time, whether on one computer or multiple computers, requires either a Flexible Use License or multiple Standalone Licenses. Use occurs only during the time that the computer's processor is executing the Software; it does not include time when the Software is loaded in to memory without being executed. The minimum time period for Use on any one computer shall be ten (10) minutes, but may be longer depending on the Software function used and the size and complexity of the model.

### **STANDALONE LICENSE**

If Licensee pays for a Standalone License, Frontline grants to Licensee the right to Use the Software on one computer (the "PC") at a time, and will provide Licensee with a license code enabling such Use. The Software may be stored on one or more computers, servers or storage devices, but it may be Used only on the PC. Use of the Software may depend upon unique components of the PC, such as its hard disk ID or MAC address; in the event these components fail, Frontline will provide Licensee with a new license code, enabling Use with replacement components, at no charge. A Standalone License may be transferred to a different PC while the first PC remains in operation only if (i) Licensee requests a new license code from Frontline, (ii) Licensee certifies in writing that the Software will no longer be Used on the first PC, and (iii) Licensee pays a license transfer fee, unless such fee is waived by Frontline.

### **FLEXIBLE USE LICENSE**

If Licensee pays for a Flexible Use License, Frontline grants to Licensee the right to Use the Software as described in this paragraph, and will provide Licensee with License Server software and a license code enabling such Use. For purposes of this agreement, a "Network" is a group of computers interconnected by any networking technology that supports the TCP/IP protocol or the IPX/SPX protocol. The Software may be (i) stored on one or more computers, servers or storage devices on the Network, (ii) accessed by and copied into the memory of other computers on the Network, and (iii) Used on any of the computers on the Network, provided that only one Use occurs at any one time. Licensee must install and run the License Server software on one of the computers on the Network (the "LS"); other computers will temporarily obtain the right to Use the Software from the License Server. Operation of the License Server may depend upon unique components of the LS, such as its hard disk ID or MAC address; in the event these components fail, Frontline will provide Licensee with a new license code, enabling operation of the License Server with replacement components, at no charge. The License Server software may be transferred to a different LS while the first LS remains in operation only if (i) Licensee requests a new license code from Frontline, (ii) Licensee certifies in writing that the License Server will no longer be run on the first LS, and (iii) Licensee pays a license transfer fee, unless such fee is waived by Frontline.

### **ADDITIONAL TERMS**

This agreement does not grant to Licensee the right to make copies of the Software or otherwise enable use of the Software in any manner other than as described above, by any persons or on any computers except as described above, or by any entity other than Licensee. Licensee agrees that it will not rent or lease the Software, nor "share" use of the Software with anyone else, nor make the Software available over the Internet, a company or institutional intranet, or any similar networking technology, except as explicitly provided above in the case of a Flexible Use License. Licensee agrees that it will not attempt to alter or circumvent license control features of the Software or the License Server, nor reverse compile or reverse engineer the Software or the License Server. This agreement may be assigned to any entity that succeeds by operation of law to Licensee or that purchases all or substantially all of Licensee's assets (the "Successor"), provided that Frontline is notified of the transfer, and that Successor agrees to all terms and conditions of this agreement.

---

## **COPYRIGHT WARNING**

The Software is protected by United States copyright laws and international copyright treaty provisions. It is unlawful for any person or entity to copy or use the Software, except as permitted by the license explicitly granted by Frontline. For the LP/Quadratic Solver only: Source code is available, as part of an open source project, for portions of this software; please contact Frontline for information if you want to obtain this copyrighted source code. The law provides for both civil and criminal penalties for copyright infringement.

## **LIMITED WARRANTY**

Frontline Systems, Inc. ("Frontline") warrants that the CD-ROM, diskette or other media on which the Software is distributed and the accompanying User Guide (collectively, the "Goods"), but not the digital or printed content recorded thereon, is free from defects in materials and workmanship under normal use and service for a period of ninety (90) days after purchase, and any implied warranties on the Goods are also limited to ninety (90) days. **SOME STATES DO NOT ALLOW LIMITATIONS ON THE DURATION OF AN IMPLIED WARRANTY, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.** Frontline's entire liability and your exclusive remedy under this warranty shall be, at Frontline's option, either (i) return of the purchase price or (ii) replacement of the Goods that do not meet Frontline's limited warranty. You may return any defective Goods under warranty to Frontline or to your authorized dealer, either of which will serve as a service and repair facility.

If you purchase an Annual Support Contract from Frontline, then Frontline warrants, during the contract term, that the Software will perform substantially as described in the User Guide, when it is properly used as described in the User Guide. Frontline's entire liability and your exclusive remedy under this warranty shall be to make reasonable commercial efforts to correct any "bugs" (failures to perform as so described) reported by you, and to timely provide such corrections in the Software to you. If you do not purchase an Annual Support Contract from Frontline, or if you allow your Annual Support Contract to expire, then **THE SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.**

Whether or not you purchase an Annual Support Contract from Frontline, you understand and agree that any results obtained through your use of the Software are entirely dependent on your design and implementation of an optimization or simulation model, for which you are entirely responsible, even if you seek advice on modeling from Frontline. You understand and agree that **THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE AS USED WITH YOUR OPTIMIZATION OR SIMULATION MODEL IS ASSUMED BY YOU.**

**EXCEPT AS PROVIDED ABOVE, FRONTLINE DISCLAIMS, AND WITHOUT EXCEPTION ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE. THIS WARRANTY GIVES YOU SPECIFIC RIGHTS, AND YOU MAY HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.**

**IN NO EVENT SHALL FRONTLINE OR ITS SUPPLIERS HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE SOFTWARE OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.** In states that allow the limitation but not the exclusion of such liability, Frontline and its Suppliers' liability to you for damages of any kind is limited to the price of one copy of the Goods and one Standalone License to use the Software.

## **U.S. GOVERNMENT RESTRICTED RIGHTS**

The Software and Media are provided with **RESTRICTED RIGHTS**. Use, duplication or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of The Rights in Technical Data and Computer Software clause at 252.227-7013. Contractor/manufacturer is Frontline Systems, Inc., P.O. Box 4288, Incline Village, NV 89450.

*THANK YOU FOR YOUR INTEREST IN FRONTLINE SYSTEMS' PRODUCTS.*

---

# Contents

<b>Introduction</b>	<b>13</b>
Using the Premium Solver Platform.....	13
What's New in Version 7.0 .....	13
Solving Large Scale, Multi-Worksheet Models .....	14
Speeding Up Analysis, Solving and Reporting .....	14
Solving New Simulation Optimization Problems .....	15
Reporting Multiple Solutions from Optimization .....	15
Function-Based Models and Interactive Optimization.....	15
Object-Oriented API for Solver-Based Applications.....	16
Simplified Installation and Use.....	16
An Overview of Frontline's Solver Products .....	16
The Standard Excel Solver .....	16
The Premium Solver .....	17
The Premium Solver Platform .....	18
Risk Solver Engine .....	20
The Solver Platform SDK.....	21
Field-Installable Solver Engines .....	22
A Brief Tour of New Features .....	24
Model Analysis: The Polymorphic Spreadsheet Interpreter .....	24
Multistart Methods for Global Optimization .....	26
The Evolutionary Solver.....	27
The Interval Global Solver .....	28
The SOCP Barrier Solver .....	28
Second Order Cone Constraints.....	29
All Different Constraints .....	29
Simulation Optimization.....	30
New Types of Reports .....	30
User Interface Improvements.....	32
Speed Improvements.....	33
Programmability Improvements.....	34
How to Use This Guide.....	35
Using Online Help and the World Wide Web.....	36
Solver-Related Seminars and Books .....	37
Academic References for the Premium Solver .....	39
 <b>Installation and Licensing</b>	 <b>41</b>
What You Need .....	41
Working with Earlier Solver Versions .....	41
Installing the Software .....	42
Uninstalling the Software.....	45
Activating and Deactivating the Software.....	46
Excel 2007.....	46
Excel 2003 and Earlier .....	47
Setting Startup Options .....	47
Show Splash Screen.....	48

---

Load V7 VBA Macros.....	48
Licensing the Software.....	49
Installing Solver Engines .....	50

## **Solver Models and Optimization 53**

Introduction.....	53
Elements of Solver Models .....	53
Decision Variables and Parameters .....	53
The Objective Function .....	54
Constraints.....	54
Solutions: Feasible, “Good” and Optimal.....	55
More About Constraints.....	57
Functions of the Variables .....	59
Convex Functions .....	60
Linear Functions .....	61
Quadratic Functions.....	62
Nonlinear and Smooth Functions.....	63
Discontinuous and Non-Smooth Functions .....	64
Derivatives, Gradients, Jacobians, and Hessians .....	65
Optimization Problems and Solution Methods .....	67
Linear Programming.....	67
Quadratic Programming.....	68
Quadratically Constrained Programming.....	69
Second Order Cone Programming .....	69
Nonlinear Optimization .....	70
Global Optimization .....	71
Non-Smooth Optimization.....	73
Integer Programming .....	75
The Branch & Bound Method .....	75
Cut Generation.....	76
The Alldifferent Constraint.....	76
Simulation Optimization .....	77
Uncertain Variables .....	77
Uncertain Functions.....	78
Statistics for Uncertain Functions.....	78
Using Simulation Results in Optimization.....	78
Speed and Vectorized Evaluation.....	78

## **Building Solver Models 81**

Introduction.....	81
From Algebra to Spreadsheets .....	81
Setting Up a Model.....	81
A Sample Linear Programming Model .....	82
Decision Variables and Constraints .....	85
Variables and Multiple Selections .....	85
Using the Range Selector.....	87
Using the Variables Button.....	87
Constraint Left and Right Hand Sides .....	88
More Readable Models.....	91
Layout and Formatting.....	92
Using Defined Names .....	93
Models Defined Across Multiple Worksheets .....	94
Multiple Models and Multiple Worksheets .....	95

---

Defining Your Model with PSI Functions.....	96
Function-Based Style for Models .....	97
PSI Functions and Interactive Dialogs.....	98
Using the Insert Function Dialog.....	98
PSI Functions and Multiple Models .....	99

## **Analyzing and Solving Models 101**

Introduction.....	101
Using the Solver Model Dialog.....	101
Original Tab: Analyzing Model Structure.....	103
Using Model Statistics .....	104
Using the Check Model Button.....	105
Analyzing Model Convexity .....	105
Diagnosis Tab: Analyzing Model Exceptions .....	106
The Structure Report .....	106
Transformed Tab: Transforming a Non-Smooth Model .....	107
Effects of Model Transformation.....	108
Using Automatic Model Transformation .....	108
Options Tab: Selecting Model Features .....	110
Use Interactive Optimization .....	110
Use PSI Functions.....	110
Startup Options Group.....	111
Select Solver Engines Based on Model Type .....	111
Model Analysis When Solving.....	112
Using the Solve With Option.....	113
Using the Check For Options.....	113
Solve With Options and the Evolutionary Solver .....	115
Options Tab: Using Advanced Options .....	115
More on the Polymorphic Spreadsheet Interpreter.....	119
The Microsoft Excel Recalculator .....	119
The Polymorphic Spreadsheet Interpreter .....	121

## **Building Large-Scale Models 125**

Introduction.....	125
Designing Large Solver Models.....	125
Spreadsheet Modeling Hints .....	126
Optimization Modeling Hints .....	127
Using Multiple Worksheets and Data Sources.....	127
Quick Steps Towards Better Performance .....	128
Improving the Formulation of Your Model.....	129
Techniques Using Linear and Quadratic Functions .....	130
Techniques Using Linear Functions and Binary Integer Variables.....	131
Using Piecewise-Linear Functions.....	133
Organizing Your Model for Fast Solution .....	134
Fast Problem Setup .....	134
Using Array Formulas.....	136
Using the Add-in Functions .....	137

## **Simulation Optimization with Risk Solver Engine 143**

Monte Carlo Simulation and Optimization .....	143
Defining a Simulation Optimization Model .....	144
Using PSI Functions for Simulation .....	144



PSI Functions in Objectives and Constraints .....	145
Solving a Simulation Optimization Model .....	145
Solver Engines and Options .....	145
A Project Selection Example .....	145

## **Diagnosing Solver Results 149**

If You Aren't Getting the Solution You Expect .....	149
During the Solution Process .....	150
Choosing to Continue, Stop or Restart .....	150
When the Solver Finishes .....	151
Standard Solver Result Messages .....	151
Interval Global Solver Result Messages .....	162
Problems with Poorly Scaled Models .....	163
Dealing with Poor Scaling .....	163
Historical Note on Scaling and Linearity Tests .....	163
The Tolerance Option and Integer Constraints .....	164
Limitations on Smooth Nonlinear Optimization .....	164
GRG Solver Stopping Conditions .....	165
GRG Solver with Multistart Methods .....	166
GRG Solver and Integer Constraints .....	166
Limitations on Global Optimization .....	167
Rounding and Possible Loss of Solutions .....	167
Interval Global Solver Stopping Conditions .....	168
Interval Global Solver and Integer Constraints .....	169
Limitations on Non-Smooth Optimization .....	169
Effect on the GRG and Simplex Solvers .....	170
Evolutionary Solver Stopping Conditions .....	170

## **Solver Options 173**

Setting Options Programmatically .....	173
Object-Oriented API .....	173
The Standard Microsoft Excel Solver .....	174
Common Solver Options .....	175
Max Time and Iterations .....	175
Precision .....	176
Tolerance and Convergence .....	177
Assume Linear Model .....	177
Assume Non-Negative .....	178
Use Automatic Scaling .....	178
Show Iteration Results .....	178
Bypass Solver Reports .....	179
LP Simplex Solver Options .....	179
Pivot Tolerance .....	180
Reduced Cost Tolerance .....	180
LP/Quadratic Solver Options .....	180
Primal Tolerance and Dual Tolerance .....	181
Do Presolve .....	181
Derivatives for the Quadratic Solver .....	182
SOCP Barrier Solver Options .....	182
Gap Tolerance .....	183
Step Size Factor .....	183
Feasibility Tolerance .....	183
Search Direction .....	184

---

Power Index .....	184
GRG Nonlinear Solver Options .....	184
Convergence .....	185
Recognize Linear Variables .....	186
Derivatives and Other Nonlinear Options .....	186
Multistart Search Options .....	188
Multistart Search .....	188
Topographic Search .....	189
Require Bounds on Variables .....	189
Population Size .....	190
Random Seed .....	190
Interval Global Solver Options .....	190
Accuracy .....	191
Resolution .....	191
Max Time w/o Improvement .....	191
Absolute vs. Relative Stop .....	192
Assume Stationary .....	192
Method Options Group .....	192
Evolutionary Solver Options .....	194
Convergence .....	195
Population Size .....	195
Mutation Rate .....	195
Random Seed .....	196
Require Bounds on Variables .....	196
Local Search .....	196
Limits Tab Options .....	199
Max Subproblems .....	199
Max Feasible Solutions .....	200
Tolerance .....	200
Max Time without Improvement .....	200
Solve Without Integer Constraints .....	200
Integer Tab Options .....	201
Max Subproblems .....	201
Max Integer Solutions .....	202
Integer Tolerance .....	202
Integer Cutoff .....	203
Solve Without Integer Constraints .....	203
Use Dual Simplex for Subproblems .....	204
Preprocessing and Probing .....	204
Cut Generation .....	206
LP/Quadratic Solver Integer Tab Options .....	208
Max Subproblems .....	208
Max Feasible (Integer) Solutions .....	208
Integer Tolerance .....	209
Integer Cutoff .....	209
Maximum Cut Passes .....	210
Solve Without Integer Constraints .....	210
Use Strong Branching .....	211
Cuts & Heuristics .....	211
The Problem Tab .....	213
Loading, Saving and Merging Solver Models .....	214
Saved Model Formats .....	214
Using Multiple Solver Models .....	215
Transferring Models Between Spreadsheets .....	215

---

Merging Solver Models .....	216
-----------------------------	-----

## **Solver Reports 219**

Introduction.....	219
Selecting the Reports .....	221
The Scaling Report .....	224
An Example Model .....	226
The Answer Report .....	227
The Sensitivity Report .....	228
Interpreting Dual Values.....	229
Interpreting Range Information .....	230
The Limits Report .....	231
The Feasibility Report .....	231
The Linearity Report.....	232
The Population Report .....	234
The Solutions Report .....	235
Integer Programming Problems .....	235
Global Optimization Problems .....	235
Non-Smooth Optimization Problems.....	236
Solutions for Systems of Inequalities.....	237
Solutions for Systems of Equations .....	238

## **Using the Object-Oriented API 241**

Controlling the Solver's Operation .....	241
Why Use the Object-Oriented API? .....	241
Adding a Reference in the VBA Editor .....	243
Premium Solver Platform Object Model.....	243
Using the VBA Object Browser .....	244
Programming the Object Model.....	245
Example VBA Code Using the Object Model .....	245
Evaluators Called During the Solution Process .....	246
Refinery.xls: Multiple Blocks of Variables and Functions .....	247
CuttingStock.xls: Multiple Problems and Dynamically Generated Variables .....	247
Object-Oriented API Structure .....	251
Primary Objects .....	251
Secondary Objects.....	252
Primary Objects .....	253
Problem Object.....	253
Solver Object.....	255
Engine Object .....	256
Evaluator Object.....	257
Model Object.....	258
Variable Object.....	259
Function Object .....	261
Secondary Objects .....	263
ModelParam Object.....	264
EngineParam Object .....	265
EngineLimit Object .....	265
EngineStat Object.....	266
OptIIS Object .....	266
Statistics Object.....	267
DoubleMatrix Object.....	268
DependMatrix Object .....	269

---

## Using Traditional VBA Functions 271

Controlling the Solver's Operation .....	271
Running Predefined Solver Models .....	271
Using the Macro Recorder .....	271
Using Microsoft Excel Help .....	272
Referencing Functions in Visual Basic .....	272
Checking Function Return Values .....	272
Standard, Model and Premium Macro Functions .....	272
Standard VBA Functions .....	273
SolverAdd (Form 1) .....	273
SolverAdd (Form 2) .....	274
SolverChange (Form 1) .....	274
SolverChange (Form 2) .....	274
SolverDelete (Form 1) .....	275
SolverDelete (Form 2) .....	275
SolverFinish .....	276
SolverFinishDialog .....	277
SolverGet .....	277
SolverLoad .....	280
SolverOk .....	280
SolverOkDialog .....	281
SolverOptions .....	282
SolverReset .....	283
SolverSave .....	283
SolverSolve .....	284
Solver Model VBA Functions .....	286
SolverModel .....	287
SolverModelCheck .....	289
SolverModelGet .....	289
SolverDependents .....	291
SolverFormulas .....	291
Premium VBA Functions .....	291
SolverEVGet .....	291
SolverEVOptions .....	292
SolverGRGGet .....	294
SolverGRGOptions .....	295
SolverIGGet .....	296
SolverIGOptions .....	297
SolverIntGet .....	298
SolverIntOptions .....	300
SolverLimGet .....	302
SolverLimOptions .....	302
SolverLPGet .....	303
SolverLPOptions .....	304
SolverOkGet .....	305
SolverSizeGet .....	306

# Introduction

---

## Using the Premium Solver Platform

Thank you for using the Premium Solver or the Premium Solver Platform Version 7.0, Frontline Systems' newest and most powerful Solver products for Microsoft Excel. Both are fully compatible upgrades for the Solver bundled with Microsoft Excel, which was developed by Frontline Systems for Microsoft. This Guide covers all the features of the Premium Solver Platform and the five "Solver engines" bundled with it: the standard LP/Quadratic Solver, standard SOCP Barrier Solver, standard GRG Nonlinear Solver, standard Interval Global Solver, and standard Evolutionary Solver. The Premium Solver, Frontline's basic upgrade to the Excel Solver, includes a subset of the Premium Solver Platform features and three of the five Solver engines, as described later in this Introduction.

Before reading this User Guide, you may find it helpful to read the Solver-related topics in the online Help supplied with Microsoft Excel. These topics document the standard Solver's features and take you through the basic steps of using the Solver. We recommend that you try out the standard Solver on at least one problem of your own, or on one or more of the examples in the SOLVSAMP.XLS workbook which comes with Microsoft Excel.

This Guide goes well beyond the basics covered in the Microsoft Excel Help system. The Premium Solver Platform can solve far larger versions of the problems handled by the standard Excel Solver, and new kinds of problems using conic and global optimization, non-smooth functions, and new types of constraints. And the Premium Solver Platform can analyze and interpret your model in ways not possible with Microsoft Excel alone. This Guide will help you set up and solve much larger Solver problems, design your models for the fastest solutions, and understand the results of analyzing and solving your model – solutions, messages, and reports.

---

## What's New in Version 7.0

The Premium Solver Platform V7.0 is a major new release, designed to analyze and solve *much larger* models, *faster* than ever before, and – when used in conjunction with Frontline's new **Risk Solver Engine** – to help you find "best" solutions for problems involving uncertainty using *simulation optimization*. Version 7.0 is designed to work with Microsoft Excel 2000, Excel XP, Excel 2003, and Excel 2007; it takes full advantage of the new power of Excel 2007, the most extensive upgrade of Excel to be released in many years.

## **Solving Large Scale, Multi-Worksheet Models**

### ***Models Defined Across Multiple Worksheets***

When used with any modern version of Excel (Excel 2000, XP, 2003 or 2007), the Premium Solver Platform V7.0 supports Solver models spread across multiple worksheets in a workbook. It is no longer necessary to keep all of your decision variables and constraint left hand sides on the active worksheet. Yet you can still define a different Solver model (if desired) on each worksheet – and each of these models can include variables and constraints on any sheet in the workbook! You can still use the Load Model and Save Model buttons to create as many sets of model specifications as you like.

### ***Worksheets of 16K Columns and 1 Million Rows***

When used with Excel 2007, the Premium Solver Platform V7.0 supports worksheets with up to 16,384 columns and 1,048,576 rows – far beyond the limits of 256 columns and 65,536 rows in previous versions of Excel. This makes it much easier to lay out your models on a worksheet, without having to split up large tables of information. Many other limits, such as the maximum length of labels and formulas, are also greatly increased in Excel 2007.

### ***Reports with an Unlimited Number of Rows***

With previous versions of the Premium Solver Platform, you could easily define and solve models with more than 65,536 variables and/or constraints, but certain *reports* for those models were limited to the 65,536 rows on a single worksheet. In Excel 2007, this limit is increased to 1 million rows. But the Premium Solver Platform V7.0 will create reports of any size – even in Excel 2000, XP and 2003 – it will “wrap” the data across additional columns if the row limit is reached.

## **Speeding Up Analysis, Solving and Reporting**

Many of the built-in Solvers and field-installable Solver Engines in Version 7.0 feature improvements in the speed of solution. But the Premium Solver Platform V7.0 concentrates on speeding up “end-to-end solution time,” which includes setup time, report preparation and report generation time.

### ***Faster Model Analysis***

When you first click the Solve button with default options, or when you click the Check Model button in the Solver Model dialog, the Premium Solver Platform's Polymorphic Spreadsheet Interpreter (PSI) “parses” and analyzes your model. Parsing is faster for most models in Version 7.0, especially for models spread across multiple worksheets. Even if all of your decision variables and constraints are on one worksheet, if that sheet has many references to data on other sheets, you should see a significant speedup in Version 7.0.

### ***Faster Report Generation***

When an optimal solution is found, the Solver does some work to prepare for report generation, which can take some time for a large model, even if you don't select any reports in the Solver Results dialog (unless you've checked the box “Bypass Solver Reports”). And when you do select reports, more time is required to generate the report results. In Version 7.0, both the preparation for reports and the generation of selected reports are much faster, with speedups of 5 to 10 times for report generation.

## Solving New Simulation Optimization Problems

You can solve “simulation optimization” problems with the combination of the Premium Solver Platform V7.0 and **Risk Solver Engine**. By itself, Risk Solver Engine gives you lightning-fast, *Interactive Simulation* in Microsoft Excel – new simulation results each time you change the spreadsheet, up to 100 times faster than using Excel alone.

With Risk Solver Engine V7.0, your models can include random variables whose values are uncertain, defined by probability distributions. You can use the cells containing these random variables in any formula in your model. Risk Solver Engine will perform a Monte Carlo simulation with thousands of trials, where a different value is sampled for each random variable on each trial, and your model is recalculated with these values.

Statistics across all the trials are accumulated for any formula cell you designate. You can access these statistics in regular Excel formulas. And in the Premium Solver Platform V7.0, you can use these formulas in the objective and constraints of your optimization model. When you click Solve, the Premium Solver Platform performs a Monte Carlo simulation through Risk Solver Engine on each Trial Solution of the optimization. Doing this is usually an order of magnitude faster than in competitive products for “simulation optimization.”

## Reporting Multiple Solutions from Optimization

For global optimization, non-smooth optimization, and mixed-integer programming problems, the solution process used by most Solver engines finds several candidate solutions – for example, locally optimal solutions in searching for a globally optimal solution, or feasible integer solutions with good objective values (“incumbents”) for an integer programming problem. In the Premium Solver Platform V7.0, the full range of built-in and plug-in Solver engines support the new Solutions Report, which lists objective and decision variable values for each of these candidate solutions (the *best* solution is plugged in to the decision variable cells in your model, as usual). And using the new Object-Oriented API described below, you can easily access any of these alternative solutions in your VBA macro program code.

## Function-Based Models and Interactive Optimization

Also new in the Premium Solver Platform V7.0 are optimization models defined via functions on the worksheet, and *Interactive Optimization* that re-solves each time you change a number on the spreadsheet.

In Version 7.0, you can define your optimization model using the same interactive dialogs and VBA macros as the Excel Solver and previous versions of the Premium Solver Platform. But Version 7.0 also supports a new style of model definition, using **PSI functions** on the worksheet. These functions are compatible with the family of PSI functions used by Risk Solver Engine.

You can move easily between the interactive dialogs and PSI functions when creating your model. You can define variables and constraints by entering PSI function calls in worksheet cells, and you will find that these variables and constraints appear in the Solver Parameters dialog the next time you display it. You can also define variables and constraints in the Solver Parameters dialog, and cause PSI function calls to appear automatically in cells on the worksheet. PSI functions also offer a new alternative format for the Load Model and Save Model commands.

## ***Interactive Optimization***

The Premium Solver Platform V7.0 also supports *Interactive Optimization*: If you enable this feature, the Solver will re-optimize your model *each time you change a number* on the spreadsheet. For small to medium size models, this is not only a convenience – it can be a real decision aid: You will find that insights about your model, and decisions you can make, start to flow intuitively, when you can quickly see the impact of changing a parameter on the optimal solution.

## **Object-Oriented API for Solver-Based Applications**

The Premium Solver Platform V7.0 includes support for the “traditional” VBA functions used to programmatically control the Solver, such as SolverOK and SolverSolve. But it also provides a new, high level, **object-oriented API** (Application Programming Interface) for optimization that complements the Risk Solver Engine object-oriented API, and closely resembles the object-oriented API of the Solver Platform SDK V7.0, Frontline’s highly regarded Software Development Kit for creating optimization and simulation models in a programming language.

The new object-oriented API is more powerful and much more convenient for programming the Solver than the “traditional” VBA functions. For example, instead of writing VBA code using the Excel object model to retrieve decision variable and constraint values from cells on the worksheet, or obtain sensitivity information from cells on a report worksheet, you can simply reference an API object and property to retrieve each of these values in an array in your program. The object-oriented API can be used from VBA in Excel, or from VB.NET or C# using Visual Studio.

## **Simplified Installation and Use**

The Premium Solver Platform V7.0 is redesigned from the ground up internally. It uses a single program file **Solver32.xll** that provides the Solver Parameters, Solver Model and Solver Options dialogs, all reports, PSI functions, the object-oriented API, the PSI Interpreter and model analysis, and all five bundled Solver Engines. Solver32.xll is a COM add-in, an XLL add-in, and a COM server. In Version 7.0, the add-in file **Solver.xla** is optional – it is needed only if you wish to use the “traditional” VBA functions to program the Solver.

Thanks to this new architecture, the Premium Solver Platform V7.0 can be installed and used even if the standard Excel Solver was not included when you ran Setup for Microsoft Office and Excel. You can also install and use the Premium Solver Platform V7.0 *at the same time* as the standard Excel Solver, or an earlier version of the Premium Solver Platform or Premium Solver.

---

# **An Overview of Frontline’s Solver Products**

## **The Standard Excel Solver**

The standard Solver comes bundled with Microsoft Excel. It includes basic Solver engines for smooth nonlinear optimization problems using the GRG (Generalized Reduced Gradient) method, linear programming problems using the Simplex method, and integer programming problems using the Branch & Bound method. It is limited to problems of up to 200 decision variables and, for nonlinear problems, 100 constraints in addition to bounds on the variables. It provides three types of reports:



the Answer Report, Sensitivity Report, and Limits Report. The standard Solver can be controlled by user programs written in Visual Basic Application Edition, through a set of VBA functions such as SolverOK and SolverSolve.

## The Premium Solver

The Premium Solver is Frontline's basic upgrade to the Solver that comes with Microsoft Excel. It includes all of our latest speed and accuracy improvements to the standard Excel Solver, new diagnostic reports and ease-of-use features, and our new Evolutionary Solver, based on genetic algorithms, which complements the "classical" linear and nonlinear algorithms found in the standard Excel Solver.

Spreadsheet models you've developed for use with the standard Excel Solver can be used immediately with the Premium Solver, which will typically solve them faster and/or more accurately than before. And VBA programs you've written to control the standard Solver will work as-is with the Premium Solver.

Once the Premium Solver is installed, you can solve non-smooth optimization problems (NSPs, with the Evolutionary Solver) or smooth nonlinear optimization problems (NLPs, with the GRG Solver) of up to 400 variables and 200 constraints in addition to bounds on the variables, and linear programming problems (LPs, with the Simplex Solver) of up to 2,000 variables and 1,000 constraints – *10 times* the size of the Excel Solver. The Premium Solver also enhances the GRG Solver with multistart methods for global optimization, described below.

The Evolutionary Solver helps you find "good" solutions to Excel models made up of *any* standard or user-written (numeric) functions – even those with IF, CHOOSE, LOOKUP and similar functions that cause difficulty for the GRG Solver. Where the GRG Solver can find only a locally optimal solution, the Evolutionary Solver is much more likely to find a globally optimal (or near-optimal) solution.

User interface improvements include a more flexible way to specify decision variables (Changing Cells) that may be scattered across your worksheet; greater control over the solution of integer problems, and tolerances used by the linear Simplex and nonlinear GRG solution methods; more informative progress reporting on the Excel status bar; and faster ways to move among the Solver dialogs.

Improvements to the Solver reports include a new Population Report for the Evolutionary Solver, and new Linearity and Feasibility Reports for the GRG and Simplex Solvers, as well as the ability to create outlined reports, organizing the variables and constraints into the blocks you entered in the Solver Parameters dialog.

Solution speed in the Premium Solver is improved for all types of problems, especially for problems with all linear and integer constraints. The Premium Solver also supports *fast problem setup*, a subset of the problem analysis capabilities of the Premium Solver Platform, for linear and integer programming models built with the SUM, SUMPRODUCT and MMULT functions and the new DOTPRODUCT function. Problem setup can be *5 to 50 times faster* for models in this form. If you expect that your models will grow in size (and most users find that they do), we recommend that you design your models using these functions – or use the Premium Solver Platform, which offers much greater flexibility in problem setup and analysis.

The Premium Solver also supports a new type of constraint for integer variables, called the *alldifferent* constraint. The alldifferent constraint specifies that at the solution, each integer variable in the group must have a value that is different from all the others. Hence, the variables in the group form an ordering, or permutation, of integers. The alldifferent constraint can be used to easily model problems involving orderings or assortments, such as the well-known Traveling Salesman Problem.

In **Version 7.0**, the Premium Solver supports worksheets with up to 16,384 columns and 1,048,576 rows in Excel 2007, simulation optimization with Risk Solver Engine, function-based models and Interactive Optimization, and the new object-oriented API. But the Premium Solver requires that all decision variables and constraint left hand sides be on the active worksheet; only the Premium Solver Platform allows you to create models spread across multiple worksheets in a workbook, with variables and constraint left hand sides on any worksheet in the workbook.

### ***The Premium Solver for Education***

The Premium Solver for Education is a special version of the Premium Solver that is designed exclusively for university use, typically in undergraduate or graduate (MS or MBA) courses in engineering or business administration. It includes the functionality and user interface improvements of the Premium Solver, such as the Evolutionary Solver, the Linearity, Feasibility and Population Reports, and report outlining, but none of the speed or capacity improvements. It is limited to 200 decision variables and (for nonlinear and non-smooth problems) 100 constraints in addition to bounds on the variables, and its speed is about the same as the standard Excel Solver. It is licensed only through academic textbook publishers who are serving the university market, and it is now included in a wide range of courses and textbooks that introduce students to operations research, management science, and the concepts of optimization.

### **The Premium Solver Platform**

The Premium Solver Platform is Frontline's "flagship" product for Microsoft Excel, and the base product for optimizing all types of large-scale models. It includes all of the features of the Premium Solver, improvements to each of three bundled Solver engines – the Evolutionary Solver, nonlinear GRG Solver, and especially the linear Simplex Solver – and two additional Solver engines: the Interval Global Solver, which solves global optimization problems, and the SOCP Barrier Solver, which solves linear, quadratic, quadratically constrained, and new second order cone programming (SOCP) problems. And it solves models with decision variables and constraints on any worksheet in a workbook.

The Premium Solver Platform has two other fundamental, powerful capabilities beyond the Premium Solver: It includes a new *Polymorphic Spreadsheet Interpreter* (PSI technology) capable of analyzing Microsoft Excel formulas and models, where the Premium Solver relies on Microsoft Excel for this purpose. And it supports *multiple, field-installable Solver engines* that leverage the PSI technology to solve much larger and more challenging optimization problems. Customers have used this capability to solve problems with as many 2.4 million decision variables in Excel.

### ***Model Analysis: The Polymorphic Spreadsheet Interpreter***

The Premium Solver Platform includes a new Polymorphic Spreadsheet Interpreter for Excel formulas, developed by Frontline Systems. The Interpreter does much more than simply recalculate values for Excel formulas – it can interpret the formulas in many other ways that are advantageous for the Solver, as further described in the next section, "A Brief Tour of New Features." For example:

The Interpreter can *diagnose* your model as a linear programming, quadratic or conic, smooth nonlinear, or non-smooth optimization model, and automatically select Solver engines suitable for your model. It can even pinpoint formulas that make your model nonlinear or non-smooth.

The Interpreter can compute derivatives (the *gradients* of problem functions) directly, using the methods of *automatic differentiation*. This is much faster and more accurate than the method of formula recalculation and “finite differencing” used by the Excel Solver and Premium Solver, and it greatly improves performance of all of the field-installable Solver engines.

The Interpreter can efficiently compute second derivatives (the *Hessians* of problem functions) that are required by the new SOCP Barrier Solver and MOSEK Solver, and used by the KNITRO Solver for best performance.

The Interpreter can evaluate Excel formulas in your model over *intervals*, in addition to single numeric values. This capability is used by the new Interval Global Solver to find globally optimal solutions to optimization problems.

## **Solver Engines in the Premium Solver Platform**

The **LP/Quadratic Solver**, which replaces the Excel Solver's linear Simplex method in the Premium Solver Platform, handles both linear programming (LP) and quadratic programming (QP) problems with up to 8,000 decision variables and constraints – four times the size of the Premium Solver, and *40 times* the size of the Excel Solver. It uses state-of-the-art large-scale, sparse primal and dual Simplex, presolve, and quadratic optimization methods, for world-class LP/QP performance. Quadratic programming problems have a quadratic objective function and all linear constraints; they are often used to find “efficient portfolios” of securities using the Markowitz or Sharpe methods.

For problems involving integer variables, the LP/Quadratic Solver employs state-of-the-art Branch, Cut and Bound methods that include sophisticated branch node and variable selection, preprocessing and probing, cut generation with 12 types of cuts, and two types of heuristics. These strategies often drastically speed up the solution of integer problems, often by factors of 1000 or more over the standard Excel Solver. Tests on user models suggest that the LP/Quadratic Solver in the Premium Solver Platform Version 7.0 is five to ten times faster than in Version 6.0.

The **nonlinear GRG Solver** in the Premium Solver Platform is augmented with “multistart” or “clustering” methods for global optimization. For some smooth nonlinear problems, multistart 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 – and of course, they are far easier to use than a manual exploratory process. The GRG Solver handles smooth nonlinear (NLP) problems of up to 500 decision variables and 250 constraints, plus bounds on the variables.

The **Evolutionary Solver** in the Premium Solver Platform is actually a hybrid of genetic and evolutionary algorithms and classical optimization methods. This new hybrid Evolutionary Solver can be applied to extremely challenging problems, with both non-smooth or discontinuous functions, and hundreds of conventional constraints. It handles non-smooth (NSP) problems of up to 500 decision variables and 250 constraints, plus bounds on the variables.

The **Interval Global Solver** in the Premium Solver Platform uses state-of-the-art interval methods to find the globally optimal solution to a nonlinear optimization problem, all real solutions to a system of nonlinear equations, or an “inner solution” to a system of nonlinear inequalities, as further described in the next section, “A Brief Tour of New Features.” It handles smooth nonlinear (NLP) problems of up to 500 decision variables and 250 constraints, plus bounds on the variables.

The **SOCP Barrier Solver** in the Premium Solver Platform uses an interior point method to solve linear (LP), quadratic (QP), quadratically constrained (QCP), and

second order cone programming (SOCP) problems with up to 2,000 variables and 8,000 constraints. Many problems in quantitative finance are easier to formulate with quadratic constraints; unlike most quadratic solvers that handle only a quadratic objective, the SOCP Barrier Solver handles both quadratic objectives and quadratic constraints. Most significant, the SOCP Barrier Solver is designed for *second order cone programming (SOCP)* – the natural generalization of linear and quadratic programming. Many problems in quantitative finance and engineering design are best formulated as SOCP problems.

Finally, when used with Risk Solver Engine V7.0, the Premium Solver Platform solves *simulation optimization* problems, where the objective and constraints depend on both decision variables and random variables, defined by probability distributions – at speeds far beyond that offered by competitive software products for Excel.

## Risk Solver Engine

Frontline's Risk Solver Engine implements a new approach to Monte Carlo simulation that lets you play „what if with uncertain values as easily as you do with ordinary numbers. Each time you change a number on the spreadsheet, a simulation with thousands of trials is performed – often in no more time than a single spreadsheet recalculation – and a full range of simulation results and statistics may be displayed *on the spreadsheet*. By writing macros in Excel VBA, you can fully control Risk Solver Engine and create custom risk analysis applications.

### ***The Magic of Interactive Simulation***

With Risk Solver Engine, risk analysis for uncertain models becomes **as easy as asking „what if**. Suddenly, you find that insights about the model, and decisions you can make, start to flow intuitively. In the words of Dr. Sam Savage, a noted authority on Monte Carlo simulation, “Risk Solver Engine does for uncertainty what the spreadsheet did for numbers.”

### ***Powered by PSI Technology***

Risk Solver Engine uses Frontline's *Polymorphic Spreadsheet Interpreter* technology to achieve breakthrough simulation speeds – up to **100 times faster** than normal Excel-based Monte Carlo simulation – thus making Interactive Simulation practical every time you change a number on the spreadsheet

### ***Support for Probability Management Concepts***

Risk Solver Engine supports the practice of *Probability Management* and *Coherent Modeling* in large organizations. In addition to *Interactive Simulation* on every spreadsheet recalculation, Risk Solver Engine directly supports *Stochastic Libraries*, both on the Excel spreadsheet and in Excel VBA. *Certification Authorities* such as the “Chief Probability Officer” proposed by Dr. Sam Savage can approve such libraries for use in Risk Solver Engine.

*Certified Distributions*, created by experts – perhaps like you – and provided to end users in the form of numeric tables called **SIPs** (Stochastic Information Packets) and **SLURPs** (Stochastic Library Units, Relationships Preserved), can make simulation models far easier to create, analyze and compare.

### ***Excel Functions for Monte Carlo Models***

Risk Solver Engine allows you to define certain cells as *random variables*, whose values are drawn from your choice of more than 40 PSI Distribution functions – from

PsiBernoulli() to PsiWeibull(). You can also draw values from predefined *Certified Distributions*, by simply referring to these distributions in PsiSip() and PsiSlurp() functions. You can easily shift, truncate and correlate probability distributions with PSI Property functions such as PsiShift(), PsiTruncate() and PsiCorrMatrix().

A special feature of Risk Solver Engine is its ability to display simulation trials and summary statistics, *instantly* each time the spreadsheet changes. You simply create formulas that compute functions of your random variables, then refer to these computed results in other formulas with PsiMean(), PsiVariance(), PsiPercentile(), PsiCVaR(), and similar functions. PsiFrequency() instantly gives you a frequency distribution across simulation trials, that can be analyzed or charted in Excel.

## **VBA Objects/Properties for Monte Carlo Models**

You can easily write VBA (Visual Basic Application Edition) macros in Excel that control Risk Solver Engine. Define a Problem and instantiate it from the spreadsheet with two lines of code, then access the uncertain elements of your model via Variable and Function objects. Perform simulations, access trials and summary statistics, and present them the way you want to your end user.

Risk Solver Engine's VBA object model closely resembles the new object-oriented API in the Premium Solver Platform V7.0, and the object-oriented API of Frontline's Solver Platform SDK – which includes a compatible toolkit for Monte Carlo simulation. This makes it easier to move a simulation application from Excel to a custom program written in C/C++, Visual Basic, VB.NET, Java or MATLAB.

## **The Solver Platform SDK**

The Solver Platform SDK is Frontline's “flagship” product for software developers building optimization and simulation applications in a programming language, and the base product for solving all types of large-scale models in this form. The Solver Platform SDK is the successor to Frontline's Solver DLL Platform (offering 100% upward compatibility for existing users) that goes far beyond other “callable library” optimization products. It offers:

- A high-level, object-oriented application programming interface (API), for languages like C++, C#, Java, Visual Basic and VB.NET, that allows you to implement your model in terms of objects such as a Problem, Model, Solver, Variable, and Function.

- A conventional procedural API that provides access to all the power of the Solver Platform SDK from non-object-oriented languages like C and Fortran.

- Full support for Java and Matlab – Matlab users have access to both the object-oriented API and the procedural API in the Solver Platform SDK.

- Deep support for Microsoft .NET and Visual Studio .NET 2003/2005, with “wizards” that help you create a working application in less than a minute, and full support for “IntelliSense” and “Balloon Help” that show you available API options as you type, or even as your mouse hovers over a line of program code.

- Deep support for Microsoft COM, Visual Basic 6, and Visual C++ 6, including wizards for these language systems.

- Bundled LP/Quadratic, SOCP Barrier, GRG Nonlinear, and Evolutionary Solvers, plus a new Solver for Monte Carlo simulation problems, and support for seven field-installable large-scale Solver Engines.

Flexible licensing policies and software that enable you to easily *develop* and *deploy* your application, for desktop, Intranet/Web server, or Web service use, using the Solver Platform SDK and field-installable Solver Engines.

To learn more about the Solver Platform SDK, please visit [www.solver.com](http://www.solver.com) or contact Frontline Systems at (775) 831-0300 or [info@solver.com](mailto:info@solver.com).

## Field-Installable Solver Engines

The Premium Solver Platform and the Solver Platform SDK both support multiple, field-installable Solver engines, in addition to their “bundled” Solver engines. Such Solver engines are licensed as separate products, and they provide additional power and capacity to solve problems much larger and/or more difficult than the problems handled by the bundled Solver engines. Unlike most other optimization software, a license for one of Frontline’s Solver engines enables you to use that Solver in Excel, Matlab, Java, C/C++, C#, Visual Basic, VB.NET, and other languages, using either or both the Premium Solver Platform or the Solver Platform SDK.

Field-installable Solver engines are seamlessly integrated into the Premium Solver Platform – to use one, you simply select the Solver engine by name in the dropdown list that appears in the Solver Parameters dialog. They produce reports as Excel worksheets, like the bundled Solver engines; they recognize common Solver options and provide their own Options dialogs; and they can be controlled by VBA code in your custom applications. The Solver engines are also seamlessly integrated into the Solver Platform SDK – to use one, you simply add it to the collection of available Solver Engines, and select it with a single line of code when you want to solve a problem. 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.

### ***The Large-Scale LP/QP Solver***

Frontline’s Large-Scale LP/QP Solver is designed to solve linear and quadratic programming problems much larger than the 8,000 variable limit imposed by the built-in LP/Quadratic Solver. It uses scaled-up, state-of-the-art primal and dual Simplex, presolve, quadratic, and Branch, Cut and Bound methods to solve challenging, large scale LP/QP models. It is offered in two versions: A Standard version, handling problems of up to 32,000 variables and 32,000 constraints; and an Extended version, with **no limits** on problem size except time and memory, that customers have used to solve LP problems with millions of decision variables.

### ***The Large-Scale GRG Solver***

Frontline’s Large-Scale GRG Solver is designed to solve smooth nonlinear problems much larger than the 500 variable limit imposed by the built-in nonlinear GRG Solver. It uses sparse matrix storage methods, advanced methods for selecting a basis and dealing with degeneracy, methods for finding a feasible solution quickly, and other algorithmic methods adapted for larger problems. It is offered in two versions, one capable of solving problems of up to 4,000 variables and 4,000 constraints, the other capable of handling large problems of up to 12,000 variables and 12,000 constraints.

### ***The Large-Scale SQP Solver***

Frontline’s Large-Scale SQP Solver is our *most versatile* large-scale Solver engine, since it handles *every type* of optimization problem. It is capable of solving very large linear programming, quadratic programming, and smooth nonlinear

optimization problems, with **no limits** on problem size except time and memory. In **Version 7.0** its algorithmic methods are greatly enhanced, and it integrates a version of Frontline's Evolutionary Solver that uses the SQP method for local searches, making it possible to solve large non-smooth optimization problems. It is especially effective on nonlinear or non-smooth problems with many linear constraints or linear occurrences of variables, since it exploits information about the model supplied by the Interpreter in the Premium Solver Platform. The Large-Scale SQP Solver uses a Sequential Quadratic Programming method.

### ***The KNITRO Solver***

Frontline's KNITRO Solver – developed in close cooperation with Ziena Optimization – offers breakthrough performance in solving large scale smooth nonlinear optimization problems, with **no limits** on problem size except time and memory. The KNITRO Solver is the leading implementation of new state-of-the-art interior point methods for non-convex problems, the result of intense research in large-scale nonlinear optimization in recent years. In **Version 7.0**, the KNITRO Solver also includes active set (SLQP) methods, enabling it to perform exceptionally well on both tightly and loosely constrained large scale nonlinear optimization problems.

### ***The MOSEK Solver***

Frontline's MOSEK Solver – developed in close cooperation with MOSEK ApS – is a large-scale “upgrade” for the new SOCP Barrier Solver that solves LP, QP, QCP, and SOCP problems of virtually unlimited size. The MOSEK code has been used to solve SOCP problems of over 100,000 variables, and much larger linear programming (LP) problems, where it is competitive with the very best Solver engines. The Extended version of the MOSEK Solver Engine also solves large scale convex smooth nonlinear optimization problems. It has **no limits** on problem size except time and memory.

### ***The XPRESS Solver***

Frontline's XPRESS Solver Engine – developed in close cooperation with Dash Optimization – brings the lightning-fast performance, and virtually unlimited problem solving capacity of the Xpress<sup>MP</sup> mixed-integer linear optimizer to Excel spreadsheet users. Many professionals regard Xpress<sup>MP</sup> as the world's best general-purpose mixed-integer optimizer. The Extended version of the XPRESS Solver Engine also solves quadratic programming (QP) problems. Version 7.0 of this Solver Engine offers twenty to fifty times the already-impressive performance of its inaugural Version 4.0. The XPRESS Solver Engine offers more than 60 Solver Options. It has **no limits** on problem size except time and memory.

### ***The OptQuest Solver***

Frontline's OptQuest Solver – developed in close cooperation with OptTek Systems, Inc. – is designed to work with all types of Excel models. Your Excel model can contain any standard or user-written (numeric) functions – even discontinuous functions such as IF, CHOOSE, LOOKUP, and COUNT and non-smooth functions such as ABS, MIN, and ROUND that cause difficulty for the classical nonlinear Solvers. Also, the OptQuest Solver may find a globally optimal (or near-optimal) solution to a problem with multiple locally optimal solutions (though it cannot give any assurance of finding the globally optimal solution). The OptQuest Solver uses advanced methods including tabu search and scatter search, as described in the Solver Engine User Guide. It supports up to 5,000 variables and 1,000 constraints.

To learn more about field-installable Solver Engines, please visit [www.solver.com](http://www.solver.com) or contact Frontline Systems at (775) 831-0300 or [info@solver.com](mailto:info@solver.com).

### **Other New Solver Engines**

Frontline is constantly working to enhance both the five bundled Solver engines in the Premium Solver Platform, and the seven field-installable Solver engines briefly described above. And Frontline is constantly working on new Solver engines, to solve new types and sizes of optimization models, including models that call for good or optimal decisions in the presence of uncertainty.

Be sure to stay in contact with Frontline Systems, via phone, email, or the World Wide Web (at [www.solver.com](http://www.solver.com)) to get the latest news about the availability of new field-installable Solver engines, and other upgrades for the Premium Solver Platform.

---

## **A Brief Tour of New Features**

The Premium Solver Platform provides a wide range of new features for Solver users, including the ability to solve entirely new kinds of problems – such as new **convex and conic optimization** problems in Version 6, and new **simulation optimization** problems in Version 7. This section provides a tour of these new features, with brief explanations of what they mean for your ability to create models and find optimal solutions.

### **Model Analysis: The Polymorphic Spreadsheet Interpreter**

The Excel Solver and Premium Solver rely on Microsoft Excel itself to read and analyze (“parse”) the formulas you enter in spreadsheet cells, and calculate values for (“interpret”) these formulas whenever the Solver changes values of input cells.

The Premium Solver Platform retains the ability to use Microsoft Excel for evaluation of spreadsheet formulas, but it also includes a new Polymorphic Spreadsheet Interpreter (*PSI* technology) for Excel formulas, developed by Frontline Systems. The term “polymorphic” has much the same meaning as it does in programming languages such as C++ and Java, but for Microsoft Excel formulas. The Interpreter does much more than simply recalculate values for Excel formulas – it can interpret the formulas in many other ways that are advantageous for the Solver.

The Interpreter can handle nearly any Microsoft Excel formula syntax, including array formulas, and almost all of the Excel built-in functions, including the financial, statistical, and engineering functions in the Analysis Toolkit. For rarely used syntax forms, a few functions such as INDEX and OFFSET, and some user-defined functions written in VBA or other languages, the Premium Solver Platform can still use Microsoft Excel instead of its own Interpreter for Excel formulas.

### **Automatic Model Diagnosis**

The Polymorphic Spreadsheet Interpreter can *diagnose* your model by evaluating your Excel formulas with an „overloaded” type for each cell, operator and function. It determines which input cells are decision variables, and which are constant in the model; then it evaluates each arithmetic operation or function in your model to determine how the formula depends on each decision variable: Whether it is independent (i.e. constant), linear, quadratic, smooth nonlinear, or non-smooth as a function of that variable.



This enables the Interpreter to diagnose your model as a linear programming, quadratic programming, conic programming, smooth nonlinear, or non-smooth optimization model. Further, you can tell the Interpreter that your goal or intent was to create (say) a *linear* model, and the Interpreter will pinpoint the specific cells containing formulas that create a *nonlinear* relationship in your model. Similarly, if you intended to create a smooth nonlinear model, the Interpreter will pinpoint cells containing non-smooth operations or functions of each decision variable.

The Interpreter can also help diagnose problems of poor scaling in your model: It can evaluate your formulas while keeping track of the magnitude of each intermediate result, and pinpoint the formulas that are likely to yield a loss of accuracy due to poor scaling, that cannot be handled via automatic rescaling in the Solver engines.

### ***Automatic Tests for Convexity***

Once your model goes beyond linear programming to include quadratic or nonlinear functions, it may remain easy or it may become very difficult to solve. If your model is **convex**, it can be solved quickly and reliably to a globally optimal solution, even if it grows very large. But if your model is **non-convex**, you'll find that Solvers (of all types) can find only a locally optimal solution, and may even have trouble finding a feasible solution – and the time taken to find a solution may be so long that it limits the size of model you can solve. But most users have found it difficult or impossible to *determine whether* their nonlinear model is convex.

The Premium Solver Platform **Version 7.0** includes a ***unique, automatic test for convexity*** of your model and its objective and constraints in Excel, based on pioneering work by Frontline Systems developers. The convexity test is not always conclusive, because a conclusive test would take time exponential in the number of variables. But in many cases, with the Premium Solver Platform you can determine whether your model is convex, and identify specific functions that make your model non-convex, by pressing a button.

### ***Automatic Transformation of Non-Smooth Models***

As described in the chapter “Solver Models and Optimization,” using even one non-smooth function (such as IF, MIN, MAX, ABS, AND, OR, or NOT, with arguments that depend on the decision variables) in a model that is otherwise linear (using functions such as SUM and SUMPRODUCT) changes the model from a *linear programming* (LP) problem to a *non-smooth optimization* (NSP) problem.

Thanks to the Evolutionary Solver in the Premium Solver Platform, you can still solve such models. But the consequences of such non-smooth functions for the Solver are considerable: Where an LP can be solved very quickly and reliably up to very large size, and the solution is basically guaranteed to be optimal, an NSP takes far more time to solve, requires inherently less reliable methods, and there are no guarantees as to whether the solution is truly optimal.

In **Version 7.0**, the Premium Solver Platform can *automatically transform* your model, replacing IF, MIN, MAX, ABS, AND, OR, and NOT functions and <= and >= operators with additional variables and linear constraints that achieve the same effect, for optimization purposes, as these functions. If all non-smooth functions in your model can be transformed, the result will be a linear mixed-integer (LP/MIP) model that can be solved by a variety of Solver engines, from the standard LP/Quadratic Solver to the XPRESS Solver – giving you a better chance of finding an optimal solution with certainty, in a reasonable amount of time.

## Automatic Differentiation

Most Solvers make heavy use of derivatives or gradients of the problem functions (the objective and constraints) with respect to the decision variables. Linear programming algorithms require that derivatives be evaluated once, to obtain the LP coefficient matrix. Nonlinear optimization algorithms typically require that derivatives be evaluated many times, once at each major iteration or trial point. Hence, derivative evaluation is key to both the speed and accuracy of such optimization algorithms.

The Excel Solver and Premium Solver estimate derivative values by the method of *finite differencing*: They use Microsoft Excel itself to recalculate values for the objective and constraints at the “current point” (i.e. values of the decision variables), and at nearby points with small changes (“perturbations”) in each decision variable. This process, while often adequate, is relatively slow and inaccurate – it takes many recalculations and may lose significant digits as it performs many division operations.

The Premium Solver Platform's Polymorphic Spreadsheet Interpreter can compute derivatives directly, by evaluating your Excel formulas with an overloaded „gradient type for each cell, operator and function, using the methods of *automatic differentiation*. Analytic formulas for the derivatives are applied to each arithmetic operator and elementary function, and the chain rule is applied to compute derivatives for composite functions. Hence, derivative values can be obtained many times faster, and without any loss of accuracy beyond the actual function values themselves.

The Interpreter goes further to support the new SOCP Barrier Solver, MOSEK Solver, and KNITRO Solver in **Version 7.0**: It computes the *Hessian* (matrix of second order derivatives) of each problem function using the methods of automatic differentiation. The method of finite differencing is far too slow and inaccurate for this purpose – the Polymorphic Spreadsheet Interpreter makes such new methods and Solver engines practical in Microsoft Excel.

## Interval Arithmetic

To support the Platform's Interval Global Solver, the Polymorphic Spreadsheet Interpreter can also evaluate Excel formulas with an overloaded „interval type for each cell, operator and function. An *interval* such as [1, 2] represents all of the possible numeric values between 1 and 2. Addition, subtraction, and other arithmetic operations and functions can be defined over intervals – for example,  $[1, 2] + [3, 4] = [4, 6]$  in interval arithmetic. The Interpreter can compute interval values for all of Excel's arithmetic operators and most built-in smooth functions. It provides even more powerful facilities, including *automatic differentiation over intervals*, to Solver engines that use interval methods. See below for a brief description of the Interval Global Solver included in the Premium Solver Platform.

## Multistart Methods for Global Optimization

As explained in the chapter “Solver Models and Optimization,” the nonlinear GRG Solver – like virtually all “classical” nonlinear optimizers – will find only a *locally optimal* solution to a **non-convex** problem. Imagine a graph of the objective function with “hills” and “valleys.” The GRG Solver will typically find the peak of a hill near the starting point you specified (if maximizing), but it may not find an even higher peak on another hill that is far from your starting point. In some problems this is sufficient, but in other cases you may want to find a *globally optimal* solution.

With multistart methods in the Premium Solver and Premium Solver Platform, the nonlinear GRG Solver can be automatically run many times from judiciously chosen

starting points, and the best solution found (the “highest peak” if maximizing) will be returned as the optimal solution. An algorithm called “multi-level single linkage” randomly samples starting points, collects them into “clusters” that are likely to lead to the same locally optimal solution, and runs the GRG Solver from a representative point in each cluster. This process continues until a Bayesian statistical test estimates that all locally optimal solutions have likely been found. Then the best of these solutions is returned as the probable globally optimal solution.

## The Evolutionary Solver

The standard Excel Solver is able to find solutions for “classical” optimization models, including linear programming and integer programming problems, and problems with smooth nonlinear objectives and constraints. But the Excel formula language includes many operations and functions, such as IF, CHOOSE and LOOKUP, that don’t satisfy the requirements for linear or smooth nonlinear problems. The standard Excel Solver cannot handle models that employ such functions in the calculation of their objectives or constraints. But the Evolutionary Solver included in the Premium Solver and Premium Solver Platform can handle models whose calculation uses *any* type of (numeric) operations and functions.

The Evolutionary Solver provides an alternative to multistart methods to seek a globally optimal solution to a **non-convex** problem, even if all the problem functions are smooth. Rather than search in the neighborhood of a single starting point, it maintains a population of candidate solutions “scattered around the landscape,” and (based in part on random choices) it will attempt to improve each one.

The Evolutionary Solver appears in the dropdown list of Solver engines in the Solver Parameters dialog. To use it, you simply select the Evolutionary Solver from the list. That’s it! You don’t have to change your model, or your selections of variables, constraints or objective. You can switch Solver engines at any time. You can even apply the Evolutionary Solver to models created with the standard Excel Solver, with no extra work on your part to set up the optimization problem.

The Evolutionary Solver is based on the principles of “genetic algorithms” and “evolutionary algorithms.” In tests on a wide variety of Excel models, it outperforms competitive products whose main (or only) feature is a genetic algorithm, by finding better solutions in significantly less time. And the Evolutionary Solver doesn’t require that you learn new terminology or choose from a variety of complicated “solving methods.” You just select the Evolutionary Solver engine and click Solve.

A Solver based on genetic or evolutionary algorithms is not a panacea, however. Unlike the Simplex and GRG Solvers which are *deterministic* optimization methods, the Evolutionary Solver is a *nondeterministic* method: Because it is based partly on random choices of trial solutions, by default it will often find a different “best solution” each time you run it, even if you haven’t changed the model at all. And unlike the Simplex and GRG Solvers, the Evolutionary Solver has no way of knowing for certain that a given solution is optimal – even “locally optimal.” Similarly, the Evolutionary Solver has no way of knowing for certain whether it should stop, or continue searching for a better solution. With the Premium Solver and Premium Solver Platform, however, you are not limited to a genetic algorithm – you can apply the most appropriate Solver engine to each problem you encounter.

## Hybrid Evolutionary Solver Methods

The Evolutionary Solver in the Premium Solver Platform is actually a hybrid of genetic and evolutionary algorithms and classical optimization methods, including gradient-free direct search methods, classical gradient-based quasi-Newton

methods, and even the Simplex method for linear subsets of the constraints. The classical methods sometimes yield rapid “local improvement” of a trial solution, and they also help to “solve for” sets of constraints. The Evolutionary Solver also includes new “filtered local search” methods that greatly improve performance on smooth global optimization problems; and new “integer heuristic” methods from the local search literature that improve performance on problems with integer variables.

Working with the Polymorphic Spreadsheet Interpreter, the Evolutionary Solver can automatically apply genetic algorithm methods to the *non-smooth* parts of a problem, and apply classical methods to the *smooth nonlinear* and *linear* parts of the problem. The Evolutionary Solver is often able to solve problems with hundreds of constraints, which are typically beyond the capabilities of genetic and evolutionary algorithms working alone.

## The Interval Global Solver

The Interval Global Solver is a new kind of Solver engine built-in to the Premium Solver Platform. It uses *interval methods* to find the globally optimal solution to a nonlinear optimization problem, all real solutions to a system of nonlinear equations, or an “inner solution” to a system of nonlinear inequalities.

The Interval Global Solver uses *deterministic* methods to search for the global optimum, whereas the Evolutionary Solver and the multistart methods described above use *nondeterministic* methods, which involve an element of random chance. Given time, the Interval Global Solver will find a *proven* global optimum, and it will find *all* real solutions to a system of nonlinear equations (subject to a few caveats when using its advanced methods, as described in the chapter “Diagnosing Solver Results”). In contrast, the Evolutionary Solver and the multistart methods described above often find good solutions, but cannot find “provably optimal” solutions.

Interval methods for global optimization are the subject of considerable research currently. Frontline believes its Interval Global Solver is the first commercially supported optimizer to use these methods. We’ve pioneered several of the methods used in this Solver, such as linear enclosures, which have yielded order-of-magnitude speed improvements compared to earlier interval algorithms.

Now, you’ll have an easy-to-use way to apply these groundbreaking methods to your own problems. You don’t have to do anything special to use these methods – just define your model in the usual way, select “Interval Global Solver” from the dropdown list of Solver engines, and click the Solve button. *Note:* The Interval Global Solver is not available in the Solver Platform SDK, because it would require that you write code in some language to calculate your model using interval methods.

## The SOCP Barrier Solver

The Premium Solver Platform **Version 7.0** includes a new, fifth built-in Solver engine, the Standard SOCP Barrier Solver. This Solver finds optimal solutions for second-order cone programming (SOCP) problems, which are a superset of linear programming (LP), quadratic programming (QP), and quadratically constrained programming (QCP) problems, with up to 2,000 decision variables. It supports the new *second order cone* (SOC) constraints in the Premium Solver Platform (see below). To use the SOCP Barrier Solver, you simply select it from the dropdown list of Solver engines, and click Solve – no changes to your model are necessary.

The term “Barrier” comes from the optimization algorithms used by this Solver engine. Where the LP/Quadratic Solver uses the Simplex method, augmented for

quadratic objectives, the SOCP Barrier Solver uses a Barrier method, also called an Interior Point method. Where the Simplex method's trial solutions are always at „corners“ on the surface of the feasible region, the Barrier method's trial solutions are always in the *interior* of the feasible region, until the final steps where the optimal solution is reached. Where the number of Simplex iterations typically grows with the number of constraints, the number of Barrier iterations is *independent* of the number of constraints, and is usually between 10 and 50 (but the time taken per iteration grows with problem size).

## Second Order Cone Constraints

The Premium Solver Platform **Version 7.0** supports a new type of constraint, called a second-order cone (SOC) constraint. An SOC constraint is created like any other constraint, by clicking the Add button to display the Add Constraint dialog, selecting a range of decision variable cells for the Cell Reference or left hand side, and selecting **soc** or **src** (rotated second order cone) from the Relation dropdown list. This specifies that the vector formed by the  $n$  decision variables must lie in the second order cone (also called the Lorentz cone or “ice cream” cone) of dimension  $n$ .

A linear programming problem plus one or more SOC constraints defines a **second order cone programming (SOCP)** problem. All “ordinary” non-negative decision variables also belong to a cone, called the *non-negative orthant* – hence a linear programming (LP) problem is a special case of a conic programming problem, where the only cone constraint is non-negativity. Second order cone programming is the *natural generalization of linear programming*: It includes all quadratic programming (QP) and quadratically constrained programming (QCP) problems, and many other problems in quantitative finance and engineering design. SOCP problems are always **convex**, and they can be solved quickly and reliably to very large size.

Since the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform supports second order cone constraints, SOCP problems can be solved by either *special-purpose*, high-performance Solvers (the SOCP Barrier Solver and MOSEK Solver), or by *general-purpose* nonlinear Solvers such as the standard GRG Solver, the Large-Scale GRG and SQP Solvers, and the KNITRO Solver engine.

## All different Constraints

In the standard Excel Solver, you can specify that certain decision variables must have integer values at any feasible or optimal solution. (As a special case, binary or 0-1 integer variables are often used to represent yes/no decisions in an optimization model.) In previous versions of the Premium Solver, solution times of problems involving integer variables were often greatly improved, but the types of problem conditions you could model with integer variables remained the same.

At times, you'll encounter a problem where you want to specify that a set of integer variables (typically representing an ordering of choices) *must all be different* at the solution. An example is the Traveling Salesman Problem (TSP), where a salesman must choose the order of cities to visit so as to minimize travel time, and each city must be visited exactly once. This condition is difficult to model using conventional constraints and integer variables.

In the Premium Solver and Premium Solver Platform, you can specify directly that a set of variables must be “all different.” Such variables will then have integer values from 1 to  $N$  (the number of variables), all of them different at the solution. All of the bundled Solver engines support this new type of constraint: The Branch & Bound method used by the Simplex LP (or LP/Quadratic), SOCP Barrier, nonlinear GRG,

and Interval Global Solvers is extended to handle all different constraints as a native type, and the Evolutionary Solver implements these constraints using mutation and crossover operators for permutations. Field-installable Solver engines also support the all different constraint, implementing it in different ways. This allows you to model your problem in a high-level way, and try a variety of Solver engines to see which one yields the best performance on your problem.

## Simulation Optimization

The Premium Solver Platform and Risk Solver Engine create a powerful combination for solving simulation optimization problems – Frontline's first step in a long-term plan to empower you to build and solve large-scale models that yield good or optimal decisions in the presence of uncertainty.

Simulation optimization problems include both decision variables and *random variables* whose values are uncertain, defined by probability distributions. The objective and constraints can depend on both the decision variables and on risk measures and other statistics computed from functions of the random variables.

For example, if your model describes uncertain market demand for your products, and computes sales, inventory levels and Net Profit, you can solve a problem that seeks to maximize the expected value of Net Profit subject to constraints on maximum (say, 90<sup>th</sup> percentile) or minimum (10<sup>th</sup> percentile) inventory levels.

When you click Solve, the Premium Solver Platform performs a Monte Carlo simulation through Risk Solver Engine on each iteration of the optimization. Each Monte Carlo simulation involves thousands of trials, where a different value is sampled for each random variable on each trial. Doing this is usually an order of magnitude faster than in competitive products for "simulation optimization."

## New Types of Reports

Where the standard Excel Solver offers three types of reports – the Answer Report, Sensitivity Report, and Limits Report – the Premium Solver offers six types of reports: The Answer, Sensitivity, and Limits Reports, and the new Linearity Report, Feasibility Report, and Population Report. And the Premium Solver Platform offers ten types of reports: The first six just listed, and the Solutions Report, Scaling Report, Structure Report, and Transformation Report. You can also select automatic outlining for the first six reports, which will organize the variables and constraints into outlined groups corresponding to the blocks you entered in the Solver Parameters dialog. You can identify each block of variables and constraints with descriptive comments. This can make it much easier to find the information you need in the reports, for models with hundreds or thousands of variables and constraints.

### ***Linearity, Feasibility, and Population Reports***

When the Solver says that "The linearity conditions required by this Solver engine are not satisfied," you can produce a Linearity Report that shows you whether the objective and each constraint is a linear or nonlinear function of the variables, and also whether each variable occurs linearly or nonlinearly in the problem. With this information, you can locate and, if desired, eliminate nonlinear formulas in your model, thereby gaining the extra speed and accuracy available with a Solver designed for linear problems. (An even better solution – the Structure Report, produced by the Premium Solver Platform – is described below.)

When you receive the message that “Solver could not find a feasible solution,” this often means that you’ve made a mistake entering some constraint, such as using a  $>=$  relation when you meant  $<=$ . But it can be difficult to pinpoint the source of the error, especially if you have hundreds or thousands of constraints to examine. With the Premium Solver products, you can produce a Feasibility Report and let the Solver do the work. It will automatically re-solve the problem with subsets of the original constraints, until it isolates a subset of constraints (called an “Irreducibly Infeasible System” or IIS) which is infeasible, but which becomes feasible if any one of the constraints is removed. By examining just the constraints in the Feasibility Report, you can usually pinpoint the problem with your model very quickly.

When the Evolutionary Solver stops with a “best solution,” you have the option of producing a standard Answer Report and/or a new Population Report. Where the Answer Report gives you detailed information about the single “best solution” returned by the Solver, the Population Report gives you summary information about the entire population of candidate solutions at the end of the solution process. The Population Report can give you insight into the performance of the Evolutionary Solver as well as the characteristics of your model, and help you decide whether additional runs of the Evolutionary Solver are likely to yield even better solutions.

### ***Solutions, Scaling, Structure, and Transformation Reports***

The **Solutions Report** – greatly expanded in **Version 7.0** – gives you objective function and decision variable values for a number of alternative solutions found during the optimization process. For mixed-integer problems, the report shows each incumbent or feasible integer solution found by the Branch & Bound method. For global optimization problems solved with the GRG, LSGRG, LSSQP, and KNITRO Solver engines, the report shows each locally optimal solution found by the Multistart method. For the Evolutionary and OptQuest Solvers, the report shows members of the final population of solutions.

The Solutions Report has a special meaning for the Interval Global Solver. It is available for problems with no objective function to be maximized or minimized, and with all equality constraints (a *system of equations*) or all inequality constraints (a *system of inequalities*). For a system of nonlinear equations, the Answer Report shows only a single solution, but the Solutions Report shows you *all real solutions*. For a system of inequalities, the Answer Report again shows you only a single feasible point, but the Solutions Report shows you an “inner solution” – a *region* or set of points where all of the constraints are satisfied.

In the Premium Solver Platform, a new choice – the **Scaling Report** – appears in the Solver Results dialog whenever solving yields an error message or an outcome that might be due to a poorly scaled model – such as “Solver converged to the current solution” or “The linearity conditions required by this Solver engine are not satisfied.” When you select this report, the Polymorphic Spreadsheet Interpreter evaluates all Excel formulas in your model while keeping track of the magnitudes or scales of intermediate results, and reports cases that may lead to a loss of accuracy.

Using the Premium Solver Platform’s new **Solver Model** dialog, which controls the Polymorphic Spreadsheet Interpreter, you can diagnose and transform your model before you solve it, produce the Structure Report to pinpoint problems in your model, and produce the Transformation Report to show how certain problematic functions were automatically replaced with “better” functions.

For the **Structure Report**, you simply select the type of model you meant to create – linear, quadratic, smooth nonlinear, or non-smooth – and ask the Solver to report any exceptions to this desired model type. The Structure Report is both more useful and

more reliable than the Linearity Report, because it evaluates your model symbolically rather than numerically (so it cannot be “fooled” by poorly scaled models), and it can pinpoint not just overall constraints and variables, but individual cell formulas where the dependence of constraints on variables is nonlinear (if your assumed model is an LP) or non-smooth (if your assumed model is an NLP). These cell formulas are reported as “exceptions” in the Structure Report, with hyperlinks to the actual cells containing the formulas in question. In **Version 7.0**, the Structure Report also tells you whether your objective and each constraint are **convex** or **non-convex** functions, when you ask the Interpreter to check the model for convexity.

The **Transformation Report** in **Version 7.0** can be produced when you ask the Interpreter to automatically transform your model to replace non-smooth functions such as IF, MIN, MAX, ABS, AND, OR, or NOT with additional variables and linear constraints that have the same effect as the replaced functions. This report lists the new variables and constraints that are added to your model by the transformation process. See the chapter “Analyzing and Solving Models” for more details.

## User Interface Improvements

User interface improvements in the Premium Solver products are designed to give you more information, help you move more quickly through the Solver dialogs, and accomplish what you want in fewer steps.

Have you ever wondered whether your model was truly linear or smooth nonlinear, or (if it wasn't) exactly which formulas caused the model to be nonlinear or non-smooth? Or – having learned about how the large-scale Solver engines exploit sparsity in a model – have you wondered just how sparse *your* model actually is, or whether it could be solved faster – for example, by the Large-Scale SQP Solver – by exploiting linear constraints or linearly occurring variables wherever possible? The new Solver Model dialog in the Premium Solver Platform gives you answers to all these questions.

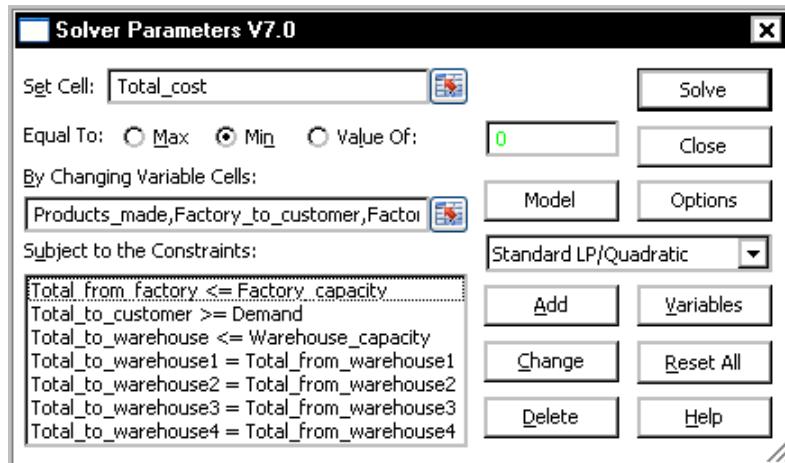
Have you ever wondered about the size of the problem you've defined, and whether it's getting close to the size limits supported by a given Solver engine? In the Premium Solver products, you can examine the number of variables, constraints, variable bounds and integer variables in your problem and the corresponding size limits at any time, by displaying the Problem tab in the Solver Options dialog available for each Solver engine.

Have you ever had trouble remembering the purpose of a block of constraints in your model? In the Premium Solver products, you can add descriptive comments to each block, which will appear in each of the Solver's reports.

Have you ever wondered how long the Solver will take, for a larger model, when “Setting Up Problem ...” appears on the status bar, or when the Solver is building a Limits Report? Thanks to improved progress reporting in the Premium Solver products, you'll know: An estimated “% Done” appears on the status bar in each of these situations.

Have you ever wanted to see more of the constraints at one time in the Constraints List box in the Solver Parameters dialog? In **Version 7.0**, simply use your mouse to resize this dialog like any other window – as shown in the example on the next page.





Have you ever found yourself selecting the Tools Solver... menu choice again and again, as you solve variants of the same problem? In the Premium Solver products, a check box in the Solver Results dialog lets you return to the Solver Parameters dialog directly, where you can change settings and click the Solve button immediately.

If you solve an integer problem and find that there is no feasible solution, the Solver Results dialog provides an option to immediately solve the “relaxation,” temporarily ignoring the integer constraints. And if there is still no feasible solution, you’ll have the option to create a Feasibility Report, to find out why.

Have you ever found it difficult to enter all of your variables (Changing Cells) via a multiple selection in the single edit box provided by the standard Solver? The Premium Solver products provide a Variables button which allows you to switch between a single edit box and a multi-line display, similar to the Constraints list box, for entering and modifying an unlimited number of variable selections.

Finally, the Premium Solver products provide access to algorithmic methods and tolerances used in each of the bundled Solver engines. By simply clicking on check boxes and radio buttons, or entering values in edit boxes in the Solver Options dialogs, you can control key tolerances in the Simplex method, stopping conditions for the nonlinear GRG Solver, the number of subproblems and integer solutions to be explored by the Branch & Bound method, the population size, mutation rate, and other options for the Evolutionary Solver, the interval methods and tolerances used by the new Interval Global Solver, and interior point methods and tolerances used by the SOCP Barrier Solver.

## Speed Improvements

The Premium Solver offers a range of speed and accuracy improvements over the standard Excel Solver. The Premium Solver Platform realizes a whole new level of speed and accuracy, compared to the Premium Solver and earlier versions of the Platform, thanks to model analysis and automatic differentiation performed by the new Polymorphic Spreadsheet Interpreter.

### **Premium Solver**

For large Excel models, the time spent “Setting Up Problem ...” can be significant – often more than the time spent on “Trial Solutions” for LP problems. The Premium Solver supports *fast problem setup* for linear programming models, which can yield answers up to 100 times faster than in the standard Excel Solver, for models that use a restricted set of Excel functions and conform to “fast problem setup format.”

The Standard Simplex LP Solver is about three times faster than the standard Excel Solver, and also uses less memory. On specific models, the Premium Solver is even faster, thanks to new “steepest edge pricing” methods. The nonlinear GRG Solver in the Premium Solver supports a new option, “Recognize Linear Variables,” that can speed up the solution process by as much as 50%, depending on how many variables occur linearly in the problem (which is, of course, nonlinear overall). And the Evolutionary Solver is often many times faster on problems with linear or smooth nonlinear constraints.

For linear mixed integer (LP/MIP) models, the Premium Solver **Version 7.0** uses a wide range of methods that often make it hundreds of times faster than the standard Excel Solver. These include pseudocost branching, a new Dual Simplex method with bound-swapping (for all problems with integer variables), new Preprocessing and Probing options (for problems with 0-1 integer variables), and new cut generation methods that automatically add new constraints to the problem, “cutting off” parts of the LP feasible region without eliminating any potential integer feasible solutions.

### ***Premium Solver Platform***

In the Premium Solver Platform, *fast problem setup* is still available for LP and QP models in restricted format, but the Polymorphic Spreadsheet Interpreter can speed up problem setup for virtually all models, regardless of the Excel formulas and functions they use.

The Polymorphic Spreadsheet Interpreter also greatly speeds up the solution process. Although solution times vary from model to model, on a comparative test of actual user LP models, the Premium Solver Platform was *two to five times* faster than the Premium Solver on average. And with the Interpreter’s automatic differentiation facilities, on a comparative test of actual user NLP models, the Premium Solver Platform was *seven times faster* than the Premium Solver on average!

On LP/MIP models, speedups are even more dramatic, thanks to powerful branching and cut generation methods in the Premium Solver Platform’s LP/Quadratic Solver. Although solution times vary greatly, a ten-fold speedup over the Premium Solver, and hundreds of times faster than the standard Excel Solver, would not be unusual.

Even more dramatic is the effect of the SOCP Barrier Solver in the Premium Solver Platform **Version 7.0** on problems with quadratic constraints that formerly required the GRG Nonlinear Solver, and on problems with second order cone constraints that could only be expressed with nonlinear analytic formulas previously: The SOCP Barrier Solver solves these nonlinear problems *nearly as fast as linear problems* of equivalent size!

## **Programmability Improvements**

The Premium Solver and Premium Solver Platform are fully programmable from Excel’s Visual Basic Application Edition. This means that you can build an application using the GRG Nonlinear Solver, Simplex LP (or LP/Quadratic) Solver, Evolutionary Solver, Interval Global Solver, SOCP Barrier Solver, or a field-installable Solver engine, hide the Premium Solver user interface, and present your own customized user interface for your end users. To get started, you can turn on Excel’s Macro Recorder, build your Solver model interactively, and let the Solver generate a working Visual Basic program for you, which reproduces your interactive steps whenever it is run, and which includes direct calls to the VBA functions that control the Solver. A complete summary of the VBA functions supported by both the standard Excel Solver and the Premium Solver products is included in this Guide.

The Premium Solver products also provide programmatic access to new features such as the Variables list, the Solver Model dialog, the seven new types of reports, report outlining, and new options in the Solver Options dialogs for all Solver engines. Since the programmatic interface is upward compatible with the standard Excel Solver, you can use your existing VBA code, and extend it as much as you wish to utilize the new Premium Solver features.

Programmability is greatly expanded in **Version 7.0** of the Premium Solver and Premium Solver Platform, with a new **object-oriented API** (Application Programming Interface) that allows you to work with high-level objects such as a Problem, Solver, Engine, Model, and blocks of Variables and Functions. The object-oriented API is compatible with the Risk Solver Engine object-oriented API, and closely resembles the object-oriented API of the Solver Platform SDK V7.0.

---

## How to Use This Guide

“Installation” takes you through the simple steps required to install the Premium Solver product you have licensed, to work with your copy of Microsoft Excel. In **Version 7.0**, installation and licensing is easier and more flexible than ever: You can install the Premium Solver products without first installing the standard Excel Solver, share a Flexible Use license over a network, and add new license codes in Excel while the Solver is running, without re-running the Setup program.

“Solver Models and Optimization” – revised for **Version 7.0** to cover **simulation optimization** – reviews the basic framework of the optimization problems that can be handled by the Solver, from linear programming problems to the non-smooth optimization problems handled by the Evolutionary Solver, and the use of new cone constraints and alldifferent constraints. It describes how a model is made up of variables, constraints and an objective, and it covers the major types of optimization problems that you can solve – including **convex** and **non-convex** problems – and the tradeoffs involved.

“Building Solver Models” – significantly revised for **Version 7.0** – is an introduction to the art of building optimization models in Microsoft Excel, translating from algebraic notation to spreadsheet formulas and Solver Parameters dialog choices. It covers multiple selections for decision variables, use of the new Variables button, the possible forms of constraint left- and right-hand sides, use of the soc and src drop-downs for cone constraints, and use of the dif dropdown for alldifferent constraints. It also provides hints on how to build more readable, better-documented models, such as layout and formatting and use of defined names. It describes models with variables and constraints spread across multiple worksheets, and the use of new **PSI functions** to specify models in Version 7.0.

“Analyzing and Solving Models” – revised for **Version 7.0** – describes how to use the new Solver Model dialog to analyze your model for linear, quadratic, smooth nonlinear, and non-smooth constraints and decision variables, automatically select “valid,” “good” or “best” Solver engines, automatically transform your model to replace non-smooth functions with additional variables and linear constraints, automatically find nonlinear or non-smooth formulas that are “exceptions” to your desired model type, and control the operation of the Polymorphic Spreadsheet Interpreter when your model is solved.

“Building Large-Scale Models” provides a number of valuable hints when building large-scale spreadsheet models and using external data sources. It describes modeling techniques such as ratio constraints, fixed-charge constraints, either-or constraints, constraints for IF functions, piecewise linear constraints, and more. And

it shows how you can use array formulas and functions such as MMULT, and the SUM, SUMPRODUCT and special add-in functions to define your objective and constraint formulas in *fast problem setup* format.

“Simulation Optimization with Risk Solver Engine” – **new in Version 7.0** – explains how you can define and solve problems involving both decision variables (factors that you can control) and uncertain variables (factors you cannot control that affect your objective and constraints). The potent combination of the Premium Solver Platform and its array of Solver engines with Risk Solver Engine for high-speed Monte Carlo simulation gives you unprecedented power to solve these problems.

“Diagnosing Solver Results” helps you determine what is wrong if you don’t get the solution you expect from the Solver, or if you encounter a message other than “Solver found a solution.” It outlines the most common problems that users have, based on our technical support experience with the Solver. This chapter covers in some detail the strengths and limitations of the GRG Solver for smooth nonlinear problems, the multistart methods and the new Interval Global Solver for smooth global optimization problems, the Evolutionary Solver for non-smooth problems, and Simplex versus interior point methods for linear, quadratic, and conic optimization problems. It also provides hints on “what to do next” when you have a solution.

“Solver Options” documents in depth the advanced options and tolerances used by each of the bundled Solver engines – including the new SOCP Barrier Solver – which can be set using new multi-tabbed Solver Options dialogs. The effect of each option and situations where you would likely choose it are described.

“Solver Reports” describes the contents of reports that may be chosen from the Solver Results dialog. It shows you how to interpret the numbers in the reports, and how to use the Sensitivity Report to predict changes in the optimal solution in response to certain kinds of changes in your input data. This chapter also provides a detailed look at the new Population Report, Linearity Report and Feasibility Report in the Premium Solver, and the new **Solutions Report**, Scaling Report, Structure Report, and Transformation Report in the Premium Solver Platform **Version 7.0**.

“Using the Object-Oriented API” – **new in Version 7.0** – describes how you can use the new, high-level object model exposed by the Premium Solver Platform to more easily control the Solver, obtain solution values and dual values, and even create applications where the solution of one problem is used to choose and create new decision variables for another problem. The Premium Solver Platform’s new object-oriented API is compatible with the object-oriented APIs of Risk Solver Engine, Frontline’s new tool for high-speed Monte Carlo simulation, and Solver Platform SDK, Frontline’s new and very popular tool for creating optimization and simulation applications in a programming language.

“Using Traditional VBA Functions” describes how you can control the Solver using the “traditional” VBA functions, upward compatible from the standard Excel Solver, in Visual Basic Application Edition, use the Macro Recorder to create macros corresponding to your interactive choices, look up the Solver functions in the Object Browser, and create “turn-key” applications using the Solver.

---

## Using Online Help and the World Wide Web

The enhanced Help system included with your Premium Solver product gives you online access to much of the information in this Guide. To access this information, just click on the Help button in any of the Solver dialogs. To access Help for the *standard* Solver included with Microsoft Excel when a Premium Solver product is

installed, select Help from the main Excel menu bar. You can type “solver” in the Index, or look under “Performing What-If Analysis on Worksheet Data” and “Analyzing Multiple-Variable Problems” in the Contents.

When you click the Help button in the Solver dialogs, a dialog like the one shown below appears, that gives you easy access to examples installed with your Premium Solver product, examples on Frontline’s website, and our online tutorial.

Click the Continue button to return to the Solver dialog. Click the Help button in this dialog to access online reference information for the Premium Solver Platform.



If you’re just getting started with the Premium Solver Platform, **we highly recommend that you click the “Examples” button**, which will open the Examples.xls workbook that is installed with the other Solver files. This workbook contains seven worksheets with examples illustrating many new Premium Solver Platform features, including conic optimization and automatic transformation of a model with IF functions into an equivalent model with integer variables and linear constraints.

Clicking the **Online Examples** button will open a Web browser to our main Solutions page on **www.solver.com**, where you can download a series of additional example workbooks in Finance, Investment, Production, Distribution, Purchasing, and Scheduling. Clicking the **Tutorial Online** button will open a Web browser to the start of our highly regarded online tutorial about optimization.

You’ll find a wealth of other useful information about the standard Solver and the Premium Solver products at Frontline Systems’ Web site, **www.solver.com**. Since Solver.com is frequently updated, you’ll want to check it periodically for the latest news about the Solver.

If you have questions, check the Index in this Guide or in the online Help system. You can also submit questions via the Contact Us page on Solver.com, or send email to **info@solver.com**.

---

## Solver-Related Seminars and Books

Although this Guide will provide many valuable hints for making effective use of the Solver, it does not attempt to teach you how to formulate Solver models or apply linear and quadratic programming, smooth nonlinear and non-smooth optimization, or integer programming techniques. To make the most of the Solver, we strongly

recommend that you consult one of the books cited below, or discuss your problem with someone in your firm or at your local university with a background in operations research and/or management science. There is a vast literature on problems of various types and for various industries and business situations that have been solved successfully with the methods available in the Solver. Don't reinvent the wheel – find out how others have solved problems similar to yours!

You may also want to attend a public seminar on spreadsheet optimization and other advanced techniques in Excel. Because of the popularity of the Excel Solver, seminars taught by highly regarded instructors are offered in a variety of U.S. cities and other parts of the world. For the latest information on these seminars, visit [www.solver.com](http://www.solver.com) or contact us at [info@solver.com](mailto:info@solver.com).

***The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft*** by Stephen G. Powell and Kenneth R. Baker, published by John Wiley & Sons, ISBN 0-471-20937-6. This new textbook is a valuable aid for anyone using the Premium Solver or Premium Solver Platform, especially for users building large-scale models. Unlike any other current textbook, this book teaches you “best practices” in modeling and spreadsheet engineering, as well as techniques of linear and nonlinear optimization, Monte Carlo simulation, and data analysis using Excel. The Premium Solver for Education is included on CD-ROM with this book, but this version is much less powerful than the commercial Premium Solver or Premium Solver Platform.

***VBA for Modelers - Developing Decision Support Systems Using Microsoft Excel*** by S. Christian Albright, published by Duxbury Press, ISBN 0-534-38012-3. This unique book will prove invaluable to anyone seeking to use VBA (Visual Basic Application Edition) to programmatically control Microsoft Excel and build custom applications. It includes a basic introduction to VBA and the Excel object model, and 16 example applications developed in depth with VBA source code, many of them calling the Solver's VBA functions. The applications include blending, product mix, production scheduling and similar models, plus capital budgeting, stock trading, option pricing and portfolio optimization.

***Spreadsheet Modeling and Decision Analysis: A Practical Introduction to Management Science, 5<sup>th</sup> Edition*** by Cliff T. Ragsdale, published by South-Western College Publishing, ISBN 0-324-31256-3. This book, a favorite in new MBA courses on management science and decision analysis, features an in-depth tutorial treatment of optimization modeling, including linear and integer programming, nonlinear optimization, and evolutionary algorithms using Frontline's Evolutionary Solver (as well as other management science topics such as forecasting, regression analysis and simulation). This textbook also includes the Premium Solver for Education on CD-ROM.

***Practical Management Science, 3<sup>rd</sup> Edition*** by Wayne Winston and S. Christian Albright, published by Duxbury Press, ISBN 0-534-46512-9. This textbook, also widely used in new MBA courses on management science, provides an extensive introduction covering linear and integer programming, nonlinear optimization, and genetic and evolutionary algorithms using Frontline's Evolutionary Solver, as well as other management science topics. This book is slightly more challenging than Cliff Ragsdale's book, but includes an extensive set of spreadsheet models and a whole chapter on the Evolutionary Solver. It also includes the Premium Solver for Education on CD-ROM.

***Introduction to Mathematical Programming, 4<sup>th</sup> Edition*** by Wayne Winston and Munirpallam Venkataramanan, published by Duxbury Press, ISBN 0-534-35964-7. This book focuses entirely on optimization, at a more technical level than the textbooks described above – including topics in linear algebra, the Simplex method,

goal programming, integer programming and the Branch & Bound method, and the differential calculus topics underlying nonlinear optimization. It also includes the Premium Solver for Education on CD-ROM.

***Managerial Spreadsheet Modeling and Analysis*** by Richard Hesse, published by Richard D. Irwin, ISBN 0-256-21530-8. This “good, but hard to find” book teaches you how to formulate a model from a complex business situation, using a four-step process: Picture and paraphrase, verbal model, algebraic model and spreadsheet model. It covers types of models ranging from simple goalseeking and unconstrained problems to linear, nonlinear and integer programming problems. And it includes over 100 Microsoft Excel spreadsheets, covering a wide range of deterministic and stochastic models.

***Model Building in Mathematical Programming, 4<sup>th</sup> Edition*** by H.P. Williams, published by John Wiley, ISBN 0-471-99788-9. Though it doesn't cover spreadsheet optimization, this book is still valuable for its explanation of model-building approaches, especially if you are building large-scale optimization models. It provides an in-depth treatment of modeling for linear and integer programming problems. It mentions nonlinear models only briefly, but it offers a unique treatment of large-scale model structure and decomposition methods. It also includes a complete discussion of 24 models drawn from various industries.

## Academic References for the Premium Solver

The following academic journal articles, written by the developers of the Excel Solver, Premium Solver and Premium Solver Platform, describe many of the algorithms and technical methods used in these products. The first article describes the design of the original Excel Solver. You can download PDF versions of the first three articles at <http://www.solver.com/academic.htm>:

D. Fylstra, L. Lasdon, J. Watson and A. Waren. Design and Use of the Microsoft Excel Solver. *INFORMS Interfaces* 28:5 (Sept-Oct 1998), pp. 29-55.

I. Nenov and D. Fylstra. Interval Methods for Accelerated Global Search in the Microsoft Excel Solver. *Reliable Computing* 9 (2003): pp. 143–159.

D. Fylstra, “Introducing Convex and Conic Optimization for the Quantitative Finance Professional,” *Wilmott Magazine* (March 2005), pp. 18-22.

For a technical description of the nonlinear GRG solver included with the standard Microsoft Excel Solver and the Premium Solver, please consult the following:

L.S. Lasdon, A. Waren, A. Jain and M. Ratner. Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming. *ACM Transactions on Mathematical Software* 4:1 (1978), pp. 34-50.

L.S. Lasdon and S. Smith. Solving Sparse Nonlinear Programs Using GRG. *INFORMS Journal on Computing* 4:1 (1992), pp. 2-15.





# Installation and Licensing

---

## What You Need

In order to install the Premium Solver Platform V7.0, you must have first installed Microsoft Excel 2000, Excel XP, Excel 2003, or Excel 2007 on Microsoft Windows XP or Windows Server 2003 (standard or x64 versions) or Windows Vista. It is no longer necessary to have the standard Excel Solver installed.

The Premium Solver and Premium Solver Platform will run on the same hardware and system software configuration that you've used to run Microsoft Excel. If you try to solve very large models, however, performance may depend on the amount of main memory (RAM) in your system. Large models with many integer constraints can take substantially more time to solve, and require more memory than models without such constraints. Use of the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform may also require considerable memory. Steps you can take to improve performance are outlined in the chapter "Building Large-Scale Models."

## Working with Earlier Solver Versions

You can install the Premium Solver Platform V7.0 "alongside" the standard Excel Solver, or an earlier version of the Premium Solver Platform or Premium Solver, and *use both at the same time* interactively. You use the Premium Solver Platform V7.0 via menu choice **Tools Premium Solver**, and the other version via **Tools Solver**. In Excel 2007, you'll see **Premium Solver** and **Solver** on the **Add-Ins** tab.

By default, the PSPSetup program for Version 7.0 will install the Solver files to the folder C:\Program Files\Frontline Systems\Premium Solver Platform and its subfolders. The standard Excel Solver is installed within the Microsoft Office folder, for example C:\Program Files\Microsoft Office\Office11\Library\Solver. Earlier versions of the Premium Solver Platform and Premium Solver were also installed into this folder, backing up and replacing the standard Excel Solver.

**If your application uses VBA macros such as SolverOK and SolverSolve to control the Solver, and you have both the Premium Solver Platform V7.0 and an earlier Solver installed at the same time, you'll need to specify whether your VBA code (which contains a reference to the Solver.xla add-in) should control the Platform V7.0 or the earlier Solver version. To do this, you simply check or uncheck the box Load V7 VBA Macros in the Solver Model dialog Options tab in the Platform V7.0 to ensure that the V7.0 Solver.xla is loaded.**

In Excel 2007, the standard Excel Solver uses an add-in file **Solver.xlam** – saved in Excel's new file format, and named differently from previous versions. So you *can* use the standard Solver's VBA functions such as SolverSolve in a module with a reference to **Solver.xlam**, and also use the Platform V7.0 „traditional VBA functions in a module with a reference to **Solver.xla**. But a better option is to use the Platform V7.0's new *object-oriented API*, as described later in this User Guide.

*Note:* The Premium Solver Platform Version 7.0 works only with field-installable Solver engines Version 7.0. If the Platform V7.0 finds older versions of the Solver engines in its startup directory, you'll receive an error message "Solver Engine is incompatible with Premium Solver Platform." If you receive this message, please contact Frontline Systems at (775) 831-0300 or [info@solver.com](mailto:info@solver.com) for an upgrade under your Annual Support Contract.

---

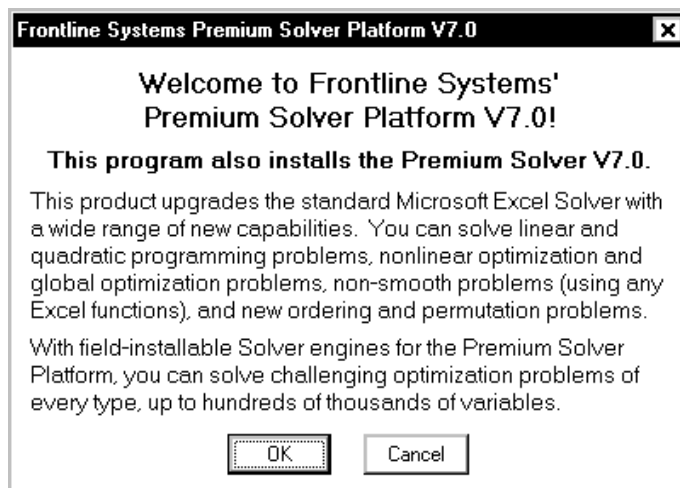
## Installing the Software

Installing the Premium Solver Platform or Premium Solver is a straightforward process. A graphical program **PSPSetup.exe**, which contains all of the Solver files in compressed form, will guide you through the steps involved.

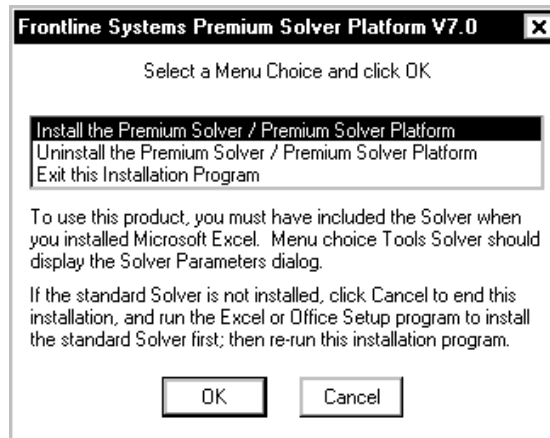
To begin the installation, insert the Frontline Systems CD or other media into your CD or disk drive. On most CD drives, the Setup program will start automatically. Frontline Systems CDs have a master program that displays a menu, allowing you to choose among several Setup programs. If nothing happens when you insert the CD, choose **Run...** from the **Start** menu. In the dialog box, type a drive letter such as **d:** if required, followed by **PSPSetup.exe** for *either* the Premium Solver or Premium Solver Platform. Then press ENTER or click OK.

If you downloaded PSPSetup.exe, depending on your Windows security settings, you might be prompted with a message "The publisher could not be verified. Are you sure you want to run this software?" You may safely click **Run** in response to this message.

After a brief pause, a dialog box like the one shown below should appear:



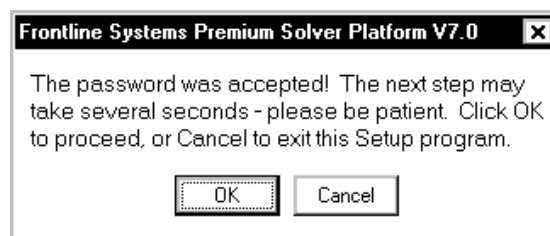
Press ENTER or click OK to proceed. Next, you'll choose the installation option you want from a dialog box like the one on the next page.



Make sure that the "Install" choice is highlighted, and press ENTER or click OK. You will then be prompted for a password for this installation, which Frontline Systems will provide to you. Enter it carefully into the dialog box, and click OK.



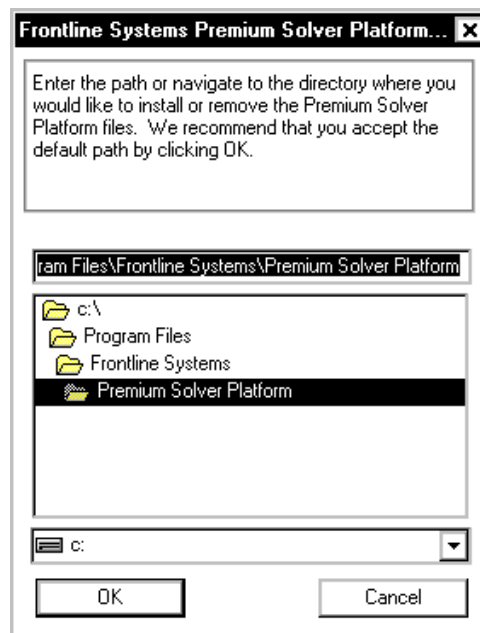
Once the password is accepted, the PSPSetup program will check whether the 15-day trial license has been used previously on this PC.



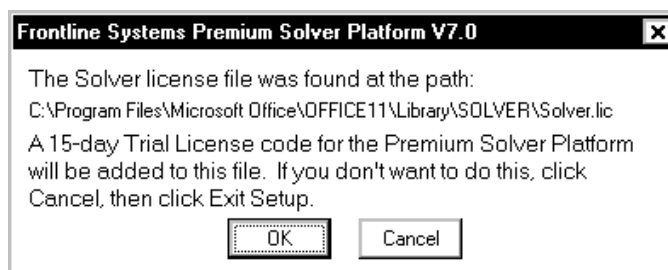
If the PSPSetup program Version 7.0 has been run previously on this computer, or if some other licensing problem was detected, you may see a dialog like the one below. If you have a problem, please contact Frontline Systems at (775) 831-0300 or info@solver.com and provide the numeric code shown in parentheses in the message.



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\Premium Solver Platform). We recommend that you simply click OK.

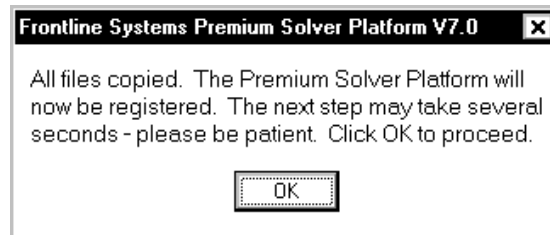


Next, the Setup program looks for a license file Solver.lic that may already exist on your system – normally the environment variable LSERVRC contains the path to this license file. If the license file is found, Setup will add a Version 7.0 license code to this file. If the license file is not found, Setup will install a Solver.lic file containing the Version 7.0 license code, in the installation folder that was just selected. A dialog appears explaining the action to be taken; you may proceed or cancel at this point.

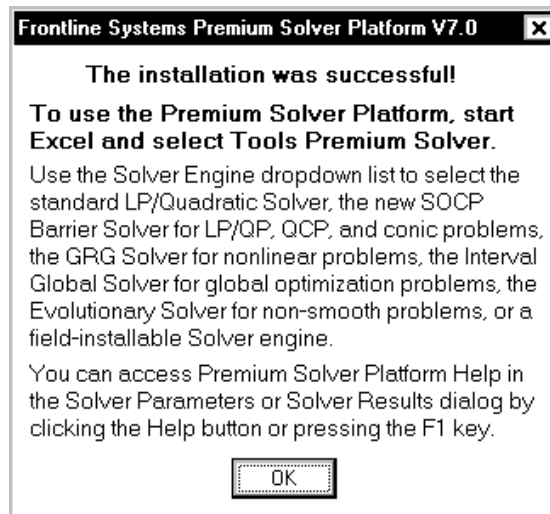


If you click OK, the Solver files will be copied to the installation folder, and the license file will be created or updated. If you click Cancel and then click Exit Setup to confirm, the Setup program will exit without copying any files.

A progress dialog appears as the Setup program copies the Premium Solver Platform files. When this is complete, a dialog like the one below appears to notify you that the program file Solver32.xll will be registered as a COM add-in (which takes a few seconds on some systems).



When the installation is complete, you will see a dialog box like the one below.



Be sure to read any special messages in the final dialog box(es) – they may give you important hints about new developments since this User Guide was updated. Then press ENTER or click OK. The Premium Solver Platform V7.0 is now installed. Simply run Microsoft Excel and choose **Tools Premium Solver...**, or in Excel 2007 select **Premium Solver** from the **Add-Ins** tab, to display the new Solver Parameters dialog. Initially, the Solver engine dropdown list will contain the names of the five built-in Version 7.0 Solver engines. If you install additional Solver engines V7.0, they will also appear in this dropdown list.

---

## Uninstalling the Software

To uninstall or remove any of the Premium Solver products, simply run the PSPSetup program (or the EngineSetup program for field-installable Solver Engines) and select the Uninstall choice from the menu in the second dialog box. You can uninstall Solver engines (with either evaluation or full licenses) without affecting the Premium Solver Platform.

You can also uninstall the Premium Solver Platform or any of the Solver engines by choosing **Settings Control Panel** from the **Start** menu (or just **Control Panel** in Windows XP), and double-clicking the **Add/Remove Programs** applet. In the list box below “Currently installed programs” or “The following software can be automatically removed by Windows...,” scroll down if necessary until you reach lines beginning with “Frontline,” select the Premium Solver Platform or the specific Solver engine that you want to remove, and click the **Add/Remove...** button. Click **OK** in the confirming dialog box to uninstall the software.

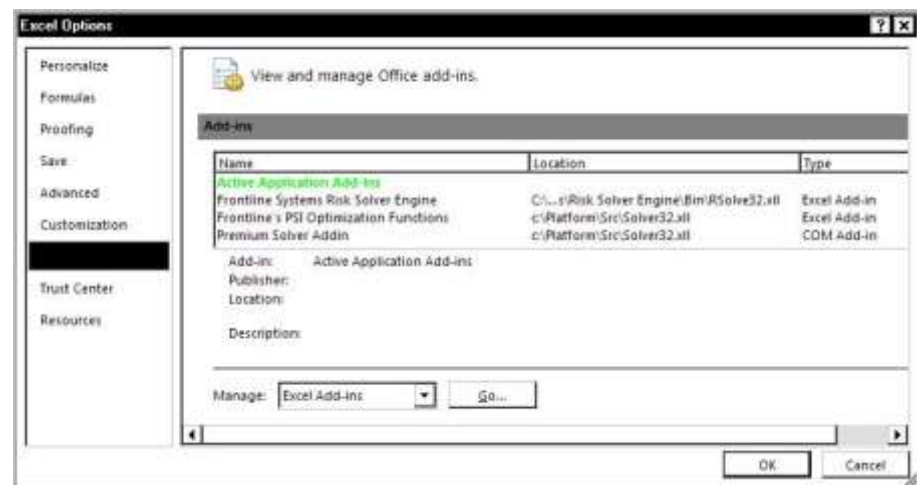
## Activating and Deactivating the Software

As explained in the Introduction, the Premium Solver Platform V7.0 uses a different architecture and interface to Excel than previous versions. Its main program file **Solver32.xll** is a COM add-in, an XLL add-in, and a COM server. In Version 7.0, the add-in file **Solver.xla** is optional – it is needed only if you wish to use the “traditional” VBA functions to program the Solver – and if the box **Load V7 VBA Macros** in the Solver Model dialog Options tab is checked, Solver32.xll will load the V7.0 Solver.xla automatically, as described in the next section.

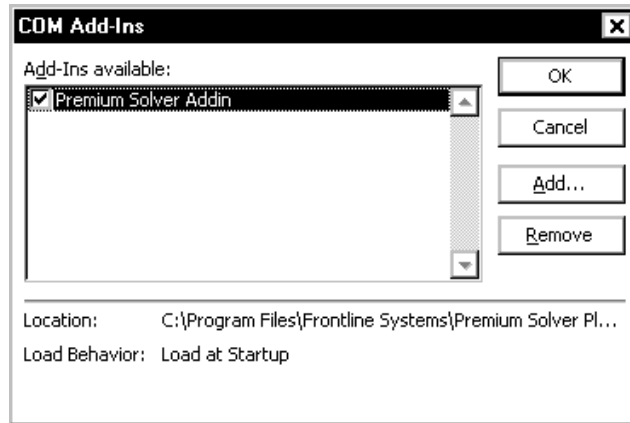
You can activate and deactivate the Premium Solver Platform V7.0, but the procedure is different from previous Solver versions, and the dialogs are slightly different in Excel 2007 and Excel 2000-2003.

### Excel 2007

In Excel 2007, you can manage all types of add-ins from one dialog, reached by clicking the upper left corner button, choosing Excel Options, then choosing Add-ins in the pane on the left, as shown below.



You can manage add-ins by selecting the type of add-in from the dropdown list at the bottom of this dialog. For example, if you select COM Add-ins from the dropdown list and click the Go button, the dialog shown on the next page appears.



If you uncheck the box next to “Premium Solver Add-In” and click **OK**, you will deactivate the Premium Solver Platform V7.0 COM add-in, which will remove Premium Solver from the Add-Ins tab, and also remove the PSI functions for optimization from the Excel 2007 Function Wizard.

## Excel 2003 and Earlier

In earlier versions of Excel, COM add-ins and other add-ins are managed in separate dialogs, and the COM Add-In dialog is available only if you display a toolbar which is hidden by default. To display this toolbar:

1. On the **View** menu, point to **Toolbars**, and then click **Customize**.
2. Click the **Commands** tab.
3. Under **Categories**, click **Tools**.
4. Under **Commands**, click **COM Add-Ins** and drag your selection to the toolbar at the top of the Excel window.

Once you have done this, you can click **COM Add-Ins** on the toolbar to see a list of the available add-ins in the COM Add-Ins dialog box, as shown above.

If you uncheck the box next to “Premium Solver Add-In” and click **OK**, you will deactivate the Premium Solver Platform V7.0 COM add-in, which will remove Premium Solver from the Tools menu, and also remove the PSI functions for optimization from the Insert Function dialog.

---

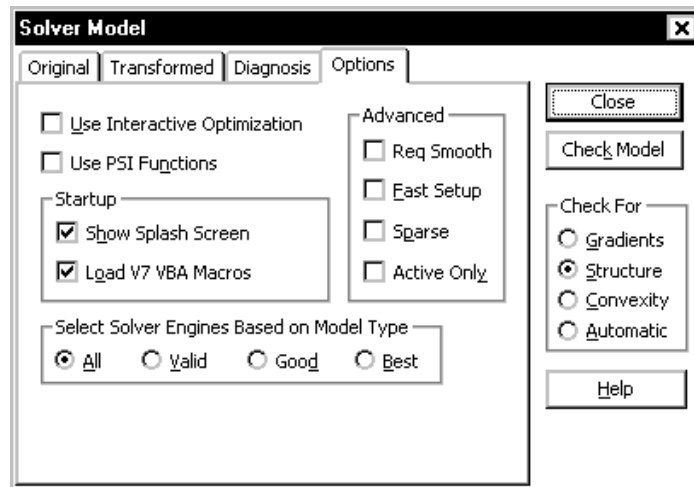
## Setting Startup Options

By default, when the Premium Solver Platform V7.0 COM add-in Solver32.xll is first activated – either when you start Excel or when you activate the add-in as described above – it displays a “splash screen” to let you know that it is available, and it ensures that the Version 7.0 Solver.xla is also loaded. If it finds an older version of Solver.xla already loaded, it will unload this version and load the V7.0 Solver.xla. You can change these behaviors by setting options in the Solver Model dialog, as described in this section.

**Solver.xla** is the add-in that defines the “traditional” VBA functions, such as SolverOK and SolverSolve, used to programmatically control the Solver. When the V7.0 version of Solver.xla is loaded, your VBA program calls to the traditional Solver functions will work with the Premium Solver Platform V7.0.

In the standard Excel Solver and in earlier versions of the Premium Solver and Premium Solver Platform, Solver.xla provides **both** the interactive dialogs and the “traditional” VBA functions. Hence, if you want to use the Premium Solver Platform V7.0 and an earlier Solver version *at the same time*, you must ensure that the earlier version of Solver.xla is loaded, and that the V7.0 Solver.xla is *not* loaded automatically by Solver32.xll.

To do this, select **Tools Premium Solver** and, in the Solver Parameters dialog, click the **Model** button. This will display the Solver Model dialog. Click the Options tab in this dialog, which will display a dialog pane like the one below.



The **Startup** options group in this dialog pane is summarized here. The options “Use Interactive Optimization” and “Use PSI Functions” are described in the chapter “Building Solver Models.” In the Premium Solver, an abbreviated dialog consisting of just these options and the Startup options is displayed. All of the options on this pane are documented in the chapter “Analyzing Solver Models.”

## Show Splash Screen

If this box is checked, a “splash screen” is briefly shown when the Premium Solver Platform COM add-in is first activated – typically when you first start Excel. If this box is unchecked, no splash screen is shown.

## Load V7 VBA Macros

If this box is checked, the Premium Solver Platform V7.0 COM add-in will load the V7.0 Solver.xla automatically; any earlier version of Solver.xla will be unloaded. This ensures that your VBA program calls to functions like SolverOK and SolverSolve will be executed by the Premium Solver Platform V7.0.

If this box is unchecked, the Premium Solver Platform V7.0 COM add-in will *not* load the V7.0 Solver.xla automatically. You must manually ensure that the version of Solver.xla that you want (V7.0, V6.x, standard Excel Solver, etc.) is loaded, using the File Open or Tools Add-Ins menu commands.

To use the standard Excel Solver or an earlier version of the Premium Solver or Premium Solver Platform *at the same time* as the Premium Solver Platform V7.0, ensure that the earlier version of Solver.xla is loaded. Use **Tools Solver** to access the Solver Parameters dialog of the earlier version, and **Tools Premium Solver** to access



the V7.0 Solver Parameters dialog. The “traditional” VBA functions will work with the earlier version; the new object-oriented API always works with Version 7.0.

**Note:** Earlier versions of the Premium Solver Platform installed an add-in function module **DotPrd32.xll** that defined functions DOTPRODUCT, QUADPRODUCT and QUADTERM. This module is not required in Version 7.0; the main program file Solver32.xll automatically defines these functions.

---

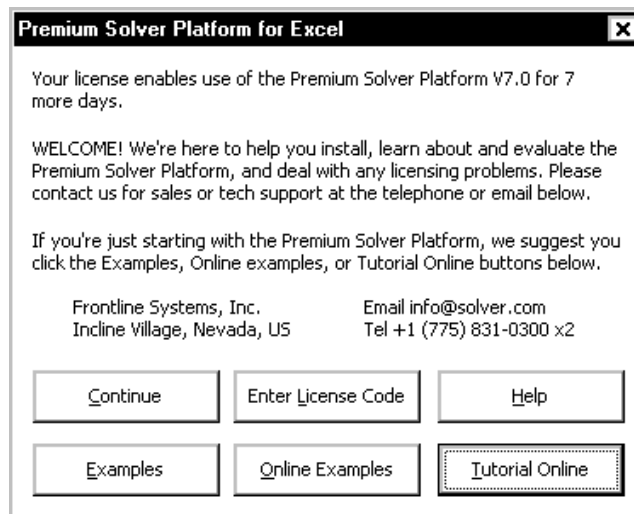
## Licensing the Software

A **license** is a grant of rights, from Frontline Systems to you, to use our software in specified ways. Information about a license is encoded in a **license code**. Several types of licenses are available for both the Premium Solver Platform and field-installable Solver engines.

Frontline offers two basic types of licenses for **development** of your model and application: Standalone Licenses and Flexible Use (often called “Concurrent”) Licenses. A Standalone License enables use on a single PC; a Flexible Use License enables shared use among several PCs on a network. The “Software License and Limited Warranty” section in the front of this User Guide describes the details of these licenses. Other types of licenses are available for **runtime** use: See the discussion on **www.solver.com**, then contact Frontline Systems for advice on license types best suited for your situation.

You can evaluate the Premium Solver Platform free of charge for a limited period – typically 15 days – using a special trial license code that is copied to your PC by the PSPSetup program. To obtain license codes that enable use of the Premium Solver Platform for longer periods of time or on a permanent basis, please contact Frontline Systems at (775) 831-0300 or **info@solver.com**.

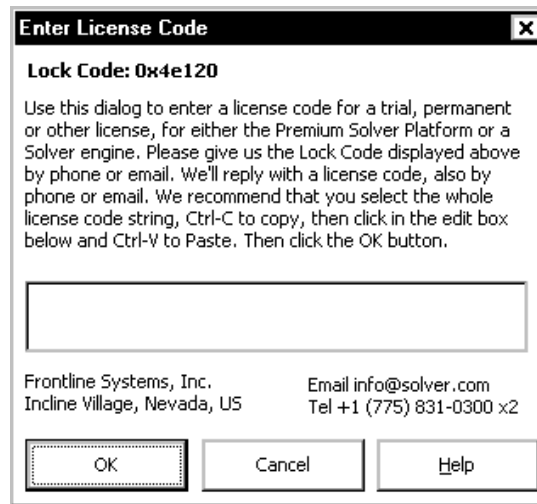
If your trial or time-limited license code is within one week of its expiration date, or if some other licensing problem was detected, you will see a dialog box like the one below when you first select Tools Premium Solver. You can display this dialog at any time by clicking the Help button in the main Solver Parameters dialog.



Click the **Continue** button to return to the Solver Parameters dialog; you will be able to create or edit a Solver model using the Add, Change and Delete buttons, but if your trial license is expired, you will receive a licensing error message if you click the

Solve button or the Check Model button in the Solver Model dialog. Click the **Help** button to display online Help information for the Premium Solver Platform.

Click the **Enter License Code** button to display the dialog shown below.



As instructed in this dialog, contact Frontline Systems at (775) 831-0300 or send email to [info@solver.com](mailto:info@solver.com). Give us the Lock Code displayed in this dialog. When you receive a license code (most often by email) from Frontline Systems, select the entire code, copy and paste it into the edit box as shown, and click OK. That's all there is to it! Your new license code should be activated immediately.

If you are unable to add the license code using the Enter License Code dialog box, you can use NotePad or WordPad to edit the file `Solver.lic` (it is a plain ASCII text file), and append the license code to the end of the file.

If you're just getting started with the Premium Solver Platform, the initial Help dialog is a great place to start. We highly recommend that you click the **Examples** button, which will open the `Examples.xls` workbook that is installed with the other Solver files. You can also click the **Online Examples** button or the **Tutorial Online** button, which will open a Web browser to additional examples or our tutorial on [www.solver.com](http://www.solver.com).

---

## Installing Solver Engines

To install additional Solver engines, you'll follow steps similar to those outlined above for the Premium Solver Platform, but you'll be running the program **EngineSetup.exe**. When you select the "Install" choice from the menu, you'll be prompted for an installation password, which will be different from the one used for the PSPSetup program.

You can evaluate any or all of the field-installable Solver engines free of charge for a limited period – typically 15 days – using a special trial license code that is copied to your PC by the EngineSetup program. (This 15-day period runs independently from any evaluation period for the Premium Solver Platform.) To obtain license codes that enable use of a specific Solver engine for longer periods of time or on a permanent basis, please contact Frontline Systems at (775) 831-0300 or [info@solver.com](mailto:info@solver.com).

After you press ENTER or click OK in the final EngineSetup dialog box, your Solver engine is installed and ready to use. Simply run Microsoft Excel and choose Tools Premium Solver... to display the Solver Parameters dialog. Then open the Solver

engine dropdown list – the name of your new Solver engine should appear in the list. Click the Options button to display a Solver Options dialog for your new Solver engine. If you click the Help button in this Solver Options dialog, you'll gain access to Solver engine-specific Help – including information on the status of your license, if you are evaluating the Solver engine.

You can enter a new license code for a Solver engine just as described above for the Premium Solver Platform: Starting in the Solver engine Options dialog, click the Help button (this will display the license status and/or time remaining for the Solver engine trial license), then click the button Enter License Code. You can also enter a license code by clicking the Help button directly in the Solver Parameters dialog; the Help dialog will display the license status and/or time remaining for the *Premium Solver Platform* and its bundled Solver Engines rather than the field-installable Solver engines, but you can still use the Enter License Code button in *this* dialog to enter a license code for any Solver engine.



# Solver Models and Optimization

---

## Introduction

This chapter explains the principles behind spreadsheet Solvers, including the types of problems you can solve, types of constraints (regular, integer, conic, all different) you can specify, the nature of linear, quadratic and nonlinear functions, convex and non-convex functions, smooth and non-smooth functions, and the algorithms and methods used by the Premium Solver Platform and field-installable Solver engines.

If you are just starting out with the Solver, you may find it helpful to read the first section below, “Elements of Solver Models,” and then proceed to the next chapter, “Building Solver Models,” for a hands-on example. If you have been using the Solver for a while, and you’d like a more in-depth review of the mathematical relationships found in Solver models, and the optimization methods and algorithms used by the Solver, read the later, more advanced sections of this chapter. We recommend that even experienced users read the new sections on **convex and conic optimization** and on **simulation optimization** in this chapter.

---

## Elements of Solver Models

The basic purpose of the Solver is to find a *solution* – that is, values for the *variables* or Changing Cells in your model – that satisfies the *constraints* and that maximizes or minimizes the *objective* or Set Cell value (if there is one). Let’s examine this framework more closely.

The model you create for use with the Solver is no different from any other spreadsheet model. It consists of input values; formulas that calculate values based on the input values or on other formulas; and other elements such as formatting. You can practice “what if” with a Solver model just as easily as with any other spreadsheet model. This familiar concept can be very useful when you wish to present your results to managers or clients, who are usually “spreadsheet literate” even if they are unfamiliar with Solvers or optimization.

### Decision Variables and Parameters

Some of the input values may be fixed numbers, which you cannot change in the course of finding a solution – for example, prevailing interest rates or supplier prices. We’ll call these values *parameters* of the model. Often you will have several “cases,” “scenarios,” or variations of the same problem to solve, and the parameter

values will change in each problem variation. Such parameter values may be conveniently captured using the Excel Scenario Manager. But the parameter values will be fixed numbers for any given run of the Solver – unless you re model the parameters as uncertain values, as discussed below under “Simulation Optimization.”

Other input values may be quantities that are variable, or under your control in the course of finding a solution. We'll refer to these as the variables, *decision variables*, or Changing Cells. The Solver will find optimal values for these variables or cells. Often, some of the same cell values you use to play “what if” are the ones for which you'll want the Solver to find solution values. These cells are listed in the By Changing Variable Cells edit box of the Solver Parameters dialog.

## The Objective Function

The quantity you want to maximize or minimize is called the *objective function* or Set Cell. This cell is listed in the Set Cell edit box of the Solver Parameters dialog. For example, this could be a calculated value for projected profits (to be maximized), or costs, risk, or error values (to be minimized).

You may have a Solver model that has nothing to maximize or minimize, in which case the Set Cell edit box will be blank. In this situation the Solver will simply find a solution that satisfies the constraints. Typically this will be only one of (infinitely) many such solutions, located close to the starting values of the decision variables.

The Excel Solver also permits you to enter a specific value that you want the objective function or Set Cell to achieve. This feature was included for compatibility with the Goal Seek... command in Excel, which allows you to seek a specific value for a cell by adjusting the value of one other cell on which it depends. In fact, entering a specific value for the Solver's Set Cell is exactly the same as leaving the Set Cell blank and entering an equality constraint for the Set Cell in the Constraint List Box.

There is rarely a good reason to use the Set Cell Value of edit box in the Solver Parameters dialog. If your problem requires only a single Set Cell value and a single variable or Changing Cell with no constraints, you can just use the Goal Seek... command. If you have nothing to maximize or minimize, we recommend that you leave the Set Cell blank and enter all of your constraints in the Constraint List Box.

## Constraints

Constraints are relations such as  $A1 \geq 0$ . A constraint is *satisfied* if the condition it specifies is true *within a small tolerance*. This is a little different from a logical formula such as  $=A1 \geq 0$  evaluating to TRUE or FALSE which you might enter in a cell. In this example, if A1 were -0.0000001, the logical formula would evaluate to FALSE, but with the default Solver Precision setting, the constraint would be satisfied. Because of the numerical methods used to find solutions to Solver models and the finite precision of computer arithmetic, it would be unrealistic to require that constraints like  $A1 \geq 0$  be satisfied exactly – such solutions would rarely be found.

In the Excel Solver, constraints are specified by giving a cell reference such as A1 or A1:A5 (the “left hand side”), a relation ( $\leq$ ,  $=$  or  $\geq$ ), and an expression for the “right hand side.” Although Excel allows you to enter any numeric expression on the right hand side, for reasons that will be explained in the chapter “Building Large-Scale Models,” we strongly encourage you to use only *constants*, or references to cells that contain *constant values* on the right hand side. (A constant value to the Solver is any value that does not depend on any of the decision variables.)

A constraint such as  $A1:A5 \leq 10$  is shorthand for  $A1 \leq 10, A2 \leq 10, A3 \leq 10, A4 \leq 10, A5 \leq 10$ . A constraint such as  $A1:A5 \leq B1:B5$  is shorthand for  $A1 \leq B1, A2 \leq B2, A3 \leq B3, A4 \leq B4, A5 \leq B5$ .

Another type of constraint is of the form  $A1:A5 = \text{integer}$ , where  $A1:A5$  are decision variables. This specifies that the solution values for  $A1$  through  $A5$  must be integers or whole numbers, such as -1, 0 or 2, *to within a small tolerance*. This form of constraint, and related forms such as  $A1:A5 = \text{binary}$  and  $A1:A5 = \text{alldifferent}$ , are explored in the next section.

A new type of constraint supported by the Premium Solver Platform is of the form  $A1:A5 = \text{conic}$ , where  $A1:A5$  are decision variables. This is called a *second order cone* constraint and is further described in the next section.

## Solutions: Feasible, “Good” and Optimal

A solution (set of values for the decision variables) for which all of the constraints in the Solver model are satisfied is called a *feasible solution*. In some problems, a feasible solution is already known; in others, finding a feasible solution may be the hardest part of the problem.

An *optimal solution* is a feasible solution where the objective function reaches its maximum (or minimum) value – for example, the most profit or the least cost. A *globally optimal solution* is one where there are no other feasible solutions with better objective function values. A *locally optimal solution* is one where there are no other feasible solutions “in the vicinity” with better objective function values – you can picture this as a point at the top of a “peak” or at the bottom of a “valley” which may be formed by the objective function and/or the constraints.

The Solver is designed to find feasible and optimal solutions. In the best case, it will find the globally optimal solution – but this is not always possible. In other cases, it will find a locally optimal solution, and in still others, it will stop after a certain amount of time with the best solution it has found so far. But like many users, you may decide that it is most important to find a *good solution* – one that is better than the solution, or set of choices, you are using now.

The kind of solution the Solver can find depends on the nature of the mathematical relationships between the variables and the objective function and constraints (and the solution algorithm used). As explained below, if your model is *smooth convex*, you can expect to find a globally optimal solution; if it is smooth but *non-convex*, you will usually be able to find a locally optimal solution; if it is *non-smooth*, you may have to settle for a “good” solution that may or may not be optimal.

Below, we summarize the capabilities of the five Solver engines bundled with the Premium Solver Platform: the LP/Quadratic Solver, SOCP Barrier Solver, nonlinear GRG Solver, Interval Global Solver, and Evolutionary Solver. (The Premium Solver includes the Linear Simplex Solver – a subset of the LP/Quadratic Solver, the nonlinear GRG Solver, and the Evolutionary Solver.) Later sections of this chapter provide an overview of the optimization methods and algorithms employed by each of these Solver engines.

### Linear Simplex and LP/Quadratic Solver

The linear Simplex Solver finds optimal solutions to problems where the objective and constraints are all linear functions of the variables. (The term *linear function* is explained below, but you can imagine its graph as a straight line.) Since all linear functions are *convex*, the Solver normally can find the *globally optimal solution*, if one exists. Because a linear function (a straight line) can always be increased or

decreased without limit, the optimal solution is always determined by the constraints; there is no natural “peak” or “valley” for the objective function itself.

In the Premium Solver Platform, the linear Simplex Solver is extended to the LP/Quadratic Solver. This Solver handles problems where the constraints are all linear, and the objective may be linear or quadratic (explained further below). If the quadratic objective function is **convex** (if minimizing, or **concave** if maximizing) the Solver will normally find a globally optimal solution. If the objective is **non-convex** (further explained below), the Solver will find only a locally optimal solution.

### **SOCP Barrier Solver**

The SOCP Barrier Solver in the Premium Solver Platform finds optimal solutions to problems where the objective and constraints are all linear or convex quadratic functions of the variables. (This is in contrast to the LP/Quadratic Solver, which permits only the objective function to be quadratic.) It also finds optimal solutions to problems with a linear objective, linear constraints, and *second order cone* (SOC) constraints; this is called a second order cone programming (SOCP) problem, as explained further below. Since all linear functions and SOC constraints are **convex**, the SOCP Barrier Solver normally finds a *globally optimal solution*, if one exists.

### **Nonlinear GRG Solver**

The nonlinear GRG Solver finds optimal solutions to problems where the objective and constraints are all smooth (**convex** or **non-convex**) functions of the variables. (The term *smooth function* is explained below, but you can imagine a graph – whether straight or curved – that contains no “breaks.”) For non-convex problems, the Solver normally can find a *locally optimal solution*, if one exists – but this may or may not be the globally optimal solution. A nonlinear objective function can have a natural “peak” or “valley,” but in most problems the optimal solution is partly or wholly determined by the constraints. The nonlinear GRG Solver can be used on problems with all-linear functions, but it is much less effective and efficient than the LP/Quadratic Solver or the SOCP Barrier Solver on such problems.

If you use multistart methods for global optimization with the nonlinear GRG Solver, you will have a better chance (but not a guarantee) of finding the globally optimal solution. The idea behind multistart methods is to automatically start the Solver from a variety of starting points, to find the best of the locally optimal solutions – ideally the globally optimal solution. These methods are more fully described (and contrasted with other methods for global search) below under “Global Optimization” and in the chapter “Solver Options.”

### **Interval Global Solver**

The Interval Global Solver finds *globally optimal solutions* to problems where the objective and constraints are all smooth (**convex** or **non-convex**) functions of the variables. Unlike the Evolutionary Solver or the GRG Solver with multistart methods, the Interval Global Solver is normally able to determine *for certain* that the solution is globally optimal. The tradeoff is that the Interval Global Solver usually takes much more time to solve a given problem than the GRG Solver, and this time rises steeply as the number of variables and constraints in the problem increases. Hence, the Interval Global Solver is practically able to solve only smaller problems, compared to the GRG Solver.



## Evolutionary Solver

The Evolutionary Solver usually finds *good solutions* to problems where the objective and constraints include non-smooth functions of the variables – in other words, where there are no restrictions on the formulas that are used to compute the objective and constraints. For this class of problems, the Solver will return the best feasible solution (if any) that it can find in the time allowed.

The Evolutionary Solver can be used on problems with all-smooth functions that may have multiple locally optimal solutions, in order to seek a globally optimal solution, or simply a better solution than the one found by the nonlinear GRG Solver alone; however, the Interval Global Solver or the combination of multistart methods and the GRG Solver are likely to do as well or better than the Evolutionary Solver on such problems. It can be used on problems with smooth convex functions, but it is usually less effective and efficient than the nonlinear GRG Solver on such problems. Similarly, it can be used on problems with all-linear functions, but there is little point in doing so when the Simplex, LP/Quadratic, or SOCP Barrier Solver is available.

---

## More About Constraints

This section explains in greater depth the role of certain types of constraints, including bounds on the decision variables, equality and inequality constraints, second order cone constraints, and different forms of integer constraints.

### Bounds on the Variables

Constraints of the form  $A1 \geq -5$  or  $A1 \leq 10$  (for example), where  $A1$  is a decision variable, are called *bounds on the variables* and are treated specially by the Solver. These constraints affect only one variable, whereas general constraints have an indirect effect on several variables that have been used in a formula such as  $A1 + A2$ . Each of the Solver engines takes advantage of this fact to handle bounds on the variables more efficiently than general constraints.

The most common type of bound on a variable is a lower bound of zero ( $A1 \geq 0$ ), which makes the variable non-negative. Many variables represent physical quantities of some sort, which cannot be negative. As a convenience, the Solver Options dialog offers a check box “Assume Non-Negative,” which automatically places a lower bound of zero on every variable which has not been given an explicit lower bound via a constraint in the Constraints list box.

Regardless of the Solver engine chosen, bounds on the variables always help speed up the solution process, because they limit the range of values that the Solver must explore. In many problems, you will be aware of realistic lower and upper bounds on the variables, but they won't be of any help to the Solver unless you include them in the Constraints list box! Bounds on the variables are especially important to the performance of the Evolutionary Solver, the Interval Global Solver, and multistart methods for global optimization, as discussed below under “Global Optimization” and in the chapter “Solver Options.” They are also very important if you want the Solver to automatically transform your model, replacing non-smooth functions (such as IF) with additional variables and linear constraints, as explained in the chapter “Analyzing and Solving Models.”

### Equations and Inequalities

Constraints such as  $A1 = 0$  are called *equality constraints* or *equations*; constraints such as  $A1 \leq 0$  are called *inequality constraints* or simply *inequalities*. An equality

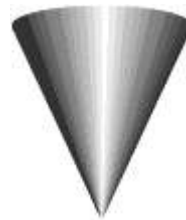
is much more restrictive than an inequality. For example, if A1 contains the formula  $=C1+C2$ , where C1 and C2 are decision variables, then  $A1 \leq 0$  restricts the possible solutions to a *half plane*, whereas  $A1 = 0$  restricts the solutions to a *line* where all possible values of C1 and C2 must sum to 0 ( $C1 = -C2$  within a small tolerance, as explained above). Since there is only a tiny chance that two randomly chosen values for C1 and C2 will satisfy  $C1+C2 = 0$ , solution methods that rely on random choices, such as genetic algorithms, may have a hard time finding any feasible solutions to problems with equality constraints. To satisfy equality constraints, the Solver generally must exploit properties of the constraint formula – such as *linearity* or *smoothness*, discussed below – to solve for one variable in terms of another.

A linear equality constraint (like  $C1+C2 = 0$  above) maintains the convexity of the overall problem, but a nonlinear equality constraint is **non-convex**, and makes the overall problem non-convex. Interior point methods may have difficulty solving problems with nonlinear equality constraints, since they restrict the ability of the Solver to follow the “central path” inside the feasible region.

A problem with *only* equality constraints (and no objective) is sometimes called a *system of equations*. The Solver can be used to find solutions to systems of both linear and nonlinear equations. If there are several different solutions (sets of values for the decision variables) that satisfy the equations, most Solver engines will find just one solution that is “close” to the starting values of the variables; but the Interval Global Solver in the Premium Solver Platform can be used to find *all* real solutions to a system of smooth nonlinear equations – a capability that was once felt to be beyond the limits of any known algorithm.

## Second Order Cone Constraints

The Premium Solver Platform supports constraints of the form  $A1:A5 = \text{conic}$ . This is called a *second order cone* (SOC) constraint; it specifies that the vector formed by the decision variables A1:A5 must lie within the second-order cone (also called the Lorentz cone, or “ice cream cone”) of dimension 5 – a **convex** set that looks like the figure below in three dimensions.



Algebraically, a second-order cone constraint specifies that, given a value for one variable, the  $L_2$ -norm of the vector formed by the remaining variables must not exceed this value: In linear algebra notation,  $a_1 \geq \|a_2:a_5\|_2$ . In Excel, this could be written as  $A1 \geq \text{SQRT}(\text{SUMSQ}(A2:A5))$ . You can also use a variant called a “rotated second order cone” constraint, as explained in the chapter “Building Solver Models.” A problem with a linear objective and linear or SOC constraints is called a *second order cone programming* (SOCP) problem; it is always a **convex** optimization problem.

Decision variables that are constrained to be non-negative also belong to a cone, called the *non-negative orthant*. A problem with all linear functions – a linear programming problem – is a special case of an SOCP problem, where the only cone constraint is non-negativity.

A convex quadratic objective or constraint can be transformed into an equivalent second order cone constraint. Hence, a problem with a quadratic objective – a

quadratic programming or QP problem – or a problem with quadratic constraints – called a QCP problem – is also a special case of an SOCP problem. The SOCP Barrier Solver and the MOSEK Solver will automatically transform quadratics into SOC form internally; you can simply define your quadratic objective and/or constraints using ordinary Excel formulas and  $\leq$  or  $\geq$  relations, and use these Solver engines to obtain fast, reliable, globally optimal solutions to your problem.

### ***Integer, Binary and Alldifferent Constraints***

As explained in the last section, integer constraints are of the form  $A1:A5 = \text{integer}$ , where  $A1:A5$  are decision variables. This specifies that the solution values for  $A1$  through  $A5$  must be integers or whole numbers, such as -1, 0 or 2, to within a small tolerance. A common special case that can be entered directly in the Constraint List Box is  $A1 = \text{binary}$ , which is equivalent to specifying  $A1 = \text{integer}$ ,  $A1 \geq 0$  and  $A1 \leq 1$ . This implies that  $A1$  must be *either 0 or 1* at the solution; hence  $A1$  can be used to represent a “yes/no” decision. Integer constraints have many important applications, but the presence of even one such constraint in a Solver model makes the problem an integer programming problem (discussed below), which may be much more difficult to solve than a similar problem without the integer constraint.

The Premium Solver and Premium Solver Platform support a new type of integer constraint, called the “alldifferent” constraint. Such a constraint is of the form (for example)  $A1:A5 = \text{alldifferent}$ , where  $A1:A5$  is a group of two or more decision variables, and it specifies that these variables must be integers in the range 1 to  $N$  ( $N = 5$  in this example), with each variable different from all the others at the solution. Hence,  $A1:A5$  will contain a *permutation* of integers, such as 1,2,3,4,5 or 1,3,5,2,4. The alldifferent constraint can be used to model problems involving ordering of choices, such as the Traveling Salesman Problem.

---

## **Functions of the Variables**

Since there are large differences in the time it takes to find a solution and the *kinds* of solutions – globally optimal, locally optimal, or simply “good” – that you can expect for different types of problems, it pays to understand the differences between linear, quadratic, smooth nonlinear, and non-smooth functions, and especially **convex** and **non-convex** functions. To begin, let’s clarify what it means to say that the spreadsheet cells you select for the objective and constraints are “functions of the decision variables.”

The objective function in a Solver problem is a cell calculating a value that depends on the decision variable cells; the job of the Solver is to find some combination of values for the decision variables that maximizes or minimizes this cell’s value. During the optimization process, *only the decision variable cells are changed*; all other “input” cells are held constant. If you analyze the chain of formulas that calculates the objective function value, you will find that parts of those formulas (those which refer to non-decision variable cells) are unchanging in value and could be replaced by a numeric constant for the purposes of the optimization.

If you have constant values on the right hand sides of constraints, then the same observation applies to the left hand sides of constraints: Parts of the constraint formulas (those which refer to non-decision variable cells) are unchanging in value, and only the parts that are dependent on the decision variables “count” during the optimization.

When you consider whether your objective and constraints are linear, quadratic, smooth nonlinear, or non-smooth, or **convex** or **non-convex** functions of the

variables, always bear in mind that only the parts of formulas that are *dependent on the decision variables* “count.” Below, we explain that linear functions are most desirable, and non-smooth and non-convex functions are least desirable in a Solver model (if you want the fastest and most reliable solutions). A formula such as  $=IF(C1 \geq 10, D1, 2 * D1)$  is non-smooth if C1 depends on the decision variables; but if C1 *doesn't* depend on the variables, then only D1 or 2\*D1 – not both – can be selected during the solution process. Hence if D1 is a linear function of the variables, then the IF expression is also a linear function of the variables.

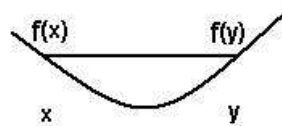
You may also find that a function that is “bad” (non-smooth or non-convex) over its full domain (any possible values for the decision variables) may be “good” (smooth and/or convex) over the domain of interest to you, determined by other constraints including bounds on the variables. For example, if C1 depends on the variables, then  $=IF(C1 \geq 10, D1, 2 * D1)$  is non-smooth over its full domain, but smooth – in fact linear – if C1 is constrained to be 10 or more.  $=SIN(C1)$  is non-convex over its full domain, but is convex from  $-\pi$  to 0, or from  $\pi$  to  $2 * \pi$ .

## Convex Functions

The key property of functions of the variables that makes a problem “easy” or “hard” to solve is *convexity*. If *all* constraints in a problem are convex functions of the variables, and if the objective is convex if minimizing, or concave if maximizing, then you can be confident of finding a globally optimal solution (or determining that there is no feasible solution), even if the problem is very large – thousands to hundreds of thousands of variables and constraints.

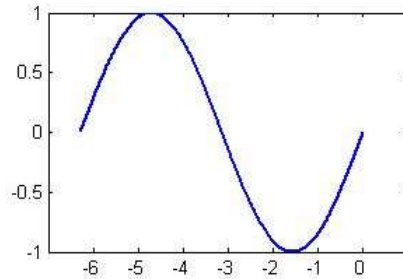
In contrast, if *any* of the constraints are non-convex, or if the objective is either non-convex, concave if minimizing, or convex if maximizing, then the problem is far more difficult: You cannot be certain of finding a feasible solution even if one exists; you must either “settle for” a locally optimal solution, or else be prepared for very long solution times and rather severe limits on the size of problems you can solve to global optimality (a few hundred to perhaps one thousand variables and constraints), even on the fastest computers. So it pays to understand convexity!

Geometrically, a function is *convex* if, at any two points  $x$  and  $y$ , the line drawn from  $x$  to  $y$  (called the *chord* from  $x$  to  $y$ ) lies *on or above* the function – as shown in the diagram below, for a function of one variable. A function is *concave* if the chord from  $x$  to  $y$  lies *on or below* the function. This property extends to any number of dimensions or variables, where  $x = (x_1, x_2, \dots, x_n)$  and  $y = (y_1, y_2, \dots, y_n)$ .



Algebraically, a function  $f$  is *convex* if, for any points  $x$  and  $y$ , and any  $t$  between 0 and 1,  $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$ . A function  $f$  is *concave* if  $-f$  is convex, i.e. if  $f(tx + (1-t)y) \geq tf(x) + (1-t)f(y)$ . A *linear* function – described below – is both convex and concave: The chord from  $x$  to  $y$  lies on the line, and  $f(tx + (1-t)y) = tf(x) + (1-t)f(y)$ . As we'll see, a problem with all linear functions is the simplest example of a convex optimization problem that can be solved efficiently and reliably to very large size.

A non-convex function “curves up and down.” A familiar example is the sine function ( $SIN(C1)$  in Excel), which is pictured on the next page.



The feasible region of an optimization problem is formed by the intersections of the constraints. The intersection of several convex constraints is always a convex region, but even one non-convex function can make the whole region non-convex – and hence make the optimization problem far more difficult to solve.

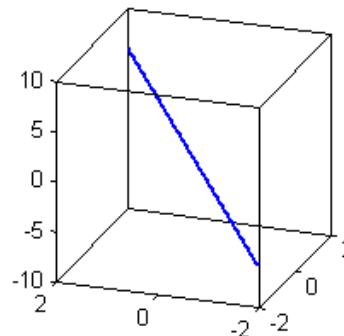
## Linear Functions

In many common cases, the objective and/or constraints are *linear* functions of the variables. This means that the function can be written as a sum of terms, where each term consists of one decision variable multiplied by a (positive or negative) constant. Algebraically, we can write:

$$a_1x_1 + a_2x_2 + \dots + a_nx_n$$

where the  $a$ s, which are called the *coefficients*, stand for constant values and the  $x$ s stand for the decision variables. A common example is `=SUM(C1:C5)`, where C1:C5 are decision variables and the  $a$ s are all 1. Note that a linear function does not have to be written in exactly the form shown above on the spreadsheet. For example, if cells C1 and C2 are decision variables,  $B1 = C1 + C2$ , and  $B2 = A1 * B1$  where A1 is constant in the problem, then B2 is a linear function ( $=A1 * C1 + A1 * C2$ ).

Geometrically, a linear function is always a straight line, in  $n$ -dimensional space where  $n$  is the number of decision variables. Below is a perspective plot of  $2x_1 + 1x_2$ . As noted above, a linear function is always convex.



Remember that the  $a$ s need only be *constant in the optimization problem*, i.e. not dependent on any of the decision variables. For example, suppose that the function is  $=B1/B2 * C1 + (D1 * 2 + E1) * C2$ , where only C1 and C2 are decision variables, and the other cells contain constants (or formulas that don't depend on the variables). This would still be a linear function, where  $a_1 = B1/B2$  and  $a_2 = (D1 * 2 + E1)$  are the coefficients, and  $x_1 = C1$  and  $x_2 = C2$  are the variables.

Note that the SUMPRODUCT and DOTPRODUCT functions compute exactly the algebraic expression shown above. If we were to place the formula `=B1/B2` in cell

A1, and the formula  $= (D1 * 2 + E1)$  in cell A2, then we could write the example function above as:

$= \text{SUMPRODUCT}(A1:A2, C1:C2)$

This is simple and clear, and is also useful for *fast problem setup* as described in the chapter “Building Large-Scale Models.” If the decision variable cells that should participate in the expression are not all contiguous on the spreadsheet, the DOTPRODUCT function can be used instead of SUMPRODUCT.

As explained below in the section “Derivatives, Gradients, Jacobians and Hessians,” each coefficient  $a_i$  in the linear expression  $a_1x_1 + a_2x_2 + \dots + a_nx_n$  is the first partial derivative of the expression with respect to variable  $x_i$ . These partial derivatives are always *constant* in a linear function – and all higher-order derivatives are zero.

A *nonlinear* function (explained further below), as its name implies, is any function of the decision variables which is not linear, i.e. which cannot be written in the algebraic form shown above – and its partial derivatives are *not* constant. Examples would be  $= 1/C1$ ,  $= \text{LOG}(C1)$ ,  $= C1^2$  or  $= C1 * C2$  where both C1 and C2 are decision variables. If the objective function or any of the constraints are nonlinear functions of the variables, then the problem cannot be solved with an LP Solver.

### Testing for a Linear Model

What if you have already created a complex spreadsheet model without using functions like SUMPRODUCT, and you aren't sure whether your objective function and constraints are linear or nonlinear functions of the variables? If you have the Premium Solver Platform, you can easily find out by pressing the Check Model button in the Solver Model dialog, as explained in the chapter “Analyzing and Solving Models.” Moreover, you can easily obtain a report showing exactly which cells contain formulas that are nonlinear.

If you have the Premium Solver, you can try solving the model with the standard Simplex LP Solver. If the problem contains nonlinear functions of the variables, you will (in virtually all cases) receive the message “The linearity conditions required by this Solver engine are not satisfied” in the Solver Results dialog. You can then ask the Solver to produce a Linearity Report, which shows whether the objective and each of the constraints is a linear or nonlinear function of the variables. This report also shows which variables occur linearly, and which occur nonlinearly in your model – another way of summarizing the same information. You should next look closely at the objective or constraint formulas that the Linearity Report indicates are nonlinear, and decide whether (or not) the formula can be written in linear form.

### Quadratic Functions

The last two examples of nonlinear functions above,  $= C1^2$  or  $= C1 * C2$ , are simple instances of *quadratic* functions of the variables. A more complex example is:

$= 2 * C1^2 + 3 * C2^2 + 4 * C1 * C2 + 5 * C1$

A quadratic function is a sum of terms, where each term is a (positive or negative) constant (again called a *coefficient*) multiplied by a single variable or the product of two variables. In linear algebra notation, we can write  $x^T Q x + c x$  where  $x$  is a vector of  $n$  decision variables,  $Q$  is an  $n \times n$  matrix of coefficients, and  $c$  is an  $n$  vector of linear coefficients. The QUADPRODUCT function computes values of exactly this form. If we put the constant 5 in A1, 0 in B1, 2 in A2, 4 in B2, 0 in A3 and 3 in B3, then we could write the above example as:

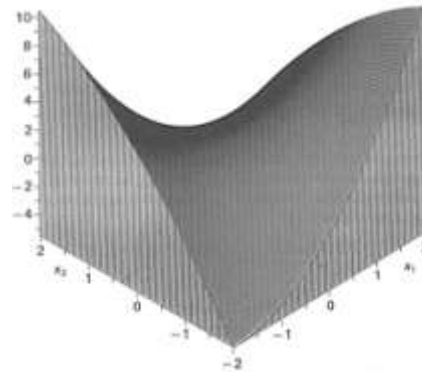
$= \text{QUADPRODUCT}(C1:C2, A1:B1, A2:B3)$

Common uses for quadratic functions are to compute the mean squared error in a curve-fitting application, or the variance or standard deviation of security returns in a portfolio optimization application.

As explained below in the section “Derivatives, Gradients, Jacobians and Hessians,” the coefficients that multiply single variables in a quadratic function are the first partial derivatives of the function with respect to those variables; the coefficients that multiply the *products* of two variables are the *second* partial derivatives of the function, with respect to those two variables. In a quadratic function, these first and second order derivatives are always *constant*, and higher order derivatives are zero. The matrix Q of second partial derivatives is called the *Hessian* of the function.

## Convex, Concave and Non-Convex Quadratics

A quadratic function of at least two variables may be convex, concave, or non-convex. The matrix Q in the general form  $x^T Q x$  has a closely related algebraic property of *definiteness*. If the Q matrix is *positive definite*, the function is convex; if the Q matrix is *negative definite*, the function is concave. You can picture the graph of these functions as having a “round bow I” shape with a single bottom (or top). If the Q matrix is *semi-definite*, the function has a bow I shape with a “trough” where many points may have the same objective value, but it is still convex or concave. If the Q matrix is *indefinite*, the function is **non-convex**: It has a “saddle” shape, but its true minimum or maximum is not found in the “interior” of the function but on its boundaries with the constraints, where there may be many locally optimal points. Below is a plot of an example non-convex quadratic  $x_1^2 + 2x_1x_2 - \frac{1}{2}(x_2^2 - 1)$ :



A problem with **convex** quadratic functions is easily solved to global optimality up to very large size, but a problem with **non-convex** quadratic functions is a difficult global optimization problem that, in general, will require solution time that grows *exponentially* with the number of variables. The way that the Solver handles such functions is explained further below under “Quadratic Programming.”

## Nonlinear and Smooth Functions

A *nonlinear* function is any function of the variables that is not linear, i.e. which cannot be written in the algebraic form:

$$a_1x_1 + a_2x_2 + \dots + a_nx_n$$

Examples, as before, are  $=1/C1$ ,  $=\text{LOG}(C1)$ , and  $=C1^2$ , where C1 is a decision variable. All of these are called *continuous* functions, because their graphs are curved but contain no “breaks.”  $=\text{IF}(C1 > 10, D1, 2 * D1)$  is also a nonlinear function, but it is “worse” (from the Solver’s view point) because it is *discontinuous*: Its graph contains a “break” at  $C1 = 10$  where the function value jumps from  $D1$  to  $2 * D1$ . At

this break, the rate of change (i.e. the derivative) of the function is undefined. As explained below in the section “Derivatives, Gradients, Jacobians and Hessians,” most Solver algorithms rely on derivatives to seek improved solutions, so they may have trouble with a Solver model containing functions such as =IF(C1>10,D1,2\*D1). The Interval Global Solver does not accept discontinuous functions at all.

If the graph of the function's *derivative* also contains no breaks, then the original function is called a *smooth* function. If it does contain breaks, then the original function is *non-smooth*. Every discontinuous function is also non-smooth. An example of a continuous function that is non-smooth is =ABS(C1) – its graph is an unbroken “V” shape, but the graph of its derivative contains a break, jumping from -1 to +1 at C1=0. Many nonlinear Solver algorithms rely on second order derivatives of at least the objective function to make faster progress, and to test whether the optimal solution has been found; they may have trouble with functions such as =ABS(C1). The Interval Global Solver does not accept any non-smooth functions.

As explained below in the section “Derivatives, Gradients, Jacobians and Hessians,” general nonlinear functions have first, second, and sometimes higher order derivatives that *change* depending on the point (i.e. values of the decision variables) at which the function is evaluated.

### **Convex, Concave and Non-Convex Smooth Functions**

A general nonlinear function of even one variable may be convex, concave or non-convex. A function can be convex but non-smooth: =ABS(C1) with its V shape is an example. A function can also be smooth but non-convex: =SIN(C1) is an example. But the “best” nonlinear functions, from the Solver's point of view, are both *smooth* and *convex* (concave for the objective if you are maximizing).

If a smooth function's second derivative is always nonnegative, it is a *convex* function; if its second derivative is always nonpositive, it is a *concave* function. This property extends to any number of dimensions or variables, where the second derivative becomes the Hessian and “nonnegative” becomes “positive semidefinite.”

## **Discontinuous and Non-Smooth Functions**

Microsoft Excel provides a very rich formula language, including many functions that are discontinuous or non-smooth. As noted above, discontinuous functions cause considerable difficulty, and non-smooth functions cause some difficulty for most nonlinear Solvers; such functions are not accepted by the Interval Global Solver. Some models can only be expressed with the aid of these functions; in other cases, you have a degree of choice in how you model the real-world problem, and which functions you use. Even when you have a “full arsenal” of Solver engines available, as you do with the Premium Solver products, you'll get better results if you try to use the most “Solver-friendly” functions in your model.

By far the most common discontinuous function in Excel is the IF function where the conditional test depends on the decision variables, as in the example =IF(C1>10,D1,2\*D1). Here is a short list of common *discontinuous* Excel functions:

IF, CHOOSE  
LOOKUP, HLOOKUP, VLOOKUP  
COUNT  
INT, ROUND  
CEILING, FLOOR



Here is a short list of common *non-smooth* Excel functions:

ABS  
MIN, MAX

Formulas involving relations such as  $\leq$ ,  $=$  and  $\geq$  (on the worksheet, not in the Constraints list box) and logical functions such as AND, OR and NOT are discontinuous at their points of transition from FALSE to TRUE values. Functions such as SUMIF and the database functions are discontinuous if the criterion or conditional argument depends on the decision variables.

If you aren't sure about a particular function, try graphing it (by hand or in Microsoft Excel) over the expected range of the variables; this will usually reveal whether the function is discontinuous or non-smooth. If you have the Premium Solver Platform, just create a model using the function, and use the Solver Model dialog to automatically diagnose the model type.

The Premium Solver Platform Version 7.0 can **automatically transform** a model that uses IF, AND, OR, NOT, ABS, MIN and MAX, and relations  $<$ ,  $\leq$ ,  $\geq$  and  $>$  to an equivalent model where these functions and relations are replaced by additional binary integer and continuous variables and additional constraints, that have the same effect – for the purpose of optimization – as the replaced functions. This powerful facility may be able to transform your non-smooth model into a smooth or even linear model with integer variables. A real-life example is shown in the EXAMPLE5 worksheet of the Examples.xls workbook, installed with the Solver files, which you can easily open from the Solver Parameters dialog by clicking Help, then clicking Examples. For more information, see the chapter “Analyzing and Solving Models.”

## Derivatives, Gradients, Jacobians, and Hessians

To find feasible and optimal solutions, most optimization algorithms rely heavily on derivatives of the problem functions (the objective and constraints) with respect to the decision variables. First derivatives indicate the direction in which the function is increasing or decreasing, while second derivatives provide curvature information.

The partial derivatives of a function  $f(x_1, x_2, \dots, x_n)$  with respect to each variable are denoted  $f'/x_1, f'/x_2, \dots, f'/x_n$ . They give the rate of change of the function in each dimension. For a linear function  $a_1x_1 + a_2x_2 + \dots + a_nx_n$ , the partial derivatives are the coefficients:  $f'/x_1 = a_1, f'/x_2 = a_2$ , and so on.

To recap the comments about derivatives made in the sections above:

Linear functions have *constant* first derivatives – the coefficients  $a_i$  – and all higher order derivatives (second, third, etc.) are zero.

Quadratic functions have *constant* first and second derivatives, and all higher order (third, etc.) derivatives are zero.

Smooth nonlinear functions have first and second derivatives that are *defined*, but not constant – they change with the point at which the function is evaluated.

Non-smooth functions have second derivatives that are *undefined* at some points; discontinuous functions have first derivatives that are undefined at some points.

The *gradient* of a function  $f(x_1, x_2, \dots, x_n)$  is the vector of its partial derivatives:

$$[ f'/x_1, f'/x_2, \dots, f'/x_n ]$$

This vector points in the direction (in n-dimensional space) along which the function increases most rapidly. Since a Solver model consists of an objective and constraints,

all of which are functions of the variables  $x_1, x_2, \dots, x_n$ , it is often useful to collect these gradients into a matrix, where each row is the gradient vector for one function:

$$\begin{vmatrix} f_1/x_1 & f_1/x_2 & \dots & f_1/x_n \\ f_2/x_1 & f_2/x_2 & \dots & f_2/x_n \\ \dots & \dots & \dots & \dots \\ f_m/x_1 & f_m/x_2 & \dots & f_m/x_n \end{vmatrix}$$

This matrix is called the *Jacobian* matrix. In a linear programming problem, this is the LP coefficient matrix, and all of its elements (the  $a_{ij}$ ) are constant.

The second partial derivatives of a function  $f(x_1, x_2, \dots, x_n)$  with respect to each pair of variables  $x_i$  and  $x_j$  are denoted  $\partial^2 f / \partial x_i \partial x_j$ . There are  $n^2$  second partial derivatives, and they can be collected into an  $n \times n$  matrix:

$$\begin{vmatrix} \partial^2 f / \partial x_1 \partial x_1 & \partial^2 f / \partial x_1 \partial x_2 & \dots & \partial^2 f / \partial x_1 \partial x_n \\ \partial^2 f / \partial x_2 \partial x_1 & \partial^2 f / \partial x_2 \partial x_2 & \dots & \partial^2 f / \partial x_2 \partial x_n \\ \dots & \dots & \dots & \dots \\ \partial^2 f / \partial x_n \partial x_1 & \partial^2 f / \partial x_n \partial x_2 & \dots & \partial^2 f / \partial x_n \partial x_n \end{vmatrix}$$

This matrix is called the *Hessian* matrix. It provides second order (curvature) information for a single problem function, such as the objective. The Hessian of a linear function would have all zero elements; the Hessian of a quadratic function has all *constant* elements; and the Hessian of a general nonlinear function may change depending on the point (values of the decision variables) where it is evaluated.

When reading the next section, “Optimization Problems and Solution Methods,” bear in mind that the different classes of Solver problems, and the computing time required to solve these problems, is directly related to the nature of the derivatives (constant, changing, or undefined) of their problem functions, as outlined above.

For example, because the first derivatives of linear functions are *constant*, they need be computed only once – and second derivatives (which are zero) need not be computed at all. For quadratic functions, the first and second derivatives can be computed only once, whereas for general nonlinear functions, these derivatives may have to be computed many times.

A major difference between the Premium Solver Platform and the Premium Solver and Excel Solver is the method used to compute derivatives. As described in the chapter “Analyzing and Solving Models,” the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform can supply fast, accurate derivatives to Solver engines via a process called *automatic differentiation*.

What if your optimization problem requires the use of non-smooth or discontinuous functions? With the Premium Solver Platform, you have several choices. First, for common non-smooth functions such as ABS, MAX and MIN, and even for some IF functions, the nonlinear GRG, Large-Scale GRG and Large-Scale SQP Solvers often yield acceptable results, though you may need to use multistart methods to improve the chances of finding the optimal solution. Second, you can use the Evolutionary Solver (which does not require any derivative values) to find a “good” solution, though you will have to give up guarantees of finding an optimal solution, and it is likely to take considerably more computing time to find a solution. Third, you can use the automatic transformation feature to replace many of these functions with additional variables and linear constraints; if all discontinuous or non-smooth functions in the model are automatically replaced, the problem should be solvable with the nonlinear Solvers, or even with the linear Solvers in some cases. Fourth, you can manually reformulate your model with binary integer variables and associated constraints. You can then use the nonlinear GRG Solver, or even the linear Simplex or LP/Quadratic Solver, in combination with the Branch & Bound

method, to find the *true optimal solution* to your problem. These ideas are explored further in the chapter “Building Large-Scale Models.”

---

## Optimization Problems and Solution Methods

A model in which the objective function and all of the constraints (other than integer constraints) are linear functions of the decision variables is called a *linear programming* (LP) problem. (The term “programming” dates from the 1940s and the discipline of “planning and programming” where these solution methods were first used; it has nothing to do with computer programming.) As noted earlier, a linear programming problem is always **convex**.

If the problem includes integer constraints, it is called an *integer linear programming* problem. A linear programming problem with some “regular” (continuous) decision variables, and some variables that are constrained to integer values, is called a *mixed-integer programming* (MIP) problem. Integer constraints are **non-convex**, and they make the problem far more difficult to solve; see below for details.

A *quadratic programming* (QP) problem is a generalization of a linear programming problem. Its objective is a **convex quadratic** function of the decision variables, and all of its constraints must be *linear* functions of the variables. A problem with linear and convex quadratic constraints, and a linear or convex quadratic objective, is called a *quadratically constrained* (QCP) problem.

A model in which the objective function and all of the constraints (other than integer constraints) are smooth nonlinear functions of the decision variables is called a *nonlinear programming* (NLP) or *nonlinear optimization* problem. If the problem includes integer constraints, it is called an *integer nonlinear programming* problem. A model in which the objective or any of the constraints are non-smooth functions of the variables is called a *non-smooth optimization* (NSP) problem.

### Linear Programming

Linear programming (LP) problems are intrinsically easier to solve than nonlinear (NLP) problems. First, they are convex, where a general nonlinear problem is often non-convex. Second, since all constraints are linear, the globally optimal solution always lies at an “extreme point” or “corner point” where two or more constraints intersect. (In some problems there may be multiple solutions with the same objective value, all lying on a line between two corner points.) This means that an LP Solver needs to consider many fewer points than an NLP Solver, and it is always possible to determine (subject to the limitations of finite precision computer arithmetic) that an LP problem (i) has no feasible solution, (ii) has an unbounded objective, or (iii) has a globally optimal solution.

#### ***Problem Size and Numerical Stability***

Because of their structural simplicity, the main limitations on the size of LP problems that can be solved are time, memory, and the possibility of numerical “instabilities” which are the cumulative result of the small errors intrinsic to finite precision computer arithmetic. The larger the model, the more likely it is that numerical instabilities will be encountered in solving it.

Most large LP models are *sparse* in nature: While they may include thousands of decision variables and constraints, the typical constraint will depend upon only a few of the variables. This means that the Jacobian matrix of partial derivatives of the

problem functions, described earlier, will have many elements that are *zero*. Such sparsity can be exploited to save memory and gain speed in solving the problem.

## ***The Simplex Method***

LP problems are most often solved via the Simplex method. The standard Microsoft Excel Solver uses a straightforward implementation of the Simplex method to solve LP problems, when the Assume Linear Model box is checked in the Solver Options dialog. The Premium Solver uses an improved implementation of the Simplex method with bounds on the variables, the dual Simplex method, and steepest-edge pricing, when the Simplex LP Solver is chosen from the Solver engine dropdown list in the Solver Parameters dialog. The Premium Solver Platform uses a far more sophisticated implementation of the Simplex method which exploits sparsity in the LP model and uses techniques such as presolving, matrix factorization using the LU decomposition, a fast and stable LU update, and dynamic Markowitz refactorization.

The Large-Scale LP/QP Solver engine for the Premium Solver Platform uses an even more powerful implementation of the Simplex method, with performance rivaling the best LP solvers available. It has been tested on LP problems with over two million variables and constraints.

The Large-Scale SQP Solver engine for the Premium Solver Platform includes a powerful linear programming Solver that uses “active set” methods (closely related to the Simplex method). It is practical for problems up to 100,000 variables and constraints. This same Solver engine also handles large-scale QP and NLP problems very efficiently.

The XPRESS Solver engine is Frontline’s fastest and most powerful Solver for linear programming and especially *mixed-integer* linear programming problems. Its ultra-sophisticated primal and dual Simplex and Barrier methods, combined with state-of-the-art Branch and Cut methods for integer problems, yield solutions in record time.

## **Quadratic Programming**

Quadratic programming problems are more complex than LP problems, but simpler than general NLP problems. They have only one feasible region with “flat faces” on its surface (due to their linear constraints), but the optimal solution may be found anywhere within the region or on its surface. Since a QP problem is a special case of an NLP problem, it *can* be solved with the standard nonlinear GRG Solver, but this may take considerably more time than solving an LP of the same size. The Premium Solver Platform’s LP/Quadratic Solver solves QP problems very efficiently, using a variant of the Simplex method to determine the feasible region, and special methods based on the properties of quadratic functions to find the optimal solution.

Most quadratic programming algorithms are specialized to handle only positive definite (or negative definite) quadratics. The LP/Quadratic Solver, however, can also handle semi-definite quadratics; it will find one of the equivalent (globally) optimal solutions – which one depends on the starting values of the decision variables. When applied to an indefinite quadratic objective function, the LP/Quadratic Solver provides only the guarantees of a general nonlinear Solver: It will converge to a locally optimal solution (either a saddle point in the interior, or a locally optimal solution on the constraint surface).

The Large-Scale LP/QP Solver, Large-Scale GRG Solver, Large-Scale SQP Solver, KNITRO Solver, and XPRESS Solver engines can all be used to efficiently solve large QP problems.

## Quadratically Constrained Programming

A problem with linear and convex quadratic *constraints*, and a linear or convex quadratic objective, is called a *quadratically constrained* (QCP) problem. Such a problem is more general than a QP or LP problem, but less general than a convex nonlinear problem. The Simplex-based methods used in the Premium Solver Platform's LP/Quadratic Solver, the Large-Scale LP/QP Solver, and the XPRESS Solver Engine handle only quadratic objectives, not quadratic constraints. But QCP problems – since they are **convex** – can be solved efficiently to global optimality with Barrier methods, also called Interior Point methods.

The Premium Solver Platform's new SOCP Barrier Solver uses a Barrier method to solve LP, QP, and QCP problems. The MOSEK Solver Engine uses an even more powerful Barrier method to solve very large scale LP, QP, and QCP problems, as well as smooth convex nonlinear problems. Both of these Solvers form a logarithmic “barrier function” of the constraints, combine this with the objective, and take a step towards a better point on each major iteration. Unlike the Simplex method, which moves from one corner point to another on the *boundary* of the feasible region, a Barrier method follows a path – called the *central path* – that lies strictly *within* the feasible region.

A Barrier method relies heavily on second derivative information, specifically the Hessian of the Lagrangian (combination of the constraints and objective) to determine its search direction on each major iteration. The ability of the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform to efficiently compute this second derivative information is key to the performance of this method.

## Second Order Cone Programming

Second order cone programming (SOCP) problems are a further generalization of LP, QP, and QCP problems. An SOCP has a linear objective and one or more linear or *second order cone* (SOC) constraints. As explained earlier, a second order cone constraint such as “A1:A5 = conic” specifies that the vector formed by the decision variables A1:A5 must lie within the second-order cone (also called the Lorentz cone) of dimension 5. Algebraically, the constraint specifies that  $a_1^2 \leq a_2^2 + a_3^2 + a_4^2 + a_5^2$ . SOCPs are always **convex**; the Premium Solver Platform's new SOCP Barrier Solver and the MOSEK Solver Engine are both designed to solve SOCP problems, efficiently to global optimality.

Any convex quadratic constraint can be converted into an SOC constraint, with several steps of linear algebra. A convex quadratic objective  $x^T Q x + c x$  can be handled by introducing a new variable  $t$ , making the objective minimize  $t$ , adding a constraint  $x^T Q x + c x \leq t$ , and converting this constraint to SOC form. The SOCP Barrier Solver and the MOSEK Solver Engine both make these transformations automatically; in effect they solve all LP, QP, QCP and SOCP problems in the same way. Second order cone programming can be viewed as the *natural generalization of linear programming*, and is bound to become more popular in the future.

You can also solve an SOCP with the GRG Nonlinear Solver or the Large-Scale GRG, Large-Scale SQP, or KNITRO Solver engines. Although these Solvers do not recognize SOC constraints directly, the Premium Solver Platform will compute values and derivatives for SOC constraints, based on their algebraic form shown above. Hence, these general nonlinear Solvers handle SOC constraints like other general nonlinear constraints. Using these Solvers, you can find optimal solutions for problems containing a mix of linear, general nonlinear, and SOC constraints – bearing in mind that such problems may be non-convex.

## Nonlinear Optimization

As outlined above, nonlinear programming (NLP) problems are intrinsically more difficult to solve than LP, QP, QCP or SOCP problems. They may be **convex** or **non-convex**, and since their second derivatives are not constant, an NLP Solver must compute or approximate the Hessians of the problem functions many times during the course of the optimization. Since a non-convex NLP may have multiple feasible regions and multiple locally optimal points within such regions, there is no simple or fast way to determine with certainty that the problem is infeasible, that the objective function is unbounded, or that an optimal solution is the “global optimum” across all feasible regions. But some NLP problems *are* convex, and many problems include linear or convex quadratic constraints in addition to general nonlinear constraints. Frontline’s field-installable nonlinear Solver engines are each designed to take advantage of NLP problem structure in different ways, to improve performance.

If you use the GRG Nonlinear Solver – the only choice for NLPs in the standard Excel Solver and the Premium Solver – bear in mind that it applies the *same* method to *all* problems, even those that are really LPs or QPs. If you don’t select another Solver engine from the dropdown list box in the Solver Parameters dialog (or, in the standard Microsoft Excel Solver, if you don’t check the Assume Linear Model box in the Solver Options dialog), this Solver will be used – and it may have difficulty with LP or QP problems that could have been solved easily with one of the other Solvers. The Premium Solver Platform can automatically determine the type of problem, and select only the “good” or “best” Solver engine(s) for that problem.

### The GRG Method

The standard Excel Solver, Premium Solver and Premium Solver Platform include a standard nonlinear GRG Solver, which uses the Generalized Reduced Gradient method as implemented in Lasdon and Waren’s GRG2 code. The GRG method can be viewed as a nonlinear extension of the Simplex method, which selects a basis, determines a search direction, and performs a line search on each major iteration – solving systems of nonlinear equations at each step to maintain feasibility. This method and specific implementation have been proven in use over many years as one of the most robust and reliable approaches to solving difficult NLP problems.

As with the Simplex method, the GRG method in the standard Excel Solver uses a “dense” problem representation, and its memory and solution time increases with the number of variables *times* the number of constraints. It is also subject to problems of numerical instability, which may be even more severe than for LP and QP problems. The Large-Scale GRG Solver engine for the Premium Solver Platform uses sparse storage methods and better numerical methods for nonlinear models, such as matrix condition testing and degeneracy handling, to solve much larger NLP problems.

### The SQP Method

The Large-Scale SQP Solver engine uses a Sequential Quadratic Programming (SQP) method to solve nonlinear optimization problems. This method forms and solves a QP subproblem, with a quadratic merit function and linearized constraints, on each major iteration. Because it includes a highly efficient QP solver, a powerful linear programming solver using “active set” methods, and sparsity-exploiting matrix factorization, updating and refactorization methods, the Large-Scale SQP Solver engine is very fast in solving all types of LP, QP and NLP problems. However, the SQP method typically follows a path of *infeasible* trial points until it finds the solution that is both feasible and optimal. Hence, if you stop the Solver before it

reports an optimal solution, the GRG method is far more likely than the SQP method to return a feasible solution as its “best point so far.”

### ***Interior Point and SLQP Methods***

The KNITRO Solver engine uses a Barrier or Interior Point method, specialized for **non-convex** problems, to solve general nonlinear optimization problems. As with the SOCP Barrier and MOSEK Solvers, this method forms a logarithmic “barrier function” of the constraints, combines this with the objective, and takes a step towards a better point on each major iteration. (The actual process of taking a step and the path followed are more complex, because KNITRO assumes that the problem may be non-convex.) The KNITRO Solver uses the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform to efficiently compute second derivative information, but it also has options to work with only first derivative information.

The KNITRO Solver engine also includes a new, high performance Sequential Linear-Quadratic (SLQP) method, which is an “active set” method similar to the SQP method. On highly constrained problems, notably those with equality constraints, this method typically outperforms the Interior Point method. On loosely constrained or unconstrained problems, the Interior Point method can greatly outperform SQP and GRG methods, solving problems much larger than either of these methods. Benchmark studies in the academic literature have demonstrated exceptionally good performance for the KNITRO Solver, on a wide range of test problems.

The GRG, SQP and Interior Point methods are all subject to the intrinsic limitations cited above for nonlinear optimization problems: For smooth **convex** nonlinear problems, they will (subject to the limitations of finite precision computer arithmetic) find the globally optimal solution; but for **non-convex** problems, they can only guarantee a locally optimal solution. To have a reasonable chance – let alone a guarantee – that you will find the globally optimal solution to a non-convex problem, you must use special methods for global optimization.

## **Global Optimization**

The Premium Solver Platform includes powerful tools to help you find the globally optimal solution for a smooth nonlinear **non-convex** problem. These tools include *multistart methods*, which can be used with the nonlinear GRG Solver, the Large-Scale GRG Solver, the Large-Scale SQP Solver, and the KNITRO Solver; a new Interval Global Solver that offers, for the first time, powerful interval methods for global optimization in a commercial software product; and the OptQuest and Evolutionary Solvers, for global solutions of smooth and non-smooth problems.

### ***The Multistart Method***

The basic idea of the multistart method is to automatically run a nonlinear Solver from different starting points, reaching different locally optimal solutions, then select the best of these as the proposed globally optimal solution. Both *clustering* and *topographic search* multistart methods are included in the Premium Solver Platform.

The multistart method operates by generating candidate starting points for the nonlinear Solver (with randomly selected values between the bounds you specify for the variables). These points are then grouped into “clusters” – through a method called *multi-level single linkage* – that are likely to lead to the same locally optimal solution, if used as starting points for the Solver. The nonlinear Solver is then run repeatedly, once from (a representative starting point in) each cluster. The process continues with successively smaller clusters that are increasingly likely to capture

each possible locally optimal solution. A Bayesian test is used to determine whether the process should continue or stop.

For many smooth nonlinear problems, the multistart method has a limited guarantee that it will “converge in probability” to a globally optimal solution. This means that as the number of runs of the nonlinear Solver increases, the probability that the globally optimal solution has been found also increases towards 100%. (To attain convergence for constrained problems, an exact penalty function is used in the process of “clustering” the starting points.) For most nonlinear problems, this method will at least yield very good solutions. As discussed below, the multistart method, like the Evolutionary Solver, is a *nondeterministic* method, which by default may yield different solutions on different runs. (To obtain the *same* solution on each run, you can set a Random Seed option for either of these solution algorithms, as discussed in the chapter “Solver Options.”)

As discussed below, in recent versions of the Premium Solver Platform, the Evolutionary Solver has been enhanced with “filtered local search” methods that offer many of the benefits of multistart methods – making the Evolutionary Solver even more effective for global optimization problems.

The multistart method can be used on smooth nonlinear problems that also contain integer variables and/or “all different” constraints. But this can take a great deal of solution time, since the multistart method is used for each subproblem generated by the Branch & Bound method for integer problems, and it can also impact the Solver’s ability to find feasible integer solutions, as described in the chapter “Diagnosing Solver Results.” If you have many integer variables, or all different constraints, try the Evolutionary Solver as an alternative to the multistart method.

## ***The Interval Branch & Bound Method***

In contrast to the multistart methods and the Evolutionary Solver’s methods, which are *nondeterministic* methods for global optimization that offer no firm guarantees of finding the globally optimal solution, the new Interval Global Solver in the Premium Solver Platform uses a *deterministic* method: An *Interval Branch & Bound* algorithm that will find the globally optimal solution – given enough time, and subject to some limitations related to roundoff error, as discussed in the chapter “Diagnosing Solver Results.”

The Interval Branch & Bound algorithm processes a list of “boxes” that consist of bounded intervals for each decision variable, starting with a single box determined by the bounds that you specify. On each iteration, it seeks lower and upper bounds for the objective and the constraints in a given box that will allow it to discard all or a portion of the box (narrowing the intervals for some of the variables), by proving that the box can contain no feasible solutions, or that it can contain no objective function values better than a known best bound on the globally optimal objective. Boxes that cannot be discarded are subdivided into smaller boxes, and the process is repeated. Eventually, the boxes that remain each enclose a locally optimal solution, and the best of these is chosen as the globally optimal solution.

Several methods are used to obtain good bounds on the values of the objective and constraints within a box or region. *Classic interval methods* rely on the ability of the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform to evaluate Excel functions over intervals and interval gradients. Local constraint propagation methods (also known as *hull consistency* methods) are used to narrow intervals at each stage of evaluation of the problem functions. *Second-order methods* rely on the Interpreter’s ability to compute interval Hessians of Excel functions, and use a variant of the *Interval Newton* method to rapidly minimize function values within a



region. Innovative *linear enclosure* methods – implemented for the first time in the Interval Global Solver – bound each problem function with a linear approximation that can be used in a Simplex method-based test for feasibility and local optimality.

The Interval Global Solver also has a unique ability to find *all real solutions* for a system of nonlinear equations – which can be listed in the new Solutions Report. It can also find an “inner solution” for a system of nonlinear inequalities – a region or “box” (bounds on the variables) within which *all* points satisfy the inequalities. These capabilities are summarized in the chapter “Solver Reports.”

## Non-Smooth Optimization

The most difficult type of optimization problem to solve is a non-smooth problem (NSP). Such a problem may not only have multiple feasible regions and multiple locally optimal points within each region – because some of the functions are non-smooth or even discontinuous, derivative or gradient information generally cannot be used to determine the direction in which the function is increasing (or decreasing). In other words, the situation at one possible solution gives very little information about where to look for a better solution.

In all but the simplest problems, it is impractical to exhaustively enumerate all of the possible solutions and pick the best one, even on a fast computer. Hence, most methods rely on some sort of controlled random search, or sampling of possible solutions – combined with deterministic (non-random) methods for exploring the search space. The Evolutionary Solver, based on genetic algorithms, relies fairly heavily on controlled random search, whereas the OptQuest Solver Engine, based on tabu search and scatter search, relies more heavily on deterministic search methods.

A drawback of these methods is that a solution is “better” only in comparison to other, presently known solutions; both the Evolutionary and OptQuest Solvers normally *have no way to test whether a solution is optimal*. This also means that these methods must use heuristic rules to decide *when to stop*, or else stop after a length of time, or number of iterations or candidate solutions, that you specify.

### Genetic and Evolutionary Algorithms

A non-smooth optimization problem can be attacked – though not often solved to optimality – using a genetic or evolutionary algorithm. (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. (To obtain the *same* solution on each run, you can set a Random Seed option for the Evolutionary Solver, as discussed in the chapter “Solver Options.”)

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 five 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 actually employs multiple variations of four 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.

### **Hybrid Evolutionary and Other Algorithms**

You might imagine that better results could be obtained by combining the strategies used by an evolutionary algorithm with the “classical” optimization methods used by the nonlinear GRG and linear Simplex Solvers. In the Premium Solver and Premium Solver Platform, Frontline Systems has done just that.

The Evolutionary Solver operates as described above, but it also employs classical methods in two situations: First, when the evolutionary algorithm generates a new best point, a local search is conducted to try to improve that point. This step can use a “random local search” method, a gradient-free, deterministic direct search method, a gradient-based quasi-Newton method, or a “linearized local gradient” method. Second, when the evolutionary algorithm generates an infeasible point, the Solver can use “repair methods”, a quasi-Newton method, or even a specialized Simplex method (for subsets of the constraints that are linear) to transform the infeasible point into a feasible one.

In the Premium Solver Platform, the Evolutionary Solver takes maximum advantage of the diagnostic information available from the Polymorphic Spreadsheet Interpreter: It automatically applies genetic algorithm methods to non-smooth variable occurrences (where classical methods cannot be used) and classical methods to smooth and linear variable occurrences. In the local search phase, it can either fix non-smooth variables, or allow them to vary. And it can automatically select the most appropriate local search method, based on linearity and smoothness of the problem functions.

The Evolutionary Solver uses a “distance filter” and a “merit filter” to determine whether to carry out a local search when the genetic algorithm methods find an improved starting point. The “distance filter” plays a role similar to “clustering” in the multistart methods described earlier; both filters contribute to the excellent performance of the Evolutionary Solver on global optimization problems.

The “Achilles heel” of most evolutionary algorithms is their handling of constraints – they are typically unable to handle more than a few inequalities, or any equality constraints at all. In contrast, the hybrid Evolutionary Solver in the Premium Solver and Premium Solver Platform has been able to find good solutions to non-smooth problems with many – even hundreds – of constraints.

### ***Tabu Search and Scatter Search***

The OptQuest Solver Engine for the Premium Solver Platform is based on the principles of tabu search and scatter search. These methods have strong analogies with – and actually predate – genetic algorithm methods, but they rely less heavily on random choice. They work with a population of solutions, which are modified and combined in different ways, then subjected to a selection process. Scatter search methods can sample the space of possible solutions, avoid becoming “trapped” in regions close to local optima, and adaptively diversify or intensify the search. Tabu search uses memory of past search steps to avoid repeated steps and improve future searches. Use of the OptQuest Solver is described in more depth in *Frontline’s Solver Engine User Guide*.

---

## **Integer Programming**

When a Solver model includes integer constraints (for example  $A1:A10 = \text{integer}$ ,  $A1:A10 = \text{binary}$ , or  $A1:A10 = \text{alldifferent}$ ), it is called an *integer programming* problem. Integer constraints effectively make a model **non-convex**, and finding the optimal solution to an integer programming problem is equivalent to solving a global optimization problem. Such problems may require *far* more computing time than the same problem without the integer constraints.

The standard Microsoft Excel Solver uses a basic Branch & Bound method, in conjunction with the linear Simplex or nonlinear GRG Solver, to find optimal solutions to problems involving general integer or binary integer variables. The Premium Solver and Premium Solver Platform use a much more sophisticated Branch & Bound method that is extended to handle alldifferent constraints, and that often greatly speeds up the solution process for problems with integer variables. The Premium Solver Platform’s LP/Quadratic Solver uses improved pseudocost-based branch and variable selection, reduced cost fixing, primal heuristics, cut generation, Dual Simplex and preprocessing and probing methods to greatly speed up the solution of *integer linear* programming problems.

The Evolutionary Solver handles integer constraints, in the same form as the other Solver engines (including alldifferent constraints), but it does not make use of the Branch & Bound method; instead, it generates many trial points and uses “constraint repair” methods to satisfy the integer constraints. (The constraint repair methods include classical methods, genetic algorithm methods, and integer heuristics from the local search literature.) The Evolutionary Solver can often find good solutions to problems with integer constraints, but where the Branch & Bound algorithm can *guarantee* that a solution is optimal or is within a given percentage of the optimal solution, the Evolutionary Solver cannot offer such guarantees.

### **The Branch & Bound Method**

The Branch & Bound method begins by finding the optimal solution to the “relaxation” of the integer problem, ignoring the integer constraints. If it happens that in this solution, the decision variables with integer constraints already have integer

values, then no further work is required. If one or more integer variables have non-integral solutions, the Branch & Bound method chooses one such variable and “branches,” creating two new subproblems where the value of that variable is more tightly constrained. For example, if integer variable A1 has the value 3.45 at the solution, then one subproblem will have the additional constraint  $A1 \leq 3$  and the other subproblem will add the constraint  $A1 \geq 4$ . These subproblems are solved and the process is repeated, “branching” as needed on each of the integer decision variables, until a solution is found where all of the integer variables have integer values (to within a small tolerance).

Hence, the Branch & Bound method may solve many subproblems, *each one a “regular” Solver problem*. The number of subproblems may grow *exponentially*. The “bounding” part of the Branch & Bound method is designed to eliminate sets of subproblems that do not need to be explored because the resulting solutions cannot be better than the solutions already obtained.

## Cut Generation

The Premium Solver Platform's LP/Quadratic Solver, the Large-Scale LP/QP Solver, Large-Scale SQP Solver, MOSEK Solver, and the XPRESS Solver Engine all make use of “cut generation” methods to improve performance on integer linear programming problems. Cut generation derives from so-called “cutting plane” methods that were among the earliest methods applied to integer programming problems, but they combine the advantages of these methods with the Branch & Bound method to yield a highly effective approach, often referred to as a “Branch & Cut” algorithm.

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. Many cuts may be added to a given LP subproblem, and there are many different methods for generating cuts. For example, *Gomory cuts* are generated by examining the reduced costs at an LP solution, while *knapsack cuts*, also known as *lifted cover inequalities*, are generated from constraints involving subsets of the 0-1 integer variables. 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 problems, cut generation enables the overall Branch & Cut algorithm to more quickly discover integer solutions, and eliminate branches that cannot lead to better solutions than the best one already known.

## The Alldifferent Constraint

In the Premium Solver and Premium Solver Platform, a constraint such as  $A1:A5 = \text{alldifferent}$  specifies that the variables A1:A5 must be integers in the range 1 to 5, with each variable different from all the others at the solution. Hence, A1:A5 will contain a *permutation* of the integers from 1 to 5, such as 1,2,3,4,5 or 1,3,5,2,4.

To solve problems involving alldifferent constraints, the Premium Solver products employ an extended Branch & Bound method that handles these constraints as a native type. Whenever variables in an “alldifferent group” have non-integral solution values, or integral values that are not all different, the Branch & Bound method chooses one such variable and “branches,” creating two new subproblems where the value of that variable is more tightly constrained.

The nonlinear GRG Solver bundled with the Premium Solver and Premium Solver Platform, and the Large-Scale GRG Solver, Large-Scale SQP Solver, KNITRO

Solver, and MOSEK Solver engines use this extended Branch & Bound method to solve problems with integer and alldifferent constraints.

The Large-Scale LP/QP Solver and the XPRESS Solver use their own Branch & Cut methods. They transform alldifferent constraints into equivalent sets of binary integer variables and additional linear constraints, then apply their preprocessing, probing and cut generation methods to these variables and constraints.

The Evolutionary Solver uses methods from the genetic algorithm literature to handle alldifferent constraints as permutations, including several mutation operators that preserve the “alldifferent property,” and several crossover operators that generate a “child” permutation from “parents” that are also permutations.

Since Solver engines use quite different methods to handle the alldifferent constraint, you ll want to try a variety of Solver engines to see which one performs best on your model. This is especially true if your model uses smooth nonlinear or – even better – linear functions aside from the alldifferent constraint.

---

## Simulation Optimization

With the Premium Solver Platform and Risk Solver Engine, you can define and solve *simulation optimization* problems that seek good or optimal decisions, where the parameters of the problem are not fixed numbers, but are uncertain values.

To introduce simulation optimization, we must first explain how you can use Risk Solver Engine to build a model that incorporates uncertainty, and use Monte Carlo simulation to analyze the effects of that uncertainty.

Spreadsheets have traditionally allowed you to change numbers, asking „what if questions, and instantly see the results. But many real-world problems involve so much uncertainty, about so many different factors, that it is impractical to explore all the possibilities, even with the aid of a spreadsheet. Risk analysis, using Monte Carlo simulation, offers an automated way to explore the possibilities, and get a much better idea of the range of possible outcomes.

### Uncertain Variables

In any problem, there are factors or inputs that you, as a decision-maker, can control – for example, the price you set for a product. There are other factors or inputs that you cannot control – for example, customer demand, competitor actions, interest rates, etc. In a quantitative model, it is useful to distinguish between these two kinds of factors: You can use **decision variables** to represent inputs over which you have direct control, and **uncertain variables** (called *random variables* in mathematics) to represent inputs beyond your control.

The values of uncertain variables can be drawn from a probability distribution. For example, you might expect that future short-term interest rates will fall between 3% and 6%, with all possibilities equally likely within this range. You d model this with a function call such as =PsiUniform(.03, .06) which specifies a uniform distribution over uncertain values. Or you might expect that customer demand for shirt sizes will be clustered around “average” sizes, and use a function call such as =PsiNormal(15, 2) which specifies a Normal distribution with mean 15 and standard deviation 2.

Risk Solver Engine provides more than 40 PSI Distribution functions – from PsiBernoulli() to PsiWeibull(). You can also draw values from predefined *Certified Distributions*, by simply referring to these distributions in PsiSip() and PsiSlurp() functions. You can easily shift, truncate and correlate probability distributions with

PSI Property functions such as PsiShift(), PsiTruncate() and PsiCorrMatrix(). For more information on PSI Distribution functions and Certified Distributions, please see the Risk Solver Engine User Guide.

## Uncertain Functions

You will also have outputs or results of interest – such as Net Profit – that you can compute, using formulas that depend on the factors influencing the problem – possibly both decision variables and uncertain variables. We'll use the term **uncertain functions** for quantities whose calculation depends on uncertain variables (in mathematics these are called *functions of random variables*).

On request, Risk Solver Engine will perform a **Monte Carlo simulation**, which involves some number (say 1000, that you specify) of Monte Carlo **trials**. On each trial, **sample** values are drawn randomly for the uncertain variables, and the uncertain functions are calculated from their formulas using these input values, yielding sample values for the uncertain functions. In effect, a Monte Carlo simulation performs a „what-if“ analysis for 1000 or more sample values, and accumulates the results.

## Statistics for Uncertain Functions

While you can access the results of individual trials, you will usually be interested in summary statistics for all the values taken by the uncertain functions. For example, you might want to know the mean, minimum and maximum Net Profit over all the trials. You can access these summary statistics by calling functions such as PsiMean(), PsiMin(), and PsiMax() and passing the cell address for Net Profit.

Risk Solver Engine provides 20 PSI Statistics functions – from PsiAbsDev() to PsiVariance(), including functions to obtain percentiles, frequency bins, and “quantile” measures such as Value at Risk. You'll find complete descriptions of the PSI Statistics functions in the Risk Solver Engine User Guide.

## Using Simulation Results in Optimization

Once you have cell formulas using PSI Statistics functions like the ones just mentioned, you can use these cells as your objective (Set Cell) or as the left hand sides of constraints. For example, if you have a worksheet that defines uncertain market demand for your products, and computes sales, inventory levels and Net Profit, you could define and solve an optimization problem that seeks to maximize Net Profit subject to constraints on maximum (say, 90<sup>th</sup> percentile) or minimum (10<sup>th</sup> percentile) inventory levels. This is an example *simulation optimization* problem.

To solve such a problem, the Premium Solver Platform runs a Solver engine that performs a search, where it will generate a series of Trial Solutions – supplying new values for all of the decision variables. On each Solver iteration or Trial Solution, Risk Solver Engine performs a Monte Carlo simulation consisting of 1000 or more trials, and accumulates statistics across all these trials. On each Trial Solution, the objective and constraint values seen by the Solver are computed from summary statistics of the Monte Carlo trials.

## Speed and Vectorized Evaluation

When performed with traditional tools in Excel, simulation optimization is very slow, because the problem might require hundreds or even thousands of Trial Solutions, and each Trial Solution requires a new simulation and 1000 or more Monte Carlo

trials, performed by spreadsheet recalculations. But the combination of the Premium Solver Platform and Risk Solver Engine is up to 100 times faster than this approach.

Risk Solver Engine uses a special version of the Polymorphic Spreadsheet Interpreter – the same PSI Technology used to enhance speed and accuracy in the Premium Solver Platform. Where the PSI Interpreter employs special methods to evaluate your spreadsheet formulas to compute gradients needed by Solver engines, in Risk Solver Engine it employs special “vectorized” evaluation methods to compute results for Monte Carlo trials.

Beyond the computing time required for a Monte Carlo simulation on each Trial Solution, a simulation optimization problem is often a difficult nonlinear or non-smooth optimization problem in its own right. But the Premium Solver Platform includes Solver engines to handle such problems – from the standard GRG Nonlinear and Evolutionary Solvers to a variety of plug-in, large-scale Solver engines.

Thanks to advanced technology, you can use the combination of the Premium Solver Platform and Risk Solver Engine to find optimal solutions to simulation optimization problems, in a reasonable amount of time, that are far larger and more challenging than you could solve with traditional tools in Excel. And if you need even more speed, you have two more alternatives: (i) Move your model to Frontline's Solver Platform SDK, where your problem functions can be evaluated at the speed of compiled code, or (ii) wait for future versions of the Premium Solver Platform and Risk Solver Engine, where deeper integration will yield even greater speed in Excel.





# Building Solver Models

---

## Introduction

This chapter takes you step by step through the process of setting up and solving a simple linear programming model in Excel, using the Premium Solver products. It then describes in depth the features of the Solver Parameters dialog that you can use to define the essential elements of your Solver model: decision variables, constraints, and the objective function.

Our step-by-step example is a “quick and dirty” setup that can be used to solve the example problem, but is not well documented or easy to maintain. Microsoft Excel has many features that can help you organize and display the structure of your model, through tools such as defined names, formatting and outlining. As models become larger, the problems of managing data for constraints, coefficients, and so on become more significant, and a properly organized spreadsheet model can help manage this complexity. Hence, the last section in this chapter provides hints for designing a model that is understandable, maintainable and scalable.

---

## From Algebra to Spreadsheets

Optimization problems are often described in algebraic terms. In this section, we'll show how you can translate from the algebraic statement of a problem to a spreadsheet model that the Solver can optimize.

### Setting Up a Model

To set up an optimization model as a Microsoft Excel spreadsheet, you will follow these essential steps:

1. Reserve a cell to hold the value of each decision variable.
2. Pick a cell to represent the objective function, and enter a formula that calculates the objective function value in this cell.
3. Pick other cells and use them to enter the formulas that calculate the left hand sides of the constraints.
4. The constraint right hand sides can be entered as numbers in other cells, or entered directly in the Solver's Add Constraint dialog box.

Within this overall structure, you have a great deal of flexibility in how you lay out and format the cells that represent variables and constraints, and which formulas and built-in functions you use. For example, the formulas needed for a linear programming problem can always be specified with the SUMPRODUCT function, or with the DOTPRODUCT add-in function included with the Premium Solver products. If the model is easily expressed in vector-matrix algebraic notation, you may want to use defined names for the vectors and built-in functions such as MMULT to compute the constraint left hand sides.

## A Sample Linear Programming Model

Consider the following LP problem, a variation on the “Product Mix” worksheet in the SOLVSAMP.XLS workbook included with Microsoft Excel. Our factory is building three products: TV sets, stereos and speakers. Each product is assembled from parts in inventory, and there are five types of parts: chassis, picture tubes, speaker cones, power supplies and electronics units. Our goal is to produce the mix of products that will maximize profits, given the inventory of parts on hand.

### *The Algebraic Form*

This problem can be described in algebraic form as follows. The decision variables are the number of products of each type to build:  $x_1$  for TV sets,  $x_2$  for stereos and  $x_3$  for speakers. There is a fixed profit per unit for each product, so the objective function (the quantity we want to maximize) is:

Maximize  $75x_1 + 50x_2 + 35x_3$  (Profit)

Building each product requires a certain number of parts of each type. For example, TV sets and stereos each require one chassis, but speakers don’t use one. The number of parts used depends on the mix of products built (the left hand side of each constraint), and we have a limited number of parts of each type on hand (the corresponding constraint right hand side):

Subject to  $1x_1 + 1x_2 + 0x_3 \leq 400$  (Chassis)  
 $1x_1 + 0x_2 + 0x_3 \leq 200$  (Picture tubes)  
 $2x_1 + 2x_2 + 1x_3 \leq 800$  (Speaker cones)  
 $1x_1 + 1x_2 + 0x_3 \leq 400$  (Power supplies)  
 $2x_1 + 1x_2 + 1x_3 \leq 600$  (Electronics)

Since the number of products built must be nonnegative, we also have the constraints  $x_1, x_2, x_3 \geq 0$ .

### *The Spreadsheet Formulas*

The fastest (though not necessarily the best) way to lay out this problem on the spreadsheet is to pick (for example) cell A1 for  $x_1$ , cell A2 for  $x_2$  and cell A3 for  $x_3$ . Then the objective might be entered in cell A4 as the formula:

$=75*A1+50*A2+35*A3$

We’d go on to enter a formula in (say) cell B1 for the first constraint left hand side (Chassis), such as  $=1*A1+1*A2+0*A3$ , or perhaps the equivalent  $=A1+A2$ .

Similarly, we’d use cell B2 for the formula  $=A1$  (Picture tubes), B3 for the formula  $=2*A1+2*A2+A3$  (Speaker cones), B4 for the formula  $=A1+A2$  (Power supplies), and B5 for the formula  $=2*A1+A2+A3$  (Electronics).

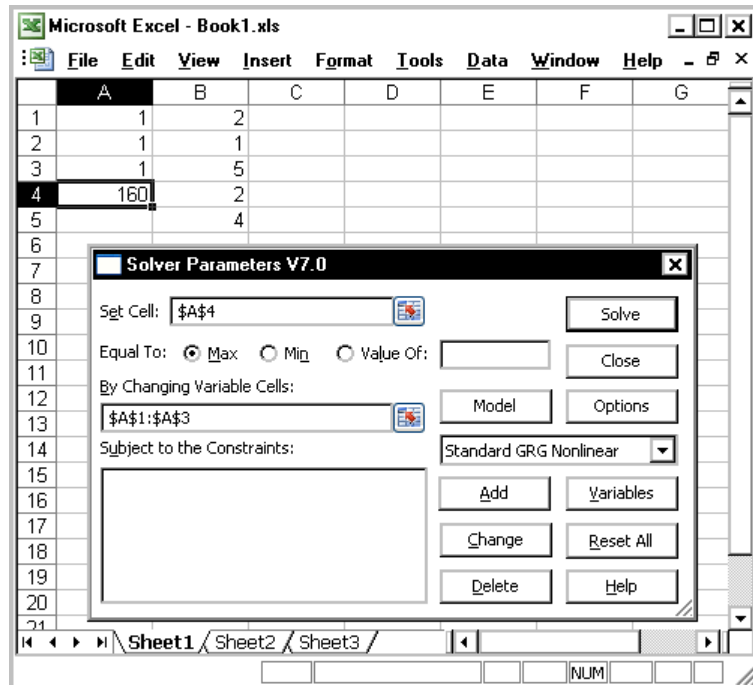
We now have a simple spreadsheet model, with which we can practice “what if.” For any values we enter for the decision variables in cells A1, A2 and A3, the objective (Total Profit) and the corresponding values of the constraint left hand sides (the

numbers of parts used) will be calculated. Note that this simple example is *not* in the form required for the Premium Solver's *fast problem setup*; in a later section of this chapter, you'll see how this model can be formulated using the SUMPRODUCT function, which is both easier to read and appropriate for fast problem setup.

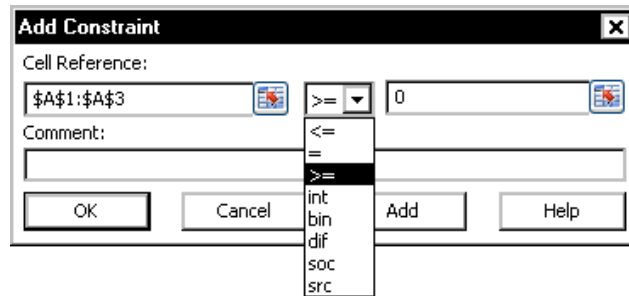
## The Solver Dialogs

To prepare the model for optimization, we will use the Solver Parameters dialog to point out to the Solver (i) the cells that we've reserved for the decision variables, (ii) the cell that calculates the value of the objective function, and (iii) the cells that calculate the constraint left hand sides. We'll also enter values for the constraint right hand sides, and non-negativity constraints on the variables.

The simplest way to proceed is to select cell A4 (the objective function), then choose the Premium Solver... command from the Tools menu. The Solver Parameters dialog will appear, with the current cell (A4, the correct one) suggested as the entry in the Set Cell box. The default choice of Max is also correct for this problem. To select the decision variables or Changing Cells, click in the Changing Cells edit box and type A1:A3, or use the mouse to select cells A1 to A3 on the spreadsheet. At this point your spreadsheet and Solver Parameters dialog should look like this:

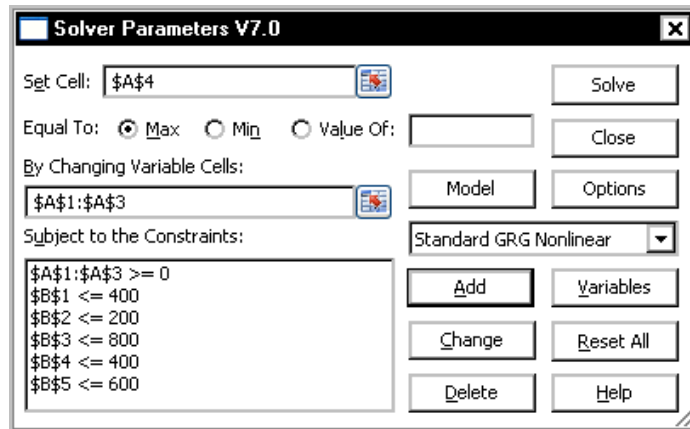


The next step is to enter the constraints, including the non-negativity constraint on the decision variables. To enter the non-negativity constraint, click on the Add button to bring up the Add Constraints dialog. The input focus is on the Cell Reference edit box, so you can either type A1:A3 or use the mouse to select cells A1 to A3 on the spreadsheet. Next, click on the Constraint (right hand side) edit box and enter 0 there. Finally, click on the down arrow next to Relation to display a list of relation symbols, and select  $\geq$  from the list. Your Add Constraint dialog box should look like the one on the next page.



To accept the non-negativity constraint and continue with entry of another constraint, click the Add button. The Add Constraint dialog box will reappear with the edit fields blank. With the input focus on Cell Reference, click on cell B1, the first constraint left hand side. Then click on the Constraint edit box, and enter 400 there. The default relation  $\leq$  is correct for this constraint, so you are now ready to click the Add button to accept this constraint.

Continue with entry of the remaining four constraints in a similar manner. When you have entered the Cell Reference (B5) and Constraint value (600) for the last constraint, click the OK button instead of the Add button. The Solver Parameters dialog will reappear, and the constraints you have entered should appear in the Constraints list box, as shown below:



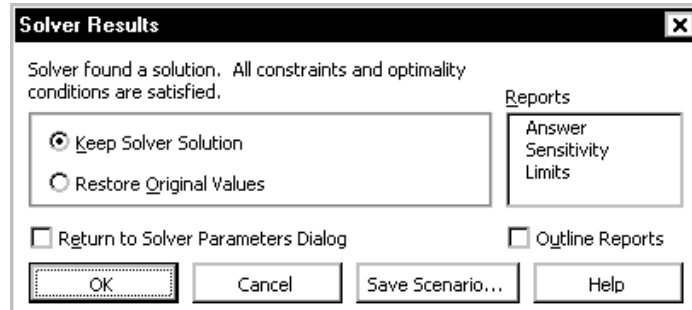
## Selecting the Solver Engine

In the standard Microsoft Excel Solver, your choices for the Solver engine to optimize this model are the nonlinear GRG Solver (the default choice) or the linear Simplex Solver. Since this is a linear programming problem, it can be solved more efficiently with the Simplex Solver. To use this Solver engine, you click on the Options button to bring up the Solver options dialog, and click on the "Assume Linear Model" check box, then click OK.

In the Premium Solver products, selecting the Solver engine is easier: There is a dropdown list of Solver engines available in the main Solver Parameters dialog. The default choice is the standard GRG Solver (which, while slower, is fully capable of solving linear as well as nonlinear problems). To choose the standard Simplex LP or LP/Quadratic Solver, click on the down arrow symbol to display the list of Solver engines, and click on the Solver engine of your choice.

## Solving the Problem

To find the optimal solution for this LP model, click on the Solve button. After an instant or two, the solution values ( $A1 = 200$ ,  $A2 = 200$ ,  $A3 = 0$ ) should appear in the cells for the decision variables, and the Solver Results dialog should appear, as shown below.



Here you have several choices: You can select one or more reports to be produced from the Reports list box (and check the Outline Reports box, if you'd like the reports outlined); you can save the solution values as a named scenario, for display later with the Excel Scenario Manager; and you can either discard the solution values (restoring the original cell values) or save the solution values in the decision variable cells. The reports are described in more detail in the chapter “Solver Reports” later in this Guide. For now, click on OK to save the optimal solution in the decision variable cells. You'll then return to worksheet Ready mode, unless you checked the box “Return to Solver Parameters Dialog,” in which case the Solver Parameters dialog would reappear, ready to solve another problem.

Congratulations – You've set up and solved a simple but complete Solver problem! The next sections will go into much greater depth on the choices available to you in the Solver Parameters dialog.

---

## Decision Variables and Constraints

You have a great deal of flexibility in how you specify the decision variables and the constraints in the Solver dialogs. In the previous section, we used the simplest forms: the decision variables were all adjacent cells in one column, and the constraint right hand sides were constants. In this section, we'll cover more general forms of specifying both variables and constraints.

### Variables and Multiple Selections

In the standard Excel Solver, the Solver Parameters dialog provides just one Changing Cells edit box to specify all of the decision variables in a model. This edit box accepts only cell selections, which may be typed in as cell coordinates (or as defined names equivalent to cell coordinates), or entered by clicking with the mouse on the desired cells in the spreadsheet. However, you can use this box to enter the most general form of cell selection permitted by Microsoft Excel, called a *multiple selection*. A multiple selection consists of one or more individual selections, separated by commas (when English is chosen in the Regional Settings – you may be using different settings). Each individual selection may be a single cell, a column or row of cells, or a rectangular set of (contiguous or adjacent) cells. An example of a multiple selection from the “Maximizing Income” sheet in the SOLV SAM P.XLS workbook included with Microsoft Excel is shown on the next page.

	A	B	C	D	E	F	G	H	I
1	<b>Example 4: Working Capital Management.</b>								
2	Determine how to invest excess cash in 1-month, 3-month and 6-month CDs so as to maximize interest income while meeting company cash requirements (plus safety margin).								
3									
4									
5		<i>Yield</i>	<i>Term</i>			<i>Purchase CDs in months:</i>			
6	1-mo CDs:	1.0%	1			1, 2, 3, 4, 5 and 6		<i>Interest</i>	
7	3-mo CDs:	4.0%	3			1 and 4		<i>Earned:</i>	
8	6-mo CDs:	9.0%	6			1		<i>Total</i>	<b>\$7,700</b>
9									
10	<i>Month:</i>	<i>Month 1</i>	<i>Month 2</i>	<i>Month 3</i>	<i>Month 4</i>	<i>Month 5</i>	<i>Month 6</i>	<i>End</i>	
11	<i>Init Cash:</i>	\$400,000	\$205,000	\$216,000	\$237,000	\$158,400	\$109,400	\$125,400	
12	<i>Matur CDs:</i>		100,000	100,000	110,000	100,000	100,000	120,000	
13	<i>Interest:</i>		1,000	1,000	1,400	1,000	1,000	2,300	
14	<i>1-mo CDs:</i>	100,000	100,000	100,000	100,000	100,000	100,000		
15	<i>3-mo CDs:</i>	10,000			10,000				
16	<i>6-mo CDs:</i>	10,000							
17	<i>Cash Uses:</i>	75,000	(10,000)	(20,000)	80,000	50,000	(15,000)	60,000	
18	<i>End Cash:</i>	\$205,000	\$216,000	\$237,000	\$158,400	\$109,400	\$125,400	\$187,700	
19									
20		-290,000							
21									

The decision variables in this problem are the amounts to invest in 1-month CDs, 3-month CDs and 6-month CDs. We have opportunities to invest in 1-month CDs every month, but 3-month CDs are available only in Month 1 and Month 4, and 6-month CDs are available only in Month 1. To enter all of these cells as decision variables, we need a multiple selection: It must consist of at least three individual selections, separated by commas. Note that although all of the cells to be selected “touch” each other, they cannot be selected as one rectangular area. We could select these cells in several different ways: For example, as (B14:G14,B15:B16,E15) or as (B14:B16,C14:G14,E15). If you display the Solver Parameters dialog for this example, you will see that the Changing Cells edit box uses another selection, (B14:G14,B15,E15,B16). All of these selections are equivalent as far as the Solver is concerned. (Note: The parentheses are not needed when entering a multiple selection in the Changing Cells edit box, but they *would* be needed when entering a multiple selection as an argument in a function such as DOTPRODUCT.)

In general, the areas of a multiple selection must be rectangular, but they need not “touch” each other as they did in the example above. You should avoid entering overlapping areas in a multiple selection: For example, Excel will allow you to enter the above selection as (B14:B16,C14:G14,E14:E15), but the duplication of variable cells will slow down the Solver during problem setup and reporting, and may yield results different from the ones you expected.

There are several ways to enter a multiple selection in a formula or a dialog box: (i) You can simply type the cell coordinates, entering each rectangular area in the form *FromCell : ToCell*, separating the areas by commas (or other language-specific separators); (ii) you can select each area by clicking with the mouse, typing a comma between each mouse selection; or (iii) you can make the entire multiple selection with the mouse by pressing the CTRL key as you make the first selection, and holding it down until you have selected all of the rectangular areas.

If you enter the individual selections by clicking with the mouse, you will notice that the cell reference is entered in “absolute” coordinates, such as \$B\$14:\$G\$14. Further, you will find that regardless of whether you include the dollar signs when you type in the cell coordinates, the cell reference is treated as absolute, and it appears with the dollar signs the next time you display the dialog. “Relative” cell references have significance when you copy a formula from one cell to another, but in the Solver dialogs all cell references are absolute.

## Using the Range Selector

The Premium Solver Platform has a convenience feature for selecting cells with the mouse, called the Range Selector. With the Range Selector, you can temporarily “hide” the dialog box where the cell selection will be entered, so that you can more easily see and move about on the worksheet itself. This is most useful for the Set Cell and Changing Cells edit boxes in the Solver Parameters dialog, since that dialog can cover large portions of the viewable worksheet area; but the Range Selector is available in all of the cell selection edit boxes in the standard Solver and the Premium Solver products.

You activate the Range Selector by clicking the small rectangular button at the right edge of the cell selection edit box, as shown, for example, in the Solver Parameters dialog just before “Selecting the Solver Engine.” This causes the dialog to be hidden and a cell cursor (“thick cross”) to appear. The text form of the currently selected range is shown just below and to the right of the cell cursor, as illustrated below:

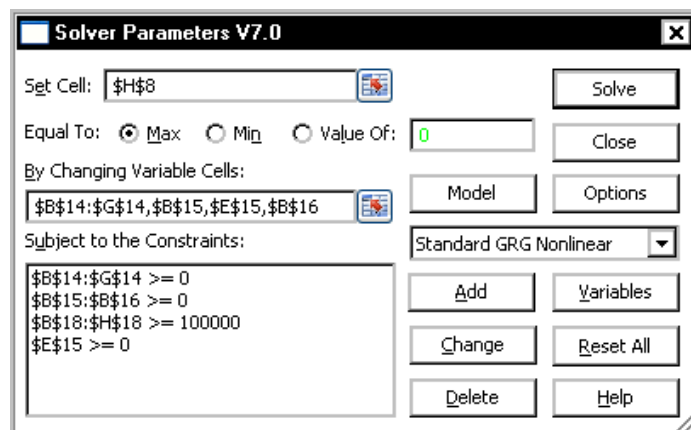
	A	B	C
1	1	2	
2	1	1	
3	1	5	
4	160	\$A\$1:\$A\$3	
5		4	

You can now select whatever cell range you want by clicking and dragging with the mouse. When you release the mouse button, the original dialog will reappear, with the cell selection in the appropriate edit box.

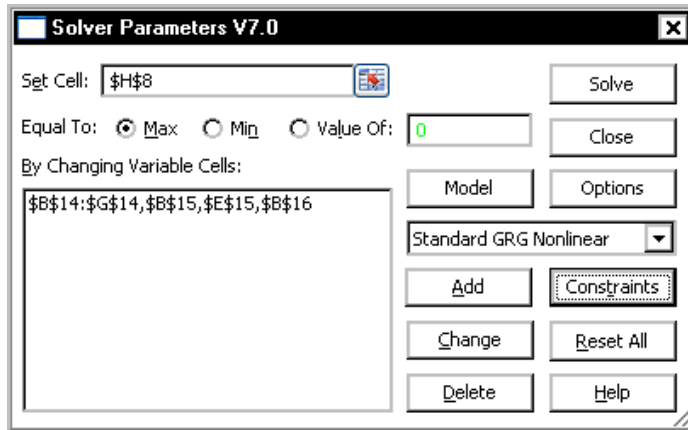
## Using the Variables Button

If your decision variable cells are scattered across your worksheet (or workbook), selecting all of them in a single multiple selection can become difficult and tedious. In most cases you can avoid this problem by organizing your model so that the decision variable cells occupy a small number of contiguous blocks – as outlined in “More Readable Models” later in this chapter. But if this is inconvenient, the Premium Solver products allow you to switch to a more flexible method of specifying decision variables, using the Variables button.

In the “Maximizing Income” example shown on the previous page, the standard display of the Solver Parameters dialog box looks like the one below:



If you click on the Variables button, the dialog box changes to show the variable selections in a list box, as shown below. The Variables button is now labeled Constraints, and clicking on it returns you to the original display.



You can now use the Add, Change and Delete buttons to add, modify or remove variable cells, just as you would for constraints. Each row in the Variable Cells list box can contain a multiple selection; however, we recommend that you use a simple selection for clarity, since there is no limit on the number of rows in the list box. If you click on the Add button for example, a dialog box like the one below appears:



One factor to bear in mind about use of the Variables button is that a Solver model where you have used more than one row in the Variable Cells list box cannot be moved to a computer where the standard Excel Solver is being used, even if the total number of variables is less than the standard Solver's limit of 200. (The same thing is true for models that depend on other extensions in the Premium Solver products.) But as your Solver models become larger and more complex, you will probably find that the alternate display of the Variable Cells list box will become more useful.

## Constraint Left and Right Hand Sides

In specifying the constraints for the sample LP model earlier in this chapter, we entered the constraint left hand sides as single cell references, and the right hand sides as constants in the Add Constraint dialog. But the Solver permits more general forms for both the left and right hand sides of constraints.

The constraint *left* hand side, entered in the Cell Reference edit box, may be any *individual* selection, such as a column, row, or rectangular area of cells. Multiple selections are not permitted here. In the example shown earlier, we could have placed the constants 400, 200, 800, 400, 600 in cells C1:C5, then entered all five constraint left hand sides at once as B1:B5, and the five right hand sides as C1:C5.



## Overlapping and Conflicting Constraints

You should be careful about entering equivalent or overlapping cell references in the left hand sides of different rows of constraints in the Constraints list box. The only situation where this makes sense is when one constraint uses the  $\geq$  relation to specify a lower bound, and the other uses the  $\leq$  relation to specify an upper bound. (Of course, the lower bound must be less than the upper bound, or else there will be no feasible solution to the problem.) If you place multiple lower or upper bounds on the same cells, the Premium Solver products will use the “tighter” bounds. For example, if you enter constraints such as  $A1:A5 \leq 10$  and  $A3:C3 \leq 5$ , you’ve specified both  $A3 \leq 10$  and  $A3 \leq 5$ , so the Solver will use  $A3 \leq 5$ .

A pair of constraints such as  $A1:A5 \leq 10$  and  $A1:A5 \geq 10$  has the same effect as  $A1:A5 = 10$ , but is considerably less efficient and may cause problems for some of the Solver’s advanced solution strategies. Hence, you should always use the form with the  $=$  relation in the constraint.

If you specify both a  $\leq$  or  $\geq$  constraint and a binary integer or alldifferent constraint for the same group of variable cells, the Premium Solver products will display an error message when you try to solve the problem, unless the bounds you specify agree with the binary integer or alldifferent constraint. For example,  $A1:A5 \geq 3$  and either  $A1:A5 = \text{binary}$  or  $A1:A5 = \text{alldifferent}$  will cause the Solver to display an error message. (The values of variables in an “alldifferent group” must vary from 1 to N, where N is the number of variables; of course, you can always use formulas in other cells to shift this range of values to another range.) As a convenience, you *can* specify a  $\leq$  or  $\geq$  constraint for a binary integer variable that further restricts it to be either 0 or 1, without getting an error message. “Fixing” variables in this way can be useful when experimenting with an integer programming model.

## Constraint Right Hand Sides

The constraint *right* hand side may be any of the following:

1. A numeric constant such as 1.
2. A cell reference such as C1.
3. An (individual) selection such as C1:C5.
4. An arbitrary formula such as  $C1+1$  or  $C2/D2$ .
5. “integer”, “binary” or “alldifferent”
6. “conic” (Premium Solver Platform only)

Option 5 is for integer constraints only and is discussed below under “Using Integer Constraints.” Option 6 is available only in the Premium Solver Platform, and is discussed below under “Using Conic Constraints.” If you use option 3 – a selection of more than one cell – the number of cells selected must match the number of cells you selected for the constraint left hand side. (The two selections need not have the same shape: For example, the left hand side could be a column and the right hand side a row.) You may also use rectangular areas of cells. In any case, when you use this form you are specifying several constraints at once, and the constraint left hand sides correspond element-by-element to the right hand sides. As we noted in the example shown earlier, you can enter the right hand side values 400, 200, 800, 400 and 600 into cells C1 to C5, and enter a single constraint such as  $B1:B5 \leq C1:C5$ . You can see examples of this form in nearly all of the sample worksheets included with the Solver, as well as throughout this Guide. It is by far the most useful form.

If the constraint right hand side is a cell reference, cell selection or formula, the Solver needs to know whether the contents of those cells, or the value of the formula is *constant* in the problem, or *variable* (i.e. dependent on the values of the decision variables). If the right hand side depends on any of the decision variables, the Solver transforms a constraint such as “L H S  $\geq$  R H S” into “L H S - R H S  $\geq$  0” internally. All Solver engines work internally with constant bounds on the constraint functions.

### ***Implicit Non-Negativity Constraints***

Many Solver problems – and perhaps *most* LP problems – have “non-negativity” constraints, or lower bounds of zero on the decision variables. To save you the trouble of entering these constraints explicitly in the Constraints list box, both the standard Solver and the Premium Solver products provide an Assume Non-Negative check box in the Solver Options dialog. When this box is checked, all variables that do not have explicit lower bounds in the Constraint list box are automatically given lower bounds of zero. You can enter constraints such as  $A1 \geq 2$  or  $A1 \geq -3$  for certain variables, overriding the implicit lower bound, and use the Assume Non-Negative box to give all other variables zero lower bounds.

### ***Efficiency of Constraint Forms***

The Solver recognizes the case where the constraint left hand side is a decision variable, or a set of decision variables. As long as the corresponding right hand sides are constant (i.e. not dependent on any of the variables), these constraints are specially treated as bounds on the variables. The most common instance of a bound on a variable is a non-negativity constraint such as  $A1 \geq 0$ , but any sort of constant bounds are handled efficiently by all of the Solver engines.

There is no difference in terms of efficiency between a constraint entered (for example) as  $A1 \leq 100$  or as  $A1 \leq B1$  where B1 contains 100; the Solver recognizes that B1 is equivalent to a constant. The form  $A1 \leq B1$  is usually better from the standpoint of maintainability of your Solver model.

On the other hand, a constraint right hand side that is a formula – even a simple one like  $2+2$  – will incrementally increase the solution time for the model. The Solver treats *any* such formula as a RHS potentially dependent on the variables, and it internally creates a constraint “L H S - R H S  $\geq$  0” – even if the formula really was a constant bound on a variable. It is better to place whatever formula you need into a cell, and reference that cell as the constraint right hand side: Because the formula has already been analyzed by Microsoft Excel when it was entered in the cell, the Solver can determine whether it is dependent on the variables.

### ***Using Integer and Alldifferent Constraints***

Integer constraints can only be applied to cells that are decision variables; hence the cells selected on the left hand side of the constraint must be a subset (or all) of the cells in the Changing Cells edit box, or the Variable Cell list box. Integer constraints specify that the selected variable cells must have solution values that are integers or whole numbers, such as -1, 0 or 2, to within a small tolerance. Variable cells that have *binary* integer constraints must be either 0 or 1 at the optimal solution. Variable cells subject to an *alldifferent* constraint must have values from 1 to N, where N is the number of cells specified on the constraint left hand side, and each cell must have a value different from all the others.

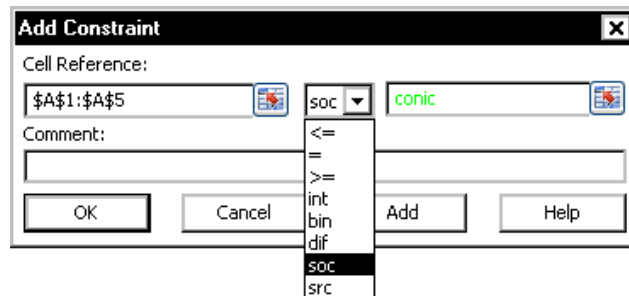
You specify an integer, binary or alldifferent constraint by selecting the “int”, “bin” or “dif” choice from the Relation dropdown list in the Add/Change Constraint dialog.

The Solver displays such constraints in the Constraint list box in the form “A1:A5 = integer,” “A1:A5 = binary” or “A1:A5 = alldifferent”.

Be sure that you select “int,” “bin” or “dif” from the Relation dropdown list. If you select “=” from the dropdown list and *type* the word “integer,” “binary” or “alldifferent” on the right hand side, the Solver will not recognize this as an integer constraint, and clicking on Solve will probably result in the error message “Solver encountered an error value in a target or constraint cell”.

## Using Conic Constraints

Conic constraints are a new feature of the Premium Solver Platform, discussed in the previous chapter “Solver Models and Optimization.” They can only be applied to cells that are decision variables; hence the cells selected on the left hand side of the constraint must be a subset (or all) of the cells in the Changing Cells edit box, or the Variable Cell list box. To add a conic constraint, you select either the “soc” (second order cone) or “src” (rotated second order cone) choice from the relation dropdown list, as shown below.



This constraint specifies that the vector formed by  $n$  decision variables must belong to the second order cone of dimension  $n$ . A “soc” constraint is equivalent to the formula  $A1 \geq \text{SQRT}(\text{SUMSQ}(A2:A5))$  – in linear algebra,  $a_1^2 \leq a_2^2 + a_3^2 + a_4^2 + a_5^2$  – or if  $A1$  is non-negative,  $A1^2 \geq \text{SUMSQ}(A2:A5)$ . A “src” constraint is equivalent to the formula  $2*A1*A2 \geq \text{SUMSQ}(A3:A5)$  – in linear algebra  $2a_1a_2 \leq a_3^2 + a_4^2 + a_5^2$ .

## More Readable Models

This section focuses on features of the Solver and Microsoft Excel you can use to build more readable, maintainable, scalable models. The approach outlined above in “From Algebra to Spreadsheets” is the “quick and dirty” way to translate from a model in algebraic form to an equivalent spreadsheet model, ready for optimization. However, that approach will soon prove to be short-sighted when you wish to change the data (for example unit profits or parts on hand), expand the model to include more products or parts, or show the model to someone unfamiliar with the problem or uncomfortable with algebraic notation.

For a better approach to laying out this model, consider the EXAMPLE1 worksheet in the Examples.xls workbook, shown on the next page, which is copied to the installation folder (typically C:\Program Files\Frontline Systems\Premium Solver Platform\Examples) during installation.

	A	B	C	D	E	F	G
1	<b>Example 1: Product mix problem.</b>						
2	Your company manufactures TVs, stereos and speakers, using a common parts						
3	inventory of power supplies, speaker cones, etc. Parts are in limited supply and you						
4	must determine the most profitable mix of products to build. This version of the						
5	model is built with only the cell coordinates and SUMPRODUCT in formulas.						
6							
7							
8				<i>TV set</i>	<i>Stereo</i>	<i>Speaker</i>	
9		<i>Number to Build</i>		100	100	100	
10	<i>Part Name</i>	<i>Inventory</i>	<i>No. Used</i>				
11	<i>Chassis</i>	450	200	1	1	0	
12	<i>Picture Tube</i>	250	100	1	0	0	
13	<i>Speaker Cone</i>	800	500	2	2	1	
14	<i>Power Supply</i>	450	200	1	1	0	
15	<i>Electronics</i>	600	400	2	1	1	
16				<i>Profits:</i>			
17				<i>By Product</i>	\$75	\$50	\$35
18				<b>Total</b>	<b>\$16,000</b>		
19							

You are encouraged to open this worksheet in Microsoft Excel and examine its formulas, row, column and cell formatting, and use of labels. If you are not familiar with Excel's "Format Cells" tabbed dialog box, this is a good opportunity to see how it works. Just select one or more cells in EXAMPLE1, choose Format Cells and the appropriate tab to see how the fonts, patterns and borders have been set.

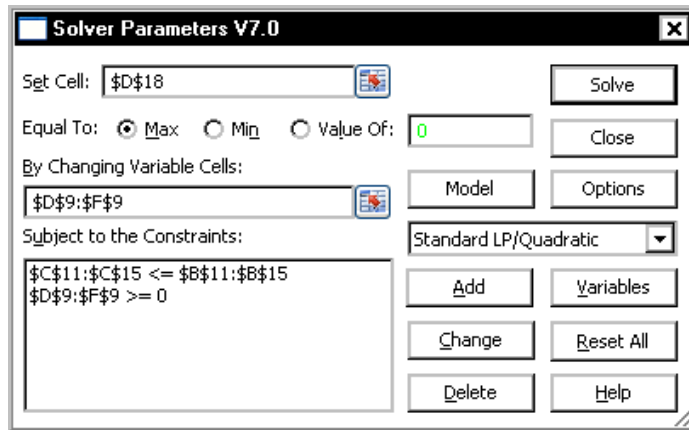
## Layout and Formatting

EXAMPLE1 shows one way (not the only way!) to set up an LP model in a more readable and maintainable fashion. To enhance readability, borders and labels have been used to draw attention to the decision variables at D9 to F9, the constraint left hand side formulas at C11 to C15, and the right hand sides at B11 to B15. Your client or management won't miss the objective function calculation at B17 to F18.

EXAMPLE1 is also much easier to maintain and expand than a model constructed with "hardwired" formulas as outlined in the previous chapter. The parts required for each product and the unit profit per product built (i.e. the coefficients of this model) are laid out in cells on the spreadsheet. To add products, you can simply insert new columns in the range of columns D through F; the constraint formulas will "expand" automatically. To add more parts, you can insert new rows between rows 11 and 15, then copy any one of the existing formulas in column C into the new rows.

The SUMPRODUCT function is used in EXAMPLE1 to calculate the value of the objective function and the constraint left hand sides. In each instance you could have used DOTPRODUCT instead. DOTPRODUCT would be preferred in sparse models, since it permits multiple selections of cells for both of its arguments.

If you choose Premium Solver... from the Tools menu, the Solver Parameters dialog for EXAMPLE1 will appear, as shown on the next page.



This dialog illustrates a simple case of the definition of blocks of constraints at one time. There are five constraints of the form C11 <= B11, C12 <= B12, etc., but they can be entered all at once in the Constraints list box. If you haven't previously tried this, click the Delete button to remove the first line in the Constraints list box, then re-enter it with the following steps:

1. Click on Add... to bring up the Add Constraints dialog.
2. With the Cell Reference field ready for input, use the mouse to select all five cells C11 through C15.
3. Click on the Constraint field (the default <= relation is OK) and use the mouse to select all five cells B11 through B15.
4. Click on OK to add this block of information.

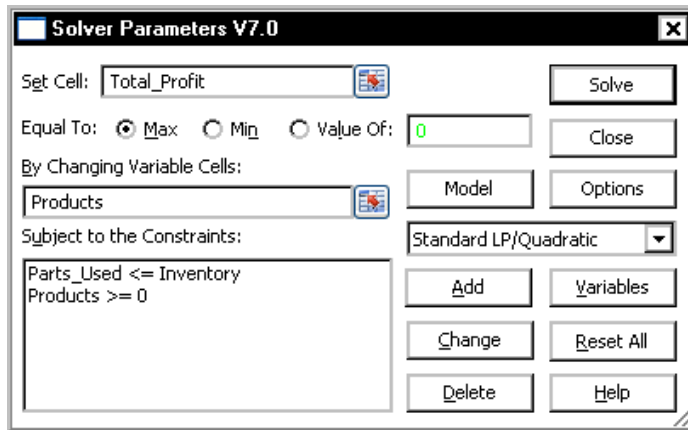
## Using Defined Names

On the worksheet and in the Solver Parameters dialog in EXAMPLE1, the decision variables and the constraint left-hand and right-hand sides were referred to by their cell coordinates. For example, the number of TV sets is at D9, and the number of products built as a whole is D9:F9. You can make your spreadsheets more readable and more flexible by using defined names instead of such cell coordinates.

A defined name is created by selecting the Insert Name Define... menu command and entering the name and the range of cells it should refer to. For example, you could define the name Products to refer to D9:F9, and then type "Products" in the Changing Cells field, or the Cell Reference field of the Add Constraint dialog.

The Insert Name Create... menu command in Excel provides a shortcut way to define a group of names at once. For example, in EXAMPLE1 you could select the range A11 to B15, choose Insert Name Create... and click OK to create the names Chassis for B11, Picture\_Tube for B12, Speaker\_Cone for B13, Power\_Supply for B14 and Electronics for B15.

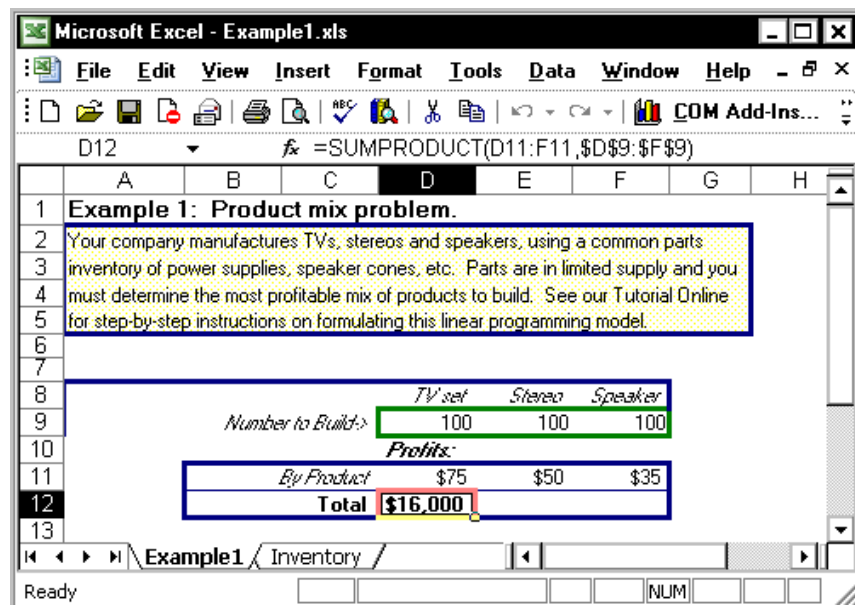
But you may find it more useful to define names for *blocks* of cells – for example, "Inventory" for cells B11:B15, and "Parts\_Used" for cells C11:C15. As you add defined names, the Solver recognizes them, and uses them in preference to cell coordinates in the Set Cell, By Changing Cells, and Constraints boxes. After defining "Total\_Profit" for cell D18 and "Products", "Inventory" and "Parts\_Used" in EXAMPLE1, you can select Tools Premium Solver... and display a much more readable Product Mix optimization model, as shown on the next page.



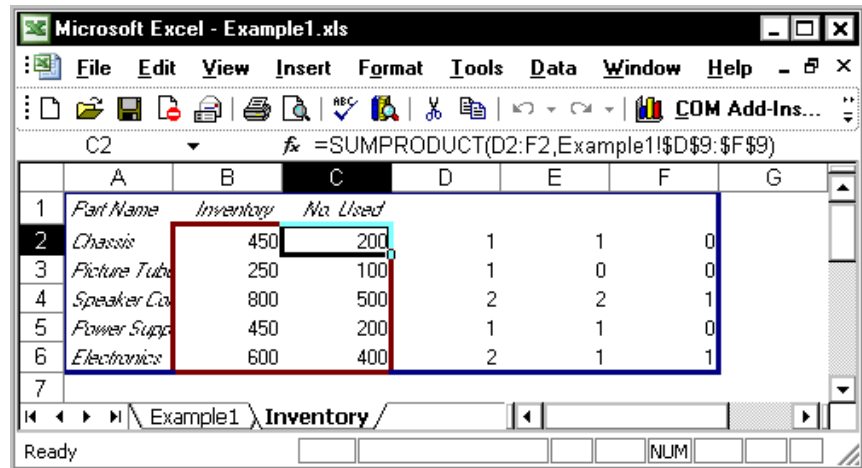
If you take the time to organize and lay out your model in “block form,” use defined names for both individual cells and groups of cells, and make effective use of cell borders, colors and other formatting, you will find that it is easier to maintain your model, and to communicate your results to coworkers, clients and management.

## Models Defined Across Multiple Worksheets

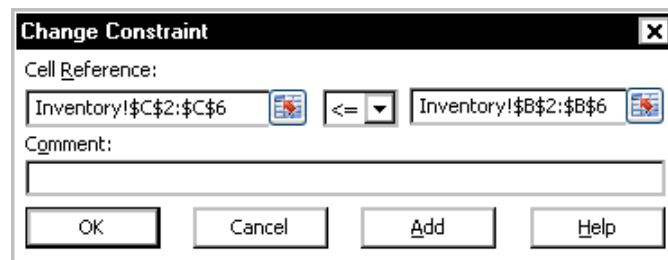
The Premium Solver requires that cells containing decision variables and constraint left hand sides are on the active worksheet. But the Premium Solver Platform allows you to define decision variables and constraint left hand sides on any worksheet of a workbook. For example, we can split the EXAMPLE1 model illustrated in the previous section into two parts:



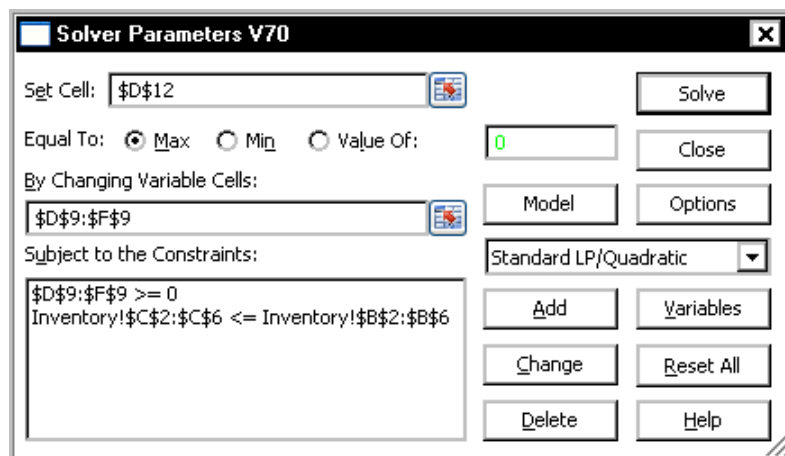
Worksheet Example1 contains the decision variables, and the coefficients and SUMPRODUCT formula for the objective. Worksheet Inventory contains the constraint left hand sides (formulas), coefficients and right hand sides.



Note that the constraint formula refers to the variables as Example1!\$D9:\$F9. Starting from worksheet Example1, we can display the Solver Parameters dialog, click Add or Change, then simply point and click to select the constraint left and right hand sides on worksheet Inventory, as shown below.



The resulting Solver Parameters dialog looks like this:



## Multiple Models and Multiple Worksheets

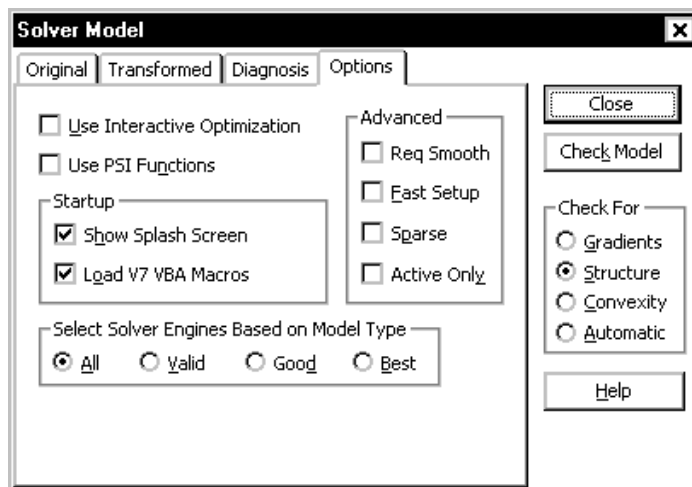
You could define another, different Solver model on worksheet Inventory, and this model could refer to variable and constraint cells on worksheets Example1 and Inventory. The Premium Solver Platform keeps track of all the models.

By default, there is one model per worksheet, and the variable, constraint and objective cell selections for this model are saved “behind the scenes.” The model displayed in the Solver Parameters dialog is the one for the worksheet that is active when you select Tools Premium Solver. But you can create additional named models if you use PSI functions for optimization, as explained in the next section.

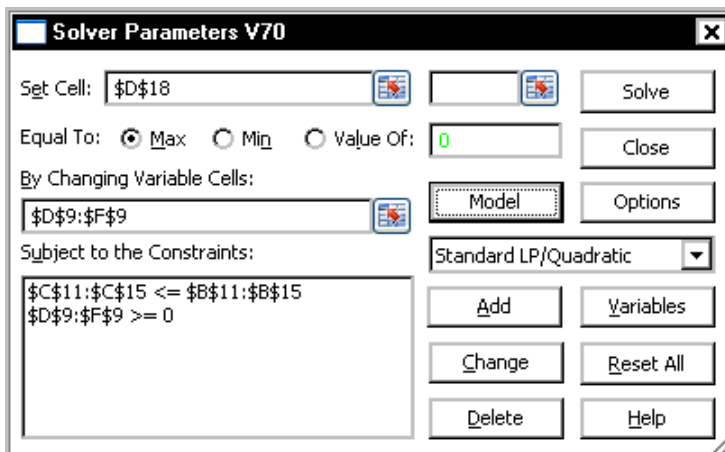
## Defining Your Model with PSI Functions

The Premium Solver Platform V7.0 introduces new PSI worksheet functions for specifying the elements of your optimization model. PSI functions provide an optional, alternative style for defining variables, constraints and the objective. They are especially useful if you are creating a simulation optimization model, where you use PSI functions to specify the probability distributions of uncertain variables (such as **PsiNormal()**), and summary statistics for uncertain functions (such as **PsiMean()**).

The PSI worksheet functions are implemented by the main Solver program file Solver32.xll, and they are always listed in Excel's Function Wizard. But they are *recognized and used* to define a Solver model only if you check the box for the option **Use PSI Functions** on the Options tab in the Solver Model dialog:



When this box is checked, the Solver Parameters dialog appears with a new edit box and range selector to the right of the Set Cell edit box, as shown below.





In this new edit box, you can type or select a cell address (for example **\$H\$12**) where a formula (for example **=Psi0 bj (\$D \$18, "M ax")**) will be automatically written. This is an example of the new function-based style for defining models.

If you click the Add or Change button when the **Use PSI Functions** option is selected, the Add or Change Constraint dialog will appear with one new edit box and range selector, as shown below.

In the new edit box, you can type or select a cell address (for example **\$H\$11**) where a formula (for example **=PsiCon(\$C\$11:\$C\$15, "<=", \$B\$11:\$B\$15)**) will be written. Similarly, you could change the second constraint in the list box so that in cell **\$H\$10** a formula such as **=PsiCon(\$D\$9:\$F\$9, ">=", "0")** appears.

PSI functions can be used for decision variables in much the same way as for constraints. If you click the Variables button and then click the Add or Change button, the Add or Change Variable dialog will appear:

In the new edit box, you can type or select a cell address (for example **\$H\$9**) where a formula (for example **=PsiVar(\$D\$9:\$F\$9)**) will be automatically written.

## Function-Based Style for Models

Use of the PSI functions illustrated above is entirely optional. If you don't check the box **Use PSI Functions** in the Solver Model dialog Options tab, the edit boxes for PSI functions won't appear in the dialogs, and any PSI functions for optimization on the worksheet will be ignored.

PSI functions provide a way for you to see your cell selections for variables, constraints, and the objective on the worksheet, without displaying the Solver Parameters dialog. For example, after creating PSI functions as described in the previous section, if you select Tools Options, click the View tab, and check the **Formulas box** in the Window options group, the PSI function formulas will be visible in cells H9:H12 on the worksheet, as shown on the next page. The last argument in each function call is the comment (currently empty) associated with this variable or function block.

	H
9	=PsiVar(\$D\$9:\$F\$9, "")
10	=PsiCon(\$D\$9:\$F\$9, ">=", "0", "")
11	=PsiCon(\$C\$11:\$C\$15, "<=", \$B\$11:\$B\$15, "")
12	=PsiObj(\$D\$18, "Max", 0, "")

As noted earlier, PSI functions are especially useful if you're creating a simulation optimization model, where you use PSI functions to specify the probability distributions of uncertain variables (such as **PsiNormal()**), and summary statistics for uncertain functions (such as **PsiMean()**).

## PSI Functions and Interactive Dialogs

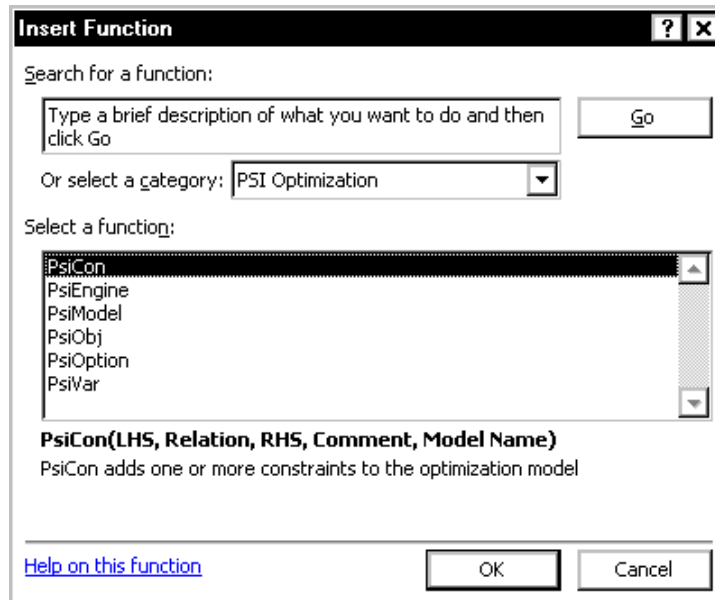
You can move easily between the interactive dialogs and PSI functions when creating your model. (This is in marked contrast to other Excel add-ins for simulation and optimization that support both functions on the worksheet and interactive dialogs, but do not synchronize the two.)

When you add or change a block of variables, constraints, or the objective through the Solver Parameters dialogs and specify cells for PSI functions, the Solver writes the PSI function calls into these cells automatically.

When you type a formula using a function such as **PsiVar**, **PsiCon** or **PsiObj** directly into a worksheet cell in Excel worksheet Ready mode, the next time you display the Solver Parameters dialog, the new function will be recognized and the variables, constraints, or objective specified will also appear in the dialog.

## Using the Insert Function Dialog

If you select menu choice Insert Function... in Excel 2000-2003, or the Function Wizard on the Formulas tab in Excel 2007, you'll see that the new PSI functions are listed. They are grouped into their own category PSI Optimization, as shown below.



If you are using Risk Solver Engine, you will see three other categories of PSI functions in this dialog: PSI Distributions, PSI Property Functions, and PSI Statistics.

The role of the Model Name argument of each PSI function is explained below under “PSI Functions and Multiple Models.” The Comment is a character string comment. Below is a summary of all of the PSI functions for optimization:

**PsiVar (Range, Comment, Model Name)** – Adds one or more decision variables to the model. The **Range** argument should be a contiguous range of decision variable cells.

**PsiCon (LHS, Relation, RHS, Comment, Model Name)** – Adds one or more constraints to the model. The **LHS** argument should be a contiguous range of constraint left hand side cells (containing formulas). The **Relation** argument may be any of the strings “<=”, “=”, “>=”, “int”, “bin”, “dif”, “soc” or “src” (same as the relation dropdown in the Add/Change Constraint dialog). The **RHS** argument may be a number, a character string that contains a valid Excel formula, or a contiguous range of cells. In the last case, the number of cells in the LHS and RHS cell ranges must match.

**PsiObj (Target Cell, Sense, Value Of, Comment, Model Name)** – Adds the objective to the model. The **Target Cell** argument should be the address of a single cell. The **Sense** argument may be “Max”, “Min” or “ValueOf” (case doesn’t matter here). The **Value Of** argument corresponds to the Value Of edit box in the Solver Parameters dialog. As explained in the Premium Solver Platform User Guide, we recommend that you use only “Max” or “Min”.

**PsiEngine (Engine Name, Model Name)** – Specifies the Solver engine that should be used to solve the model. The **Engine Name** argument should be a string such as “Standard LP/Quadratic” that matches one of the names in the Engine dropdown list of the Solver Parameters dialog.

**PsiModel (Option Name, Value, Model Name)** – Specifies a value for an option that appears in the Solver Model dialog. The **Option Name** should be a character string that matches one of the argument names of the SolverModel VBA function; the **Value** should be valid for that argument.

**PsiOption (Option Name, Value, Model Name)** – Specifies a value for an option that appears in the Solver Options dialog. The **Option Name** should be a character string that matches an argument name of one of the SolverXXXOptions VBA functions; the **Value** should be valid for that argument.

## PSI Functions and Multiple Models

As mentioned above, the Premium Solver Platform V7.0 supports multiple Solver models in the same workbook, or even on the same worksheet. The Platform maintains a “current Solver model” for each worksheet, and the variable, constraint and objective cell selections, Solver engine selection, and option settings for this model are saved “behind the scenes.”

If you use PSI functions to define variable, constraint and objective cell selections, these selections are assumed to belong to the model on the same worksheet that contains the formulas with PSI function calls, unless you include an explicit **model name** argument in your PSI function calls.

The model name can be the same as a worksheet name in the workbook – in this case, the PSI function is treated as belonging to the model on the given worksheet – or it may be a different name. If a different name is used, all of the PSI function calls that

include this model name as an argument are treated as one model. You can access this model via the **Load Model** button in the Solver Options dialog, which will replace the current worksheet model with the new set of variable, constraint and objective cell selections, Solver engine selection, and option settings.

For more information on the Load Model and Save Model buttons, see “Loading, Saving and Merging Solver Models” in the chapter “Solver Options.”

# Analyzing and Solving Models

---

## Introduction

This chapter explains how to use the Solver Model dialog, which is unique to the Premium Solver Platform, to analyze and transform your model, and control the solution process. The Solver Model dialog is the user interface to features of the new Polymorphic Spreadsheet Interpreter in the Premium Solver Platform, described briefly in the “Introduction.”

**Note:** If you are using the Premium Solver, the Solver Model dialog contains only the options Use Interactive Optimization, Use PSI Functions, Show Splash Screen, and Load V7 VBA Macros, described in the section “Options Tab: Selecting Model Features.” Other features in this chapter are not available in the Premium Solver.

The first sections of this chapter explain how to use the Solver Model dialog to diagnose your model's type (LP, QP, NLP, etc.), sparsity, and convexity; identify “problem formulas” that make your model non-linear, non-smooth or non-convex; automatically *transform* your model to replace uses of certain non-smooth functions with smooth and even linear counterparts; and set options for use of the Interpreter when you solve your model. The last section describes in greater depth how the Interpreter works, in comparison to the Microsoft Excel formula recalcuator, and how certain Solver engines take advantage of the Interpreter's greater capabilities.

---

## Using the Solver Model Dialog

You use the Solver Model dialog to analyze and optionally transform your model, but not to solve it. Through this dialog, you can run the Polymorphic Spreadsheet Interpreter on your model, without running any Solver engines. You can also set options to determine whether and how the Interpreter will be used when you *do* solve your model, by clicking the Solve button in the Solver Parameters dialog.

We can illustrate this process with the EXAMPLE5 worksheet, a simple Inventory Planning model included in the Examples.xls workbook, installed with the Solver files, shown on the next page. You can easily open this workbook from the Solver Parameters dialog by clicking Help, then clicking Examples.

EXAMPLE5 was originally designed to be a linear programming model with a few integer variables – but to properly minimize holding costs, the objective at cell B19 had to depend on I14, J14 and K14, which are sums of IF functions at I11:K14 in the “Help function box.” If you attempt to solve this model with the LP/Q quadratic

Solver, you'll receive the message "The linearity conditions required by this Solver engine are not satisfied." What can we do to improve this situation?

**Inventory Planning Model: Automatic Transformation of IF Functions**

This inventory planning model was originally designed for linear programming, but to properly minimize holding costs, the objective at cell B19 had to depend on I14, J14 and K14, which are sums of IF functions at I11:K14 in the "Help function box". Using the LP/Quadratic Solver yields the result "The linearity conditions ... are not satisfied." The IF functions have turned an otherwise simple linear mixed-integer model into difficult a **non-smooth** model. What can we do to find an optimal solution?

$X = \text{product 1}$ ,  $Y = \text{product 2}$ ,  $Z = \text{product 3}$

	X	Y	Z
Period 1	0	0	0
Period 2	0	0	0
Period 3	0	0	0
Sum	0	0	0

$I = \text{Inventory (product X, Y, Z)}$

	X	Y	Z
I1	0	Y1	-50
I2	-20	Y2	-50
I3	-50	Y3	-100

**Objective:** -2200

**Constraints:**

		>=	
0		>=	0
0		>=	20
-20		>=	50
0		>=	50
-50		>=	0
-50		>=	20
0		>=	0
0		>=	100
-100		>=	100
0		<=	150
0		<=	150
0		<=	150

With the Premium Solver Platform, you have several choices! You can use **Tools Solver...**, select **Evolutionary Solver**, and click **Solve**. Given enough time, the Solver will usually find a minimum cost of 2400. But you can use the Model dialog to **diagnose** and **transform** the model automatically into one that is easier to solve. Use **Tools Solver...** and click **Model**. Select **Structure** and click **Check Model**. The Model is **NLP** with 9 variables and 13 functions; 12 of the functions are linear, but one (the objective) is nonlinear. Now click on the **Transformed** tab, and click **Check Model** again. The Transformed problem is **LP Convex**, with 27 variables and 67 functions - the extra variables and functions were created internally by the Solver. Check the box **Solve Transformed Problem**, and click **Close**. Select the **LP/Quadratic Solver** and click **Solve**. The transformed model is now solved to optimality as an LP/MIP model, again yielding the objective 2400.

Pictured below is the Solver Parameters dialog for this model. Notice that we've used the mouse to resize this dialog so you can see all of the variable cells and all of the rows in the Constraints list box.

**Solver Parameters V7.0**

Set Cell:

Equal To: ☐ Max ☒ Min ☐ Value Of:

By Changing Variable Cells:

Subject to the Constraints:

- 
- 
- 
- 
- 

Buttons: Solve, Close, Model, Options, Add, Variables, Change, Reset All, Delete, Help

Clicking the **Model** button in this dialog will display the Solver Model dialog with diagnostic information for the current model, as shown on the next page. Clicking the **Solve** button will run the currently selected Solver engine on the current model, using the current settings in the Solver engine's options dialog, and the current settings for the Polymorphic Spreadsheet Interpreter in the Solver Model dialog.

Unknown	Variables	Functions	NonZeroes
All	9	13	
Smooth			
Quadratic			
Linear			
Bounds	18	Sparsity %	
Integers	9	Total Cells	

Solve With: ☒ PSI Interpreter ☐ Excel Interpreter

Check For: ☐ Gradients ☐ Structure ☐ Convexity ☒ Automatic

The **Original** tab of the Solver Model dialog displays statistics about the model, as you originally defined it, on the current Excel worksheet. The **Transformed** tab displays the same statistics for the “transformed” model, where the Interpreter has replaced certain non-smooth functions (if any) with new variables and constraints. The **Diagnosis** tab is used to set options for analyzing your model and reporting exceptions. The **Options** tab is used to set Interpreter options that affect the process of analyzing, transforming, and solving your model.

## Original Tab: Analyzing Model Structure

Initially, the Solver Model dialog shows only the numbers of variables, functions (including the objective and constraints), bounds on variables, and integer variables in this model. We can get more information by selecting the Check For Structure option, and clicking the Check Model button – yielding the dialog shown below. (See “Using the Check Model Button” below for a complete discussion of the analysis performed for the Check For Gradients, Structure, and Convexity options.)

NLP	Variables	Functions	NonZeroes
All	9	13	36
Smooth	9	13	36
Quadratic	0	0	0
Linear	0	12	27
Bounds	18	Sparsity %	30.77
Integers	9	Total Cells	52

Solve With: ☒ PSI Interpreter ☐ Excel Interpreter

Check For: ☐ Gradients ☒ Structure ☐ Convexity ☐ Automatic

The upper left corner of the Solver Model dialog displays **NLP**, the Interpreter's diagnosis of the type of model on the EXAMPLE5 worksheet. Recall that this model was originally intended to be an LP (linear programming) model. But the IF functions at I11:K14 are not linear functions. (They are actually non-smooth – but by default, Interpreter treats IF functions as smooth nonlinear and computes gradients at

each Trial Solution for them ; see “Using Analyzer Advanced Options” below .) In the section “Analyzing Model Exceptions” below , we ll use the Interpreter to pinpoint these IF functions – which could be hard to find in a large Solver model. But first we ll review the displayed statistics, options and buttons in this dialog.

## Using Model Statistics

The columns of the Solver Model dialog contain counts of the **Variables**, **Functions**, and **NonZeroes** in your model. The rows labeled **All**, **Smooth**, **Quadratic**, and **Linear** will display – respectively – (i) the *total* number of variables, functions, and nonzeros (dependencies) in the model; (ii) the number of *smooth* variables, functions, and dependencies, (iii) the number of *quadratic* variables, functions, and dependencies, and (iv) the number of *linear* variables, functions, and dependencies in the model. The Bounds box displays the total number of bounds on variables, and the Integers box shows the total number of integer, binary, and alldifferent variables in your model. The Sparsity % and Total Cells boxes help measure the total size and sparsity of your model, as further discussed below.

### Types of Functions and Variables

For an explanation of linear, quadratic, and smooth functions, please consult the section “Functions of the Variables” in the chapter “Solver Models and Optimization.” *All functions* includes both non-smooth and smooth functions; *smooth functions* includes all smooth nonlinear, quadratic and linear functions. However, *quadratic functions* and *linear functions* include only functions of those types. This means that the number of smooth functions shown is always at least the sum of the number of quadratic functions and linear functions.

A decision variable is a “linear variable” if, everywhere that it occurs in formulas of your model, the expression where it occurs (taken alone) would be a linear function. A variable is counted as a “quadratic variable” if, everywhere that it occurs in formulas of your model, the expression where it occurs (taken alone) would be a quadratic function. A variable is counted as a “smooth variable” if, everywhere that it occurs, the expression (taken alone) would be a smooth nonlinear, quadratic or linear function. For example, if your model had only an objective defined by the formula,  $=3*A1 + A2^2 + INT(A3)$ , variable A1 would be counted as a linear variable, since the expression  $3*A1$  taken alone is a linear function; variable A2 would be counted as a quadratic variable; both A1 and A2 would be counted as smooth variables; and variable A3 would appear only in the count of all variables.

### Types of Dependencies and NonZeroes

A “nonzero” is counted each time that a given function is found to depend on a given variable. For example, if cell B1 is the objective function, it contains  $=A1+A2-B2$ , cell B2 contains  $=A2*A3$ , and cells A1:A3 are all variables, this function would contribute 3 nonzeros to the total count. (Note that a variable, such as A2 in this example, is counted only once per function, even if it is referenced more than once in the function’s cell formulas.) Since B1 *depends on* cells A1:A3, the corresponding *partial derivatives* (elements of the Jacobian matrix, as discussed in the chapter “Solver Models and Optimization) will be *nonzero*.

The Total Cells box contains a count of all of the cells in your model (the „top-level cells for the objective and constraints, and all cells that they reference). The Sparsity % box contains the results of calculating  $[All\ NonZeroes] / ([All\ Variables] * [All\ Functions])$ , expressed as a percentage. Model sparsity was mentioned briefly in the chapters “Introduction” and “Solver Models and Optimization,” but here we can



describe it more precisely: A *dense* model will have a high Sparsity % and a *sparse* model will have a low Sparsity % figure. The EXAMPLE5 model has a sparsity of 30.77% -- an intermediate figure.

As mentioned in the earlier chapters, large optimization models tend to be sparse in nature. Often, a linear programming (LP) model of over 10,000 variables will have a Sparsity % figure of as little as 2% or 3%. Frontline's Large-Scale LP/QP, Large-Scale GRG, Large-Scale SQP, KNITRO and XPRESS Solvers are designed to exploit sparsity in a model to save memory and solution time, and improve accuracy.

## Using the Check Model Button

The three radio buttons in the “Check For” option group determine how much analysis the Interpreter will carry out for your model when you click the **Check Model** button, and when you click the **Solve** button in the main Solver Parameters dialog. Clicking this button on the **Original** tab analyzes the original model; clicking this button on the **Transformed** tab analyzes the transformed model.

Selecting **Gradients** causes the Interpreter to scan all of the formulas in your model, to determine whether it can compute values and gradients for all of these formulas. If you've used Excel features that the Interpreter does not support – such as circular references, some references to other workbooks, and some user-defined functions – an error message dialog will appear.

Selecting **Structure** causes the Interpreter to perform the same analysis as Gradients (which may yield an error message), then analyze the structure (dependencies) in your model, fill in the model statistics described above, and classify your model as an LP, QP, QCP, SOCP, NLP, or NSP. (See the chapter “Solver Models and Optimization” for an explanation of these abbreviations.)

Selecting **Convexity** causes the Interpreter to perform the same analysis as Structure, then seek to determine whether each function in your model is convex or non-convex over the feasible region, as described below. The overall result is displayed in the upper left corner of the Solver Model dialog.

The Check Model button always counts all of the formula cells in your model, and displays this total in the Total Cells box in the Solver Model dialog. If you select Structure or Convexity, the Sparsity % box is also filled, as described above.

## Analyzing Model Convexity

An innovation in the Premium Solver Platform, not available in other modeling systems, is automatic testing of problem functions for convexity. As mentioned briefly in the Introduction, this test may yield conclusive results (that the problem is either **convex** or **non-convex**) or inconclusive results (meaning that the test was not able to prove convexity, nor was it able to prove non-convexity.) A convexity test that always yielded conclusive results would take time that grew exponentially with the number of variables, and hence would be impractical for even modest-size models. The methods used in the Premium Solver Platform are designed to yield useful results in many, but not all cases, while taking a “reasonable” amount of time. The methods used to analyze model convexity are further described later in this chapter, in the section “More on the Polymorphic Spreadsheet Interpreter.”

As explained in the Introduction, an optimization model is convex only if *all* of its functions are convex (if the objective is being *maximized* rather than minimized, then this function must be *concave* rather than convex). The overall convexity result for the model is displayed in the upper left corner of the Solver Model dialog following

the result of structure diagnosis (LP, QP, QCP, etc.) as “Convex,” “Nonconvex,” or blank if the convexity test is inconclusive. You can also obtain a report of the convexity test results for each problem function, as explained below.

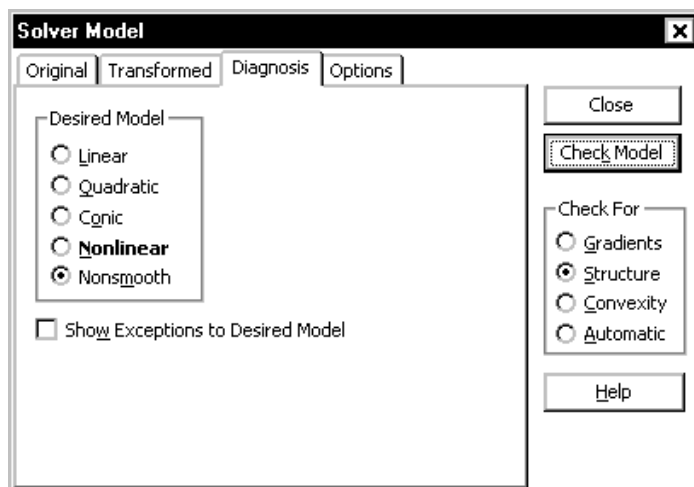
---

## Diagnosis Tab: Analyzing Model Exceptions

In EXAMPLE5, we intended to create a linear programming (LP) model, but when we clicked the Check Model button, the Interpreter reported that the model is nonlinear (NLP). In a model that we intended to be smooth nonlinear, which we hoped to solve using (say) the Large-Scale GRG or KNITRO Solver, the Interpreter might report that the model is non-smooth (NSP). How do we find and correct the problem in a large Solver model? The Solver Model dialog can tell you which cell formulas contributed to a diagnosis of a model type different from what you intended.

### The Structure Report

To find out why EXAMPLE5 was diagnosed as NLP rather than LP, click the Diagnosis tab in the Solver Model dialog, which will display the options shown below.



The **Desired Model** option group and the **Show Exceptions to Desired Model** check box are used together to help you find formulas that are “exceptions” to the type of model that you intended to create. Simply click the desired type of model – Linear, Quadratic, Conic, Nonlinear or Nonsmooth – check the box to Show Exceptions, and click the Check Model button again. This will re-run the analysis, and create a report worksheet that is inserted in the workbook, just to the left of the model worksheet. If you do this for EXAMPLE5 and select **Linear** in the Desired Model group, a report like the one shown on the next page will be inserted into your workbook.

Examples.xls

Microsoft Excel 11.0 Structure Report  
Worksheet: [Examples.xls]EXAMPLE5  
Report Created: 07/30/2006 12:00:00 PM  
Model Type: NLP Assumption: LP

Statistics

	Variables	Functions	Dependents
All	9	13	36
Smooth	9	13	36
Linear	0	12	27

Target Cell (Min)

Cell	Name	Variable	Formula	Exception 1	Exception 2
\$B\$19	Objective:	\$B\$10	EXAMPLE5!\$B\$19	\$B\$11	

This report shows that the objective function at cell B19 is an exception to your assumption of a linear model – it is a nonlinear function. Further, it depends nonlinearly on variables B10 and B11 (and possibly others). This dependence was first found at cell B19 on worksheet EXAMPLE5. If your objective formula had referred to a chain of other cell formulas, the report would show you the specific cell where a nonlinear operation or function was first found. EXAMPLE5!\$B\$19 in the report is a hyperlink – you can click on it to jump to the specific cell in question. In a large model, these links will help you quickly identify the “problem” formulas.

To save space, a maximum of three exceptions of each type is shown in the report for each problem function. After modifying any formulas that were shown as exceptions in the report, you should create a new report to identify any further exceptions to your assumed model type – until the model type changes to the type you expected.

If you've selected **Check For Convexity** instead of Structure when you click the Check Model button (with “Show Exceptions to Desired Model” checked), the Structure Report will additionally list all functions that are diagnosed as non-convex or whose convexity could not be determined. For example, if you select Desired Model Quadratic and Check For Convexity, the report will list any problem functions that are (i) not quadratic or linear, *and* any that are (ii) quadratic but are not convex.

## Transformed Tab: Transforming a Non-Smooth Model

As described in the chapter “Solver Models and Optimization,” the presence of the non-smooth function IF in EXAMPLE5 makes the model much harder to solve. With the Evolutionary Solver in the Premium Solver Platform, you can still solve such models. But where an LP can be solved very quickly and reliably up to very large size, and the solution is basically guaranteed to be optimal, a non-smooth model may take far more time to solve, and there are no guarantees as to whether the solution is truly optimal.

In EXAMPLE5, the IF functions are not *essential* to correctly model the real-world problem: you could use binary integer variables and linear constraints to achieve the same effect (a “fixed-charge constraint”). Techniques for doing this are described under “Improving the Formulation of Your Model” in the chapter “Building Large-Scale Models.” Modelers with training in operations research or management science often use these techniques when first formulating their models.

However, if you don't have time to learn these techniques, you may resort to the familiar functions IF, MIN, MAX, ABS, AND, OR, and NOT to model your real-

world problem. To improve your results, the Premium Solver Platform can *automatically* apply techniques analogous to those described above, to transform the model from your formulation to a different formulation that is easier to optimize.

To ensure that the transformed problem is well-scaled, it is **important to enter upper and lower bounds for all decision variables** in the Constraints list box. These bounds are used to compute well-scaled values for so-called “Big M” constants that are used in the constraints added during the transformation.

## Effects of Model Transformation

If your model includes non-smooth functions such as IF, MIN, MAX, ABS, AND, OR, and NOT, but is otherwise linear, the result of the Platform’s automatic transformation will be a linear mixed-integer (LP/MIP) model, with more variables and constraints. This transformed model may be solved by a variety of Solver engines, from the built-in LP/Quadratic and SOCP Barrier Solvers to the Large-Scale LP/QP, SQP and XPRESS Solvers. The result may be a much faster solution that is guaranteed to be optimal.

The automatic transformation process is not “magic.” You will still pay a price in solution time for the use of non-smooth functions, because the transformed model will be larger (more variables and constraints) and will include integer variables. As described in the chapter “Solver Models and Optimization” of the Premium Solver Platform User Guide, the presence of integer variables in a model makes it much harder to solve.

However, the automatic transformation from a problem with *non-smooth functions* to a problem with *integer variables* means that the arsenal of Solver engines available to optimize the problem is much larger. Many years of effort by Solver developers to improve the technology of solving LP/MIP models can now be applied to *your* model if it includes IF, MIN, MAX, ABS, AND, OR, and NOT functions.

The automatic transformation process uses general-purpose methods to replace IF, MIN, MAX, ABS, AND, OR, and NOT functions, and relational operators such as <, <=, >= and > with new binary integer and continuous variables, and new constraints. There are still good uses for the techniques described under “Improving the Formulation of Your Model,” because these techniques can yield a “tighter” formulation that solves in less time than the automatically transformed version of your model.

If the arguments you supply to IF, MIN, MAX, ABS, AND, OR, and NOT functions are linear functions of the variables, then the new constraints added to the problem will be linear functions; if the arguments you supply are not linear, the transformation process will still work, but the resulting model won’t be linear. If you also use other non-smooth functions (for example, CHOOSE, LOOKUP, or the Excel database functions, with arguments that depend on the variables) in your model, the result will still be a non-smooth model, and you will still need the Evolutionary Solver or the OptQuest Solver to find a solution.

## Using Automatic Model Transformation

As noted earlier, the **Original** tab displays statistics about the model as you originally defined it on the current Excel worksheet. The **Transformed** tab displays the same statistics for the “transformed” model, where the Interpreter has replaced certain non-smooth functions (if any) with new variables and constraints. You can easily see the effects of automatic model transformation: Just click on the Transformed tab, verify that Check For Structure (or Convexity) is selected, and click the Check Model

button. If you do this for the EXAMPLE5 model, the Solver Model dialog below will be displayed.

The Solver Model dialog box is shown with the 'Transformed' tab selected. It contains a table with columns: LP Convex, Variables, Functions, and NonZeroes. The 'All' row shows 27 variables, 67 functions, and 150 non-zeroes. The 'Smooth' row shows 27 variables, 67 functions, and 150 non-zeroes. The 'Quadratic' row shows 0 variables, 0 functions, and 0 non-zeroes. The 'Linear' row shows 27 variables, 67 functions, and 150 non-zeroes. The 'Bounds' row shows 54 variables and 8.29 sparsity %. The 'Integers' row shows 18 variables and 52 total cells. There are checkboxes for 'Solve Transformed Problem' and 'Show Transformations'. On the right, there are buttons for 'Close', 'Check Model', and 'Help', and a 'Check For' section with radio buttons for 'Gradients', 'Structure' (selected), 'Convexity', and 'Automatic'.

LP Convex	Variables	Functions	NonZeroes
All	27	67	150
Smooth	27	67	150
Quadratic	0	0	0
Linear	27	67	150
Bounds	54	Sparsity %	8.29
Integers	18	Total Cells	52

☐ Solve Transformed Problem ☐ Show Transformations

Check For:  
☐ Gradients  
☒ Structure  
☐ Convexity  
☐ Automatic

The transformed model has 18 additional variables, 9 of which are new integer variables, and 54 additional constraints – but it is now a linear integer (LP/MIP) model. It is also more sparse than before (8.29% versus 30.77%) – the LP/Quadratic Solver and other sparsity-exploiting Solvers will be able to take advantage of this.

If you're interested in the details of the additional variables and constraints, check the “Show Transformations” box and click the Check Model button again. This will add a Transformation Report like the one shown (in part) below to your workbook.

The screenshot shows a Transformation Report in Excel. The report includes the following information:

- Microsoft Excel 11.0 Transformation Report
- Worksheet: [Examples.xls]EXAMPLE5
- Report Created: 07/30/2006 12:00:00 PM
- Number of Artificial Variables: 18
- Number of Original Variables: 9
- Number of Artificial Constraints: 54
- Number of Original Constraints: 12

The report then lists the variables and their types:

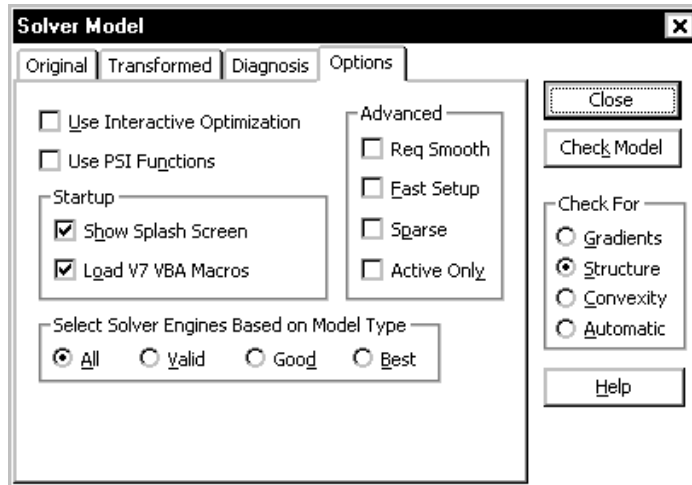
Origin	Type	Created
EXAMPLE5!\$I\$11	> or >= Binary	
EXAMPLE5!\$I\$11	IF Continuous	
EXAMPLE5!\$I\$12	> or >= Binary	
EXAMPLE5!\$I\$12	IF Continuous	
EXAMPLE5!\$I\$13	> or >= Binary	
EXAMPLE5!\$I\$13	IF Continuous	
EXAMPLE5!\$J\$11	> or >= Binary	
EXAMPLE5!\$J\$11	IF Continuous	

To solve the transformed model, simply check the box “Solve Transformed Problem” shown above on the Transformed tab. Then click the Close button to return to the main Solver Parameters dialog, and click the Solve button. The currently selected Solver engine will be used to solve the transformed problem. (To solve the original problem, return to the Solver Model dialog, click the Transformed tab, and uncheck the “Solve Transformed Problem” box.) If you do this for the EXAMPLE5 model, the Solver Results dialog will appear with the message “Solver found a solution. All constraints and optimality conditions are satisfied.” Since this is now an LP/MIP model and the Integer Tolerance is set to zero, the solution is globally optimal.

---

## Options Tab: Selecting Model Features

The Options tab of the Solver Model dialog provides a number of options to control the behavior of the Solver and the Polymorphic Spreadsheet Interpreter. The Advanced group of options on this tab is described in a later section, “Options Tab: Using Advanced Options.” The other options are described here.



### Use Interactive Optimization

This check box activates the *Interactive Optimization* feature of the Premium Solver Platform. If the box is checked, and you return to Excel worksheet Ready mode, then *each time you change a number on the worksheet*, the model will be optimized immediately – just as if you had selected **Tools Premium Solver**, clicked the **Solve** button, and clicked **OK** in the Solver Results dialog.

For small to medium size models, this is not only a convenience – it can be a real decision aid: You'll find that insights about your model, and decisions you can make, start to flow intuitively, when you can quickly see the impact of changing a parameter on the optimal solution.

For this feature to be useful, you should change a “parameter of the model” – a number in some cell that is used in the objective or a constraint, but is not itself a decision variable. As soon as you change this number (depending on the time needed to solve the problem), you'll see new values in the decision variable cells, and new calculated values for the objective and constraints.

### Use PSI Functions

This check box determines whether PSI functions for optimization – described in the chapter “Building Solver Models” under “Using PSI Functions” – are used to define your optimization model. The PSI functions for optimization, such as **PsiVar()**, **PsiCon()** and **PsiObj()**, are always listed in Excel's Function Wizard. But they are *recognized and used* to define a Solver model only if you check this box.

PSI functions provide an optional, alternative style for defining the variables, constraints and objective that make up your model. They are especially useful if you're creating a simulation optimization model, where you use PSI functions to specify the probability distributions of uncertain variables (such as **PsiNormal()**), and summary statistics for uncertain functions (such as **PsiMean()**).

When this box is *not* checked (the default setting), the edit boxes for PSI functions illustrated in the chapter “Building Solver Models” won’t appear in the dialogs, and any PSI functions for optimization on the worksheet will be ignored. If this box *is* checked, the Solver will scan for PSI functions on the worksheet each time you use Tools Premium Solver, and any new or changed PSI functions will be reflected in the objective, variables and constraints shown in the Solver Parameters dialog.

## Startup Options Group

The options in the Startup group were summarized in the chapter “Installation” and are also documented here. These two option settings are saved in the Registry (under HKEY\_CURRENT\_USER\Software\Frontline Systems\PremiumSolverPlatform\7.0) and are effective for all Excel sessions and workbooks; all other options are saved in the current workbook.

### Show Splash Screen

If this box is checked, a “splash screen” is briefly shown when the Premium Solver Platform COM add-in is first activated – typically when you first start Excel. If this box is unchecked, no splash screen is shown.

### Load V7 VBA Macros

If this box is checked, the Premium Solver Platform V7.0 COM add-in will load the V7.0 Solver.xla automatically; any earlier version of Solver.xla will be unloaded. This ensures that your VBA code calls to functions like SolverOK and SolverSolve will work with the Premium Solver Platform V7.0.

If this box is unchecked, the Premium Solver Platform V7.0 COM add-in will *not* load the V7.0 Solver.xla automatically. You must manually ensure that the version of Solver.xla that you want (V7.0, V6.x, standard Excel Solver, etc.) is loaded, using the File Open or Tools Add-Ins menu commands. See “Setting Startup Options” in the chapter “Installation” for more information.

## Select Solver Engines Based on Model Type

If you have many Solver engines installed for the Premium Solver Platform, and if you’re solving a wide range of models of different types, it may be not be immediately apparent which Solver engine is best for a given model. The Polymorphic Spreadsheet Interpreter can help, by diagnosing your model and then automatically select a subset of the Solver engines most appropriate for solving it. The selection is controlled by the option group.

When you use the Check Model button, the selection in the group **Select Solver Engines Based on Model Type** determines which Solver engines will appear in the Solver engine dropdown list in the main Solver Parameters dialog. **Choosing All** specifies that all available Solver engines should appear in the dropdown list, regardless of the model type.

**Choosing Valid** specifies that only Solver engines that are able to handle the current model type should appear. For example, if the model type is smooth nonlinear or NLP, the LP/Quadratic Solver will not appear, but the nonlinear GRG Solver, Interval Global Solver, and Evolutionary Solver will appear, since they are valid (but not necessarily good or best) for this model type.

**Choosing Good** specifies that only Solver engines that were specifically designed for this model type will appear. For example, if the model type is smooth nonlinear or

NLP, only the nonlinear GRG Solver and Interval Global Solver will appear; the Evolutionary Solver, while valid for smooth nonlinear models, is really designed for non-smooth models.

**Choosing Best** specifies that only Solver engines that are considered “best of breed” for this model type will appear. This is a very demanding criterion: If you have only the Solver engines built-in to the Platform, and you choose Best for an NLP model, you will receive a warning message, asking you for another choice. This is because certain field-installable Solver engines are considered “best” for these problems, whereas the built-in Solver engines are treated as simply “good” – so the Solver engine dropdown list would be empty!

Note that your selection in this option group will affect the Solver engine dropdown list only if you have clicked the Check Model button, with Check For set to Structure or Convexity, since the last time you used Tools Premium Solver... to display the Solver Parameters dialog. The Solver assumes that you may have edited formula cells and changed the model type between the time you close the dialog and reopen it later, so you must click the Check Model button again to redisplay model statistics and make the “Select Solver Engines Based on Model Type” option effective.

---

## Model Analysis When Solving

You can choose whether the Polymorphic Spreadsheet Interpreter or the standard Excel interpreter (recalculator) is used when solving a problem. If the PSI Interpreter is used, you can choose how much model analysis it performs before the actual solution process starts.

The Polymorphic Spreadsheet Interpreter offers many advantages when solving a problem: Besides computing values for your spreadsheet formulas, it can compute accurate *gradients* – which are needed by most Solver engines – at high speed, and it can tell the Solver engine which functions in your model are linear, quadratic, smooth nonlinear, or non-smooth – several Solver engines use this information to realize greater speed or solution accuracy.

But for some models, the gain in speed and/or accuracy isn't worth the extra time spent by the PSI Interpreter in model analysis – you are better off using the Excel interpreter. And there are a few Excel functions and formula syntax that are not supported by the PSI Interpreter, but can be used with the Excel interpreter.

1. The **Solve With** option in the Solver Model dialog Original tab determines whether the PSI Interpreter or the Excel Interpreter is used when you click Solve in the main Solver Parameters dialog.
2. If the Solve With option is set to PSI Interpreter, the **Check For** option group selection determines how much model analysis is done “up front” when you click Solve.
3. The **Advanced** options group, on the Options tab, controls the use of certain advanced features of the PSI Interpreter when you click the Solve button or the Check Model button.



Unknown	Variables	Functions	NonZeroes
All	9	13	
Smooth			
Quadratic			
Linear			
Bounds	18	Sparsity %	
Integers	9	Total Cells	

Solve With: ☒ PSI Interpreter ☐ Excel Interpreter

Check For: ☐ Gradients ☐ Structure ☐ Convexity ☒ Automatic

## Using the Solve With Option

**When the Solve With option is set to PSI Interpreter**, the Polymorphic Spreadsheet Interpreter is used when solving. The choice in the **Check For** option group determines how much model analysis is done “up front” when you click Solve, before the Solver engine starts searching for an optimal solution. If your model includes any formula syntax or Excel features not supported by the PSI Interpreter, an error message dialog will appear when you click the Solve button.

**When the Solve With option is set to Excel Interpreter**, the Polymorphic Spreadsheet Interpreter is not used when solving – instead, the Excel interpreter (recalculator) is used to compute all problem functions, and derivatives are estimated via finite differences. Note that the Interval Global Solver, SOCP Barrier Solver, and MOSEK Solver Engine cannot be used if Solve With is set to Excel Interpreter, or if the Check For option is set to Gradients.

**You ll need to choose Solve With = Excel Interpreter** if you receive an error message about unrecognized functions or formula syntax when you click the Solve button, or if the model type shown in the upper left corner of the Solver Model dialog appears as “Unknown” after you click Check Model. Most often, this means that your model uses special formula syntax, Excel functions, or user-defined VBA functions not supported by the PSI Interpreter.

**If you have Risk Solver Engine, and Interactive Simulation is “on”** at the time you click Solve, the Premium Solver Platform uses Solve With = Excel Interpreter for the optimization process – regardless of the Solve With setting in this dialog – and Solve With = PSI Interpreter for the Monte Carlo simulation process performed on each Trial Solution of the optimization. The simulation process, which typically accounts for most of the time, is up to 100 times faster than if Excel were used for each trial.

## Using the Check For Options

The Check For options group determines how much model analysis is done when you click **Check Model** in the Solver Model dialog, or click **Solve** in the main Solver Parameters dialog. The options Gradients, Structure, and Convexity take progressively more time “up front” and yield progressively more information about the model for you on Check Model, or for the Solver engine on Solve. The Automatic setting – often your best choice – allows the Solver engine to choose the option (Gradients, Structure, and Convexity) to be used when solving.

As noted above, the Interval Global Solver, SOCP Barrier Solver, and MOSEK Solver Engine will operate only if the Solve With option is set to PSI Interpreter, and the Check For option is set to Structure, Convexity, or Automatic. All other Solver engines can operate with any option setting, but they solve most efficiently with specific settings.

Please see the last section of this chapter for background information on the Polymorphic Spreadsheet Interpreter, the Excel interpreter, and the meaning of “finite differencing,” “automatic differentiation,” and “dependents analysis” in the following paragraphs.

### ***Check For = Gradients***

Choosing Gradients specifies that the PSI Interpreter should “parse” cell formulas on each Solve step, prior to running the selected Solver engine. When this is done, and the Solver engine requests function values and derivatives, they will be computed by the Interpreter; fast, accurate derivatives will be obtained via automatic differentiation. However, no structure or dependencies analysis will be available to the Solver engine.

### ***Check For = Structure***

Choosing Structure specifies that the PSI Interpreter should “parse” cell formulas and perform a structure analysis on each Solve step, prior to running the selected Solver engine. When this is done, and the Solver engine requests function values and derivatives, they will be computed by the Interpreter; fast, accurate derivatives will be obtained via automatic differentiation. Further, structure or dependencies analysis information will be available, if the Solver engine requests it.

### ***Check For = Convexity***

Choosing Convexity specifies that the PSI Interpreter should “parse” cell formulas, perform a structure analysis, and perform a convexity analysis on each Solve step, prior to running the selected Solver engine. No Solver engine currently requires this option, but it is built into the Premium Solver Platform for future use.

### ***Check For = Automatic***

Choosing Automatic – the default – specifies that either the Gradients or the Structure option will be chosen automatically, based on the type of model and the currently selected Solver engine’s ability to use the information.

As you may notice when using the Check Model button to diagnose your model, these steps can take some time for larger models – and they can also require significant amounts of memory. Structure analysis takes significantly more time and memory than Gradients analysis. The resources spent on this analysis are often repaid many times over when the Solver engine runs, but this depends on the Solver engine, and also, to some degree, on the model.

For example, the LP/Quadratic Solver and the Large-Scale LP/QP Solver both use Gradients when Solve With = Automatic, but they don’t use Structure, since the model is expected to be an LP, and a Structure analysis would simply show that all variables, functions and dependents were linear. But the Large-Scale SQP Solver and KNITRO Solver both use Structure when Solve With = Automatic, since they are designed to take advantage of linear dependents in a model that is nonlinear overall.

If you choose Structure for the LP/Quadratic Solver, the Structure analysis will be performed, but the LP/Quadratic Solver will make little or no use of this analysis, and

your overall solution time will probably be greater than if you chose Gradients or Automatic. If you choose Gradients for the Large-Scale SQP Solver, the engine will still solve the problem, but it will not be able to take advantage of Structure analysis. For most models, this will mean that the Solver engine will take more time than if you had chosen Structure; but in some cases – for example, models composed of all nonlinear functions – there would be little payoff from Structure analysis information, and the Solver engine might take the same or less total time.

In two situations, a Structure analysis is always performed when you click Solve, even if you've selected the Gradients option in the Solve With group:

1. When you check the Sparse box in the Advanced option group (described below). A Structure analysis is required for the Interpreter to operate in its own "Sparse mode."
2. When the Interval Global Solver is selected in the Solver engine dropdown list. This Solver engine doesn't accept any non-smooth functions, and it uses Structure analysis to check for them.

## Solve With Options and the Evolutionary Solver

The hybrid Evolutionary Solver in the Premium Solver Platform is designed to solve non-smooth problems, where derivatives of some of the problem functions may not be defined at certain points. But the Evolutionary Solver can also use classical methods, such as gradient search, where derivatives *are* required. Specifically, if you choose Gradient Local or Automatic Choice as the Local Search option, the Solver will attempt to compute derivatives for your problem functions. In this case, the Solve With option will determine how the gradients are computed.

When **Solve With = Excel Interpreter**, the PSI Interpreter is not used, and gradients are computed via *finite differencing*; the Solver will stop only if a simple recalculation yields an error. (The gradient values for non-smooth functions may not be valid at certain points; this may cause the gradient search to be slower or less effective.)

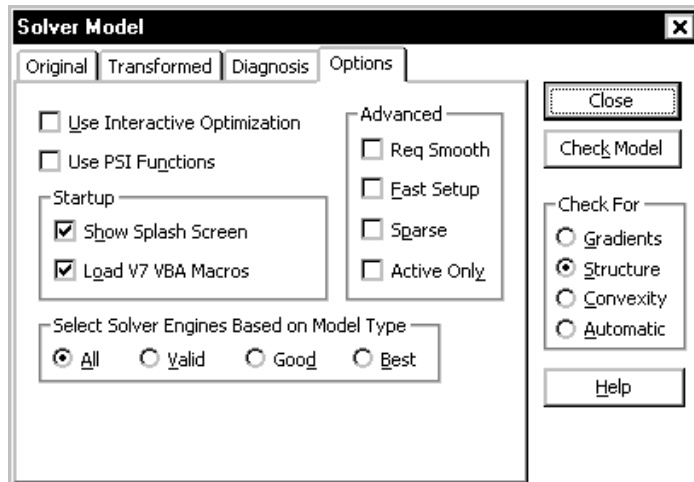
When **Check For = Gradients**, *automatic differentiation* is used, but diagnostic information is not produced, which means that the Evolutionary Solver cannot distinguish between non-smooth and smooth or linear variables and constraints. This choice is *not recommended* for use with the Evolutionary Solver.

When **Check For = Structure** or **Automatic**, *automatic differentiation* is used, and diagnostic information is produced and will be used by the Evolutionary Solver to distinguish non-smooth and smooth or linear variables and constraints. This choice often yields the best performance, provided the Solver does not stop because a gradient is undefined for a non-smooth function. If you ensure that the Advanced group option Require Smooth is *unchecked*, the Solver will compute approximate gradients for ABS, IF, MAX, MIN and SIGN when operating in this mode.

---

## Options Tab: Using Advanced Options

The Advanced group of check boxes, on the Options tab, controls advanced features of the Polymorphic Spreadsheet Interpreter during the Solve step. Except for the Require Smooth box, these options primarily affect the Interpreter's speed and memory usage on large models. The default values (all unchecked) are appropriate for most problems, but you may wish to experiment with these options to find the settings that yield the best performance on your models.



## Require Smooth

This option affects whether and how the PSI Interpreter diagnoses model types and computes derivatives for a small set of commonly used Excel functions – currently ABS, IF, MAX, MIN and SIGN. As explained in the section “Functions of the Variables” in the chapter “Solver Models and Optimization,” these functions are non-smooth, which means their derivatives are undefined at certain points: For ABS and SIGN at zero, for IF at the transitions between TRUE and FALSE in its first argument, for MAX and MIN at the transitions when one argument becomes larger or smaller than the others. But it is possible to compute derivatives of these functions at all points other than the transition points, and the Interpreter will do this automatically, *unless* the Req (Require) Smooth box is checked. Moreover, when diagnosing the type (LP, QP, QCP, SOCP, NLP or NSP) of a model, the PSI Interpreter will treat these functions as smooth nonlinear *unless* the Require Smooth box is checked.

If you are using the Solver Model dialog to diagnose your model type and optionally find “problem functions,” and you want to locate occurrences of ABS, IF, MAX, MIN or SIGN in your model, **be sure to check the Require Smooth box** before you click the Check Model button.

If Require Smooth is unchecked when you solve your model, the PSI Interpreter will evaluate derivatives of IF functions (for example) by computing a value – TRUE or FALSE – for the first argument, then computing derivatives of the second argument on TRUE, or the third argument on FALSE, using automatic differentiation. The result is very similar to that obtained when derivatives are estimated via finite differencing – the method used in the Excel Solver and Premium Solver. In most cases, these “directional derivatives” will enable a nonlinear optimization method to make progress towards a feasible or optimal solution. But the Solver will likely have trouble if the transition between TRUE and FALSE occurs near the optimal solution or the boundary of a constraint.

For the **Evolutionary Solver**, leaving Require Smooth unchecked will allow the gradient-based local search (if used) to proceed even if ABS, IF, MAX, MIN or SIGN functions are encountered. However, the option to **Fix Nonsmooth Variables** in the Evolutionary Solver will *not* fix variables that occur in these functions, because they won't be diagnosed as non-smooth.

If Require Smooth is checked, then variables occurring in these functions *will* be diagnosed as non-smooth. This in turn means that the first time the Solver engine requests derivatives of a function whose formulas use ABS, IF, MAX, MIN or SIGN, the Solver will stop with an Interpreter error message. If other non-smooth functions

– such as CHOOSE and LOOKUP – are used in your model, automatic differentiation will *always* cause the Solver to stop with an error message, unless you check the **Sparse** box, as described below.

### **Fast Setup**

In the Premium Solver Platform, use of the Polymorphic Spreadsheet Interpreter supersedes, for most models, the use of “Fast Problem Setup” as implemented in the Premium Solver and older versions of the Premium Solver Platform. Fast Problem Setup is essentially a simple and fast way to parse and interpret models that use a restricted set of Excel functions and formulas (such as SUM and SUMPRODUCT) for linear and quadratic models only, and obtain LP and QP coefficients for the Solver engines. The PSI Interpreter in the Premium Solver Platform can handle almost all Excel formulas and functions, so there is little reason to restrict models to “Fast Problem Setup form” today.

However, Fast Problem Setup still has some uses: Existing models in proper form for Fast Problem Setup may experience faster total solution times if you choose Solve With = Excel Interpreter, and for very large LP models (100,000 variables or more), Fast Problem Setup may be preferred, since the memory required for the Polymorphic Spreadsheet Interpreter may be greater than available RAM and may cause swapping to disk, which can be very slow. But, if you have a mix of models, some in Fast Problem Setup form and others not in that form, it would be inconvenient to switch frequently between Solve With = PSI Interpreter and Excel Interpreter.

**If you check the Fast Setup box**, old-style “Fast Problem Setup” will be tried first; if this fails (because the model is not in the required form), the Interpreter will be used instead. This option will have an effect only when you run a linear Solver engine, such as the LP/Quadratic Solver or the Large-Scale LP/QP Solver; it will not affect nonlinear models. It will take some extra time to examine the model using old-style Fast Problem Setup, but this process stops as soon as the first problem function not in the required form is encountered, so the time penalty is usually not very great. If you have selected Solve With = Excel Interpreter, old-style Fast Problem Setup will remain available, regardless of whether this Fast Setup box is checked.

### **Sparse**

When the Sparse box is checked, the PSI Interpreter operates internally in “Sparse mode,” when it is unchecked (the default), the Interpreter operates in “Dense mode.” This option affects *only the PSI Interpreter*, not the Solver engines – the latter are typically designed either for dense problems, like the GRG Nonlinear Solver, or for large, sparse problems, like the Large-Scale SQP Solver. Sparse mode also enables a Solver engine to request that **non-smooth variable occurrences should be ignored** when computing derivatives via automatic differentiation.

The Polymorphic Spreadsheet Interpreter can use significant amounts of memory (RAM), especially when diagnosing a model and computing derivatives via automatic differentiation. Memory usage grows with model size, and is greatest when the Solver is running (and using significant amounts of memory itself) and requesting derivatives. If your model is large enough, the memory required may exceed available RAM and cause Windows to begin swapping to disk – with a severe impact on solution time.

When it operates in “Sparse mode,” the PSI Interpreter uses sparse data structures, including “packed” gradient vectors and Hessian matrices, and index lists for the occurrences of variables in problem functions. Extra time is required to perform a

Structure analysis (which is required to take advantage of sparsity) and to create the sparse data structures.

For a large, sparse model, Sparse mode can save a significant amount of memory – and if this prevents swapping to disk, it will also save significant time. But if the model is very dense, Sparse mode can actually take more time and memory than Dense mode. Hence, you should check the Sparse box only for models where the Sparsity % figure in the Solver Model dialog is quite low.

If you have a *very* large, sparse linear or quadratic model, you should experiment to see whether the Sparse box yields the best performance. Bear in mind that checking the Sparse box will cause a Structure analysis to be performed before running the Solver engine. It is possible that the Structure analysis will require more time up-front than Sparse mode saves during the solution process.

For the **KNITRO Solver**, Sparse mode has a huge impact on performance. This Solver operates most effectively when it can obtain second derivatives (Hessians) from the Interpreter using automatic differentiation. But this process can consume large amounts of time and memory when the Interpreter is in Dense mode. If a large nonlinear model is sparse – as is usually the case – or if it includes many linear occurrences of variables (which contribute nothing to the Hessian of the function in which they occur), second derivative information can be computed far more efficiently in Sparse mode.

A Solver engine can request that, if Check For = Automatic, the Interpreter will run in Sparse mode *regardless* of the setting of the Sparse check box. The KNITRO Solver makes this request, since the Sparse box is unchecked by default, and Sparse mode is so critical to its performance. You can still force the Interpreter to run in its own Dense mode by setting Check For = Structure (or another choice different from Automatic) and leaving the Sparse box unchecked. But we recommend that you run most Solver engines with the default settings (Check For = Automatic), which will yield the best performance in the majority of cases.

For the **Evolutionary Solver**, Sparse mode can also have an important impact. Typical models for the Evolutionary Solver – which is limited to 500 decision variables – are not large enough to require this option for memory-saving purposes. But when the PSI Interpreter operates in Sparse mode, the Evolutionary Solver can – and will – ask the Interpreter to *ignore non-smooth variables* in automatic differentiation. The effect of this is to “fix” the non-smooth variables, making their partial derivatives zero, and to allow the Solver to proceed with automatic differentiation of any non-smooth function.

**So, if the Evolutionary Solver stops with the message “Solver encountered an error computing derivatives,” you should check the Advanced options group Sparse box in the Solver Model dialog, and click Solve again.**

### **Active Only**

When the Active Only box is checked, the PSI Interpreter will evaluate cells only on the active (frontmost) worksheet in the active workbook. Cells on other worksheets in the active workbook, or on sheets in other workbooks, that are referenced in formulas making up the Solver model will be treated as *constant* in the problem.

When the Active Only box is unchecked (the default), the Interpreter will evaluate all cells, on all worksheets, referenced in formulas involved in the Solver model. If the model references cells on sheets in other workbooks, these workbooks will be opened if available; otherwise the external cells “last known value” (as stored in the active workbook) is used and treated as constant in the model.

This box should be checked only if you have a large model that is spread across multiple worksheets, and you *want* all cells on worksheets other than the active sheet to be treated as constant in the problem. Note that, if these cells actually contain formulas that depend on the decision variables, this fact will be *ignored* and you will be solving a problem where these cells are effectively held constant at their last known values. If you have a large number of such cells on other worksheets, and especially if they contain formulas that do not depend on the decision variables, checking this box will save time in the Interpreter.

---

## More on the Polymorphic Spreadsheet Interpreter

The Polymorphic Spreadsheet Interpreter in the Premium Solver Platform fundamentally changes the way the Solver operates, and it affects – often dramatically – the performance of both the built-in and field-installable Solver engines. This optional section will give you more insight into how the Interpreter works, in comparison to the Microsoft Excel formula recalcuator, and how certain Solver engines take advantage of the Interpreter's considerably greater capabilities. To appreciate this section, you may need to review “Functions of the Variables” in the chapter “Solver Models and Optimization.”

### The Microsoft Excel Recalculator

Microsoft Excel includes an “Interpreter” of its own for Excel formulas, that is usually referred to as the formula recalcuator. The recalcuator is used to compute up-to-date values for formulas in your model whenever you enter or edit information in spreadsheet cells (when Excel is in “Automatic Calculation mode”) or when you press the F9 (Calc Now) key. As the standard “Interpreter” from Microsoft, it computes values for every kind of formula syntax or function that is legal in Microsoft Excel. It is controlled by options on the Calculation tab in the Tools Options dialog in Excel.

While it is invoked automatically when you work interactively with your spreadsheet, the Microsoft Excel recalcuator can also be invoked programmatically, by VBA code or by an add-in such as the Solver. Indeed, the Solver traditionally worked by writing new values into cells for decision variables, asking Excel to recalculate the model, then reading the computed values of cells for the objective and constraints.

Although it is fast and accurate, the Microsoft Excel recalcuator has a specific and limited purpose: To calculate function *values* in formula cells, given new values for other cells. It does not perform other tasks such as computing function *derivatives*, or analyzing formulas for linear or nonlinear dependents.

### Finite Differencing

Since the Microsoft Excel recalcuator computes only function values, but most Solver engines require both function values and function derivatives, the Excel Solver, Premium Solver, and previous versions of the Premium Solver Platform have traditionally used the Excel recalcuator to compute *approximations* of partial derivatives, using the method of *finite differencing*. This method is based on the definition of the partial derivative of a function  $f$  with respect to a variable  $x_j$ :

$$f / x_j = \lim_{0} \frac{f(x + e_j) - f(x)}{0}$$

where  $x$  represents the vector of decision variables  $[x_1 \ x_2 \ \dots \ x_n]$  and  $e_j$  is a unit vector (with 1 in the  $j$ th position and 0 elsewhere). While the definition applies only in the limit when  $\Delta$  goes to zero, an *approximation* of the partial derivative can be computed by choosing a very small value such as  $10^{-8}$  for  $\Delta$ . So the Solver uses the following steps:

1. Set the cells for the decision variables to  $x = [x_1 \ x_2 \ \dots \ x_n]$ .
2. Ask Excel to recalculate the model, thereby computing  $f(x)$ .
3. Set the cell for the  $j$ th variable to the “perturbed” value  $x_j + \Delta$ .
4. Ask Excel to recalculate the model, thereby computing  $f(x + e_j \Delta)$ .
5. Compute the difference of  $f(x + e_j \Delta)$  and  $f(x)$ , divided by  $\Delta$ .

These steps compute a partial derivative with respect to *one* variable. To compute the function *gradient* – the partial derivatives with respect to *all* of the variables – steps 3 through 5 above must be performed  $n$  times if there are  $n$  decision variables.

Most Solver algorithms require the gradient of the objective *and* the gradients of all the constraints. That is, they require the *Jacobian* matrix of partial derivatives, where each matrix row is the gradient of one function (see “Derivatives, Gradients, Jacobians, and Hessians” in the chapter “Solver Models and Optimization”):

$$\begin{bmatrix} f_1/x_1, & f_1/x_2, & \dots, & f_1/x_n \\ f_2/x_1, & f_2/x_2, & \dots, & f_2/x_n \\ \dots & \dots & \dots & \dots \\ f_m/x_1, & f_m/x_2, & \dots, & f_m/x_n \end{bmatrix}$$

This is not quite as expensive in computing time as it looks, because when the Solver asks Excel to recalculate the model at steps 2 and 4 above, Excel will calculate values for all of the problem functions at once. So the Solver can obtain approximate values for all partial derivatives by performing steps 1 – 2 once, and steps 3 – 5  $n$  times (once for each variable). In other words, the Solver obtains values for all of the partial derivatives in one column of the Jacobian matrix each time it asks Excel to recalculate the model at step 4.

For more than a decade, the Excel Solver and Premium Solver have used the finite differencing method to successfully solve optimization problems. But the method does have several drawbacks:

It is relatively slow, since the model must be recalculated  $n + 1$  times (and when solving a nonlinear problem, this must be done at each Trial Solution).

It is relatively inaccurate, since the subtraction and division typically result in a loss of significance – in the worst case *half* of the significant digits are lost.

If the Solver algorithm needs the gradient of only *one* function at each Trial Solution (perhaps because the constraints are all linear, with constant gradients), this takes as much time as it would to compute gradients of *all* the functions.

Each time it recalculates, Excel will compute values for *all formula cells in the spreadsheet* that depend on the perturbed decision variable cells – even cells that do not participate in the objective and constraints.

Computing second order partial derivatives (the Hessian matrix, as described in the chapter “Solver Models and Optimization”) is not practical – this would require  $n^2$  worksheet recalculations (a million for a 1,000-variable problem!) at each Trial Solution, and would yield derivative values of very low accuracy.



The slowness of finite differencing directly impacts solution time – especially for nonlinear problems, where finite differencing is performed many times. Since the Solver uses derivative values to determine the direction in which to search, the loss of accuracy in derivatives can lead to less-than-ideal search directions. While most Solver algorithms can “correct course” as they proceed, by computing a new search direction at each Trial Solution, less accurate derivatives will often mean that more major iterations will be needed to make the “course corrections,” and they may lead to less accurate final solutions.

Because many Solver models in Excel are really just part of a larger spreadsheet model that has many formulas calculating values of interest for other purposes, but not participating in the optimization problem, often the greatest drawback of using the Excel recalcuator is the fact that it always computes values for every formula cell that depends on the perturbed decision variable cells.

To achieve greater speed, accuracy, and control of the computation of derivative values, and to make it possible to evaluate the Solver model in other ways – for example, to determine linear and nonlinear dependents, and to evaluate models over *intervals* instead of single-point values – Frontline Systems developed its own Interpreter for Microsoft Excel.

## The Polymorphic Spreadsheet Interpreter

The Polymorphic Spreadsheet Interpreter in the Premium Solver Platform reads cell formulas, in the form that you write them such as `=A1*SUM(B1:B5)/EXP(-C1)`, and translates them into a compact intermediate code that can be processed efficiently each time that function values or derivatives are needed. It also builds a symbol table of names and cell references, used to look up current cell values and identify occurrences of decision variables.

The Interpreter acts in response to requests from Solver engines for function values and derivatives – or in response to your requests for a Gradients, Structure, or Convexity analysis, when you click the Check Model button. It scans the intermediate code for one or more problem functions (objective and constraints), and computes numeric values, using the current values of the decision variables (set by the Solver engine) and constants, arithmetic operators, and the like in the intermediate code.

The Microsoft Excel recalcuator and the PSI Interpreter are both designed to be very efficient. But where Excel reads and translates every cell formula that you create in a spreadsheet, the Interpreter translates only the cell formulas that are involved in calculating your objective and constraints (as you’ve defined them in the Solver Parameters dialog). And where Excel always computes values for every formula cell in the spreadsheet that depends on the changed decision variable cells, the Interpreter computes values for only the functions that the Solver engine actually needs. Because of this, on a spreadsheet where there are many cell formulas that aren’t directly involved in the optimization model, the Interpreter is usually faster than the Excel recalcuator when computing function values. But the Interpreter’s greatest benefit by far lies in computing function derivatives.

## Automatic Differentiation

When the PSI Interpreter computes partial derivatives for your objective and constraints, it uses a very different approach than the finite differencing method outlined earlier, called *automatic differentiation* in the technical literature. In essence, the PSI Interpreter *computes derivatives at the same time that it computes values* for functions, using algebraic relationships such as:

Sums:  $[f(x) + g(x)]/x = f(x)/x + g(x)/x$

Products:  $[f(x) * g(x)]/x = f(x)/x * g(x) + g(x)/x * f(x)$

Exponents:  $x^n/x = n * x^{n-1}$

Trig functions:  $\sin(x)/x = \cos(x)$ ,  $\cos(x)/x = -\sin(x)$ , etc.

The PSI Interpreter implements both “forward mode” and “reverse mode” automatic differentiation (further described in the technical literature), for both first partial derivatives (the Jacobian matrix) and second partial derivatives (the Hessian matrix). These partial derivatives are *computed to the same accuracy as the function values* themselves – hence, Solver engines can sometimes find the optimal solution with fewer Trial Solutions than required when finite differencing is used. And because of the way derivatives are computed, the time required is dramatically less than the time required for finite differencing – especially for “reverse mode” automatic differentiation (which is used for all expressions except array formulas).

Thanks primarily to automatic differentiation, on a sample of small and medium-size actual user models, total solution times (which include much more than the time spent computing derivatives) for the Premium Solver Platform were on average *twice as fast* for linear problems and *seven times faster* for nonlinear problems. Since the speed advantage of automatic differentiation *grows* with the number of variables in the problem, larger models may experience even greater speed gains (provided that they are run on PCs with sufficient RAM for the Interpreter).

### ***Interval Arithmetic and Interval Differentiation***

As described in the “Introduction,” the PSI Interpreter can also evaluate Excel formulas over intervals rather than single numeric values. An *interval* such as [1, 2] represents all of the possible numeric values between 1 and 2. Addition, subtraction, and other arithmetic operations and functions can be defined over intervals – for example,  $[1, 2] + [3, 4] = [4, 6]$  in interval arithmetic. The Interpreter can compute interval values for all of Excel’s arithmetic operators and most built-in smooth functions. At present, the interval values are not displayed in spreadsheet cells, but the Interpreter computes function values and first and second partial derivatives over intervals, for use by the new Interval Global Solver in the Premium Solver Platform.

The ability to evaluate Solver models over intervals is a fundamental new capability in the Premium Solver Platform that is not yet available in other, far more expensive modeling systems and Solvers. Interval analysis makes it possible for the Interval Global Solver to overcome the limitations of classical methods for nonlinear optimization and equation-solving, and find the true global optimum of a constrained problem, or find *all real solutions* of a system of nonlinear equations – a capability described under the heading “What is Not Possible” in one classic optimization textbook. With the Premium Solver Platform, this capability is not only possible, but is readily available and very easy to use.

### ***Model Diagnosis and Structure Analysis***

The PSI Interpreter is also responsible for diagnosing your model as linear (LP), quadratic (QP), quadratically constrained (QCP), second order cone (SOCP), smooth nonlinear (NLP) or non-smooth (NSP) and providing statistics on linear, quadratic and smooth variables, functions and nonzeros that you see when you click the Check Model button in the Solver Model dialog. And it provides model type, sparsity and “Structure analysis” information to Solver engines during the solution process, when you select the Check For = Structure option.

The way the PSI Interpreter does this is very similar to the way it computes derivatives via automatic differentiation. The Interpreter computes symbolic “values” for the dependence of functions on decision variables for every cell formula in your model, using algebraic relationships such as:

*Sums: The sum of two linear functions is a linear function*

*Products: The product of a constant (independent) function and a linear function is a linear function; the product of two linear functions is a quadratic function*

The PSI Interpreter is also responsible for the Scaling Report, described later in this Guide in the chapter “Solver Reports.” It computes symbolic “values” for every cell formula in your model, based on the magnitudes of the values of decision variables, and algebraic relationships that capture the effect of addition, multiplication, and similar operations on the magnitudes of function results. This is another unique capability of the Premium Solver Platform.

## Convexity Analysis

Finally, the PSI Interpreter is responsible for the new, automatic test for convex models and functions in the Premium Solver Platform. As mentioned earlier, this test may yield conclusive or inconclusive results; a convexity test that always yielded conclusive results would take time that grew exponentially with the number of variables, and hence would be impractical for even modest-size models. The methods used in the Premium Solver Platform are designed to yield useful results in many, but not all cases, while taking a “reasonable” amount of time. In the worst case, the test involves computing the interval Hessians of all of the problem functions and performing interval vector-matrix operations on each of these Hessians.

A **linear function** is always convex (and concave), and its Hessian matrix is always zero. Hence the convexity test takes no extra time for linear functions beyond the analysis done for the Check For Structure option.

A **quadratic function** has a constant Hessian matrix, which means that the interval Hessian is the same as the real Hessian, and the convexity test will yield a conclusive result, based on the positive (or negative) definiteness of the Hessian matrix. (A positive definite or semidefinite Hessian means that the quadratic function is convex; a negative definite or semidefinite Hessian means that the function is concave.)

For **general smooth nonlinear functions**, the convexity test first computes an “outer approximation” of the feasible region, which can be pictured as a box (bounds on the decision variables) that encloses the actual feasible region determined by the intersections of the constraints. (The Interpreter starts with the variable bounds that you specify, then uses constraint propagation methods to “shrink” this box.)

The convexity test then quickly computes a result based on the sign of the interval Hessian over this box. For some – but not all – functions, this is sufficient to determine the convexity of the function. If this test is not sufficient, the full interval Hessian is computed, and several numerical methods are applied to test whether this interval matrix is positive (or negative) definite.

The convexity test is yet another capability of the Premium Solver Platform that is not available in other, far more expensive modeling systems and Solvers.

## Excel Built-in Functions

Microsoft Excel has over 320 built-in functions, including the financial, statistical, and engineering functions that are part of the Excel Analysis ToolPak. The Inter-

preter supports almost all of these functions. Functions that are recognized but not supported include:

CALL	HYPERLINK
CELL	OFFSET
GETPIVOTDATA	REGISTER.ID
INDIRECT	SQLREQUEST
INFO	CUBExxx (Excel 2007)

The most commonly used of these functions – OFFSET and perhaps INDIRECT – are not supported because they can reference arbitrary cell “addresses,” and the Interpreter is designed to process only the formula cells that participate in calculation of your objective and constraints, not the full spreadsheet “grid.”

Since the Interpreter computes values for your problem functions without using the Microsoft Excel recalcuator, how confident can you be that the values computed this way will match the values that would have been computed by Excel? While discrepancies are always possible, one reason for confidence is that Frontline Systems actually developed, under contract to Microsoft, implementations of most of the Excel built-in functions for use in the Internet Explorer “spreadsheet component” that is included with Microsoft Office 2000, XP, 2003 and 2007 (file msowcf.dll in the Microsoft Office program directory). Frontline’s implementation of these functions was tested against the same functions in Microsoft Excel, in an extensive quality assurance process during the development of Office 2000. Because of this, Frontline Systems was uniquely qualified to develop a full-scale Interpreter for Microsoft Excel and its extensive library of built-in functions.

When a Solver engine displays a final solution with the Solver Results dialog box, or an intermediate solution with the Show Trial Solution dialog box, the current values of the decision variables are placed in cells on the spreadsheet, and at this point the *Microsoft Excel recalcuator* is used to compute values for the objective and constraints – even when the Interpreter has been used internally during the solution process. So you can be 100% confident that the values you see won’t change when you save and later reopen your workbook!

# Building Large-Scale Models

---

## Introduction

It is a maxim that a successful Solver model will grow in size over time. When the initial results from an optimization model demonstrate ways to achieve significant cost savings, improved schedules, higher quality or increased profits, management is naturally interested in applying these methods to bigger problems. This might involve extending the model to include more plants, warehouses, assembly lines, or personnel; to bring in other divisions or geographic regions; or to cover more time periods, more detailed process steps, or more specific parts or products. The result is an increase in the number of decision variables, constraints, and cells in your model.

When your model grows in size, it becomes more challenging to design and maintain, and also more challenging to solve. Good modeling practices – touched upon in the chapter “Building Solver Models” – become *far* more important, so your model remains comprehensible to other Excel users, auditable for errors, and easy to modify. Issues such as your model type (LP, QP, QCP, SOCP, NLP or NSP), sparsity, and scaling also become *far* more important, since they strongly influence the time it takes to solve your model, and the reliability of the solutions you obtain.

This chapter can only briefly survey good modeling practices – entire books have been devoted to this subject (we will recommend some). It focuses on steps you can take to obtain faster and more reliable solutions for large models using the Premium Solver and Premium Solver Platform, including:

- Steps towards better performance that are easy to apply in most situations

- Steps you can take – with more design and modeling effort – to improve the *formulation* of your model, by replacing non-smooth or nonlinear constraints with linear (or integer linear) constraints

- Steps you can take to enable the Premium Solver (and in some cases, the Premium Solver Platform) to analyze your model more efficiently

---

## Designing Large Solver Models

A large Solver model in Microsoft Excel is both a large spreadsheet workbook and a large optimization model. If you plan to build such a model, you will be well advised to learn about good spreadsheet modeling practices, and about good optimization modeling techniques.

We highly recommend the textbook *The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft* by Stephen G. Powell and Kenneth R. Baker, published by John Wiley & Sons, listed at the end of the chapter “Introduction.” Unlike other management science textbooks, this book teaches you “best practices” in modeling and spreadsheet engineering, as well as techniques of linear and nonlinear optimization using Excel.

Other books on good spreadsheet design are hard to find, but through resources like the Amazon.com Marketplace, you may be able to locate a copy of John Nevison's book *Microsoft Excel Spreadsheet Design* (Prentice-Hall, 1990), or his earlier works *1-2-3 Spreadsheet Design* (1989) or *The Elements of Spreadsheet Style* (1987), both of which are still useful in designing modern spreadsheets. A relatively new (2003) book, *Excel Best Practices for Business* by Loren Abdulezer, includes chapters on spreadsheet construction techniques, “makeovers” of spreadsheets developed by others, and spreadsheet auditing.

Training courses in Microsoft Excel often cover at least some elements of good spreadsheet design. They are offered in many venues, from universities and community colleges to public seminars, in-house corporate training, and classes sponsored by computer dealers. Check the course outline or syllabus to see if it features spreadsheet design and good modeling practices, and other topics most relevant to you.

A readily available book on optimization modeling techniques is H. Paul Williams's *Model Building in Mathematical Programming, 4th Edition* (John Wiley, 1999), listed at the end of the chapter “Introduction.” Focusing on modeling for linear and integer programming problems, it includes a treatment of large-scale model structure and decomposition methods that is hard to find elsewhere.

## Spreadsheet Modeling Hints

Below is a brief set of suggestions for planning, designing and constructing large Solver models:

**Start with a Plan.** Plunging in and entering numbers and formulas immediately will quickly lead to problems when constructing a large spreadsheet. Write down your objectives and sketch out a design before you begin working on the real spreadsheet.

**Build a Prototype.** Plan in advance to build a prototype, *throw it away*, and then build the real spreadsheet model. What you learn from building and solving the prototype will probably save you time in the long run.

**Create a Table of Contents.** In the upper left corner of your first worksheet, include comments that point readers to the major areas or sections of the spreadsheet.

**Separate Data and Formulas.** Avoid using constants in formulas, unless they are intrinsic to the mathematical definition of the function you are using. Instead, place constants in cells, and refer to those cells in formulas. Create separate areas on the spreadsheet for input data and for calculations, and identify these with distinct colors, borders or shading.

**Document Assumptions, Parameters and Methods.** As John Nevison suggested, seek to “surface and label every assumption” in your model. Use labels or cell comments to document key formulas and complex calculations.

**Use defined names.** Use Excel's Insert Name Define and Insert Name Create commands to assign meaningful names to individual cells and cell ranges. This will help make your formulas clearer and more flexible.

**Use and Separate Two-Dimensional Tables.** Many elements of your model will lend themselves to a row-column table representation. Create separate table areas, with distinct colors, borders and shading. Collect non-table data (such as individual parameters) into a separate area.

**Use Excel Tools to View and Audit Your Spreadsheet.** Use the View Zoom command to get a high-level view of your spreadsheet's structure. Use the View tab in the Tools Options dialog to display formulas instead of values, and scan them for consistency. Learn to use the Auditing Toolbar (Tools Auditing...) to trace precedents and dependents of your formulas.

**Use a Spreadsheet Auditing Tool.** Several auditing tools are available, including *SpACE* from the UK Customs and Excise Audit unit, *OAK* from Operis Ltd. in the UK, and the *Spreadsheet Detective* from Southern Cross Software in Australia.

## Optimization Modeling Hints

**Identify Your Model's Index Sets.** Your decision variables, constraints, and many intermediate calculations will fall into groups that are *indexed* by elements such as products (A, B, ...), regions (North, South, ...), time periods (January, February, ...) and similar factors. Identify and write down these index sets and their members. Then organize the columns and rows of your table areas using these index sets. Use the top row and left column of each table area for index set member names as labels.

**Identify Your Decision Variables.** Once you've identified the quantities that will be decision variables, and how they are indexed (for example, units made by product A, B, ... or shipments by region North, South, ...), it's usually easier to determine the constraints and their indexing.

**Determine the Data You'll Need.** In building large optimization models, you will frequently spend a good part of your time figuring out what data you need, how you will get it (and keep it up to date), and how you'll have to summarize or transform it for the purposes of the model. This may involve getting help from your IT department or from other groups that create or maintain the data.

**Define Balance Constraints.** It is easy to overlook "balance" or "continuity" constraints that arise from the physical or logical structure of your model. For example, in a multi-period inventory model, the ending inventory at time  $t$  must equal the beginning inventory at time  $t+1$ . At each node of a network model (such as a warehouse), the beginning item quantity plus incoming deliveries minus outgoing shipments must equal the ending item quantity ("what goes in must come out").

**Learn to Use Binary Integer Variables.** Many relationships that you might find difficult to model at all, and many where you might otherwise use IF, CHOOSE or other non-smooth or discontinuous functions, can be effectively modeled with binary integer variables. The section below "Improving the Formulation of Your Model" describes many situations where you can use such variables to organize your model.

## Using Multiple Worksheets and Data Sources

Large Solver models and their data are often organized into multiple worksheets of a single workbook. Some large models reference data found in other workbooks. Given the large number of data elements, the sources from which you are getting the data, and the procedures you use to keep the data up to date, multiple worksheets are often necessary or at least useful for organizing your data.

The Premium Solver requires that cells containing decision variables and constraint left hand sides are on the active worksheet. But the Premium Solver Platform allows you to define decision variables and constraint left hand sides on any worksheet of a workbook. For this and many other reasons, you are well advised to upgrade to the Premium Solver Platform if your model grows in size. With either product, the formulas in your objective and constraint cells can refer to cells on other worksheets, and those cells on other worksheets can contain formulas that depend, directly or indirectly, on decision variable cells. For more information, see “Models Defined Across Multiple Worksheets” in the chapter “Building Solver Models.”

Several commentators on good spreadsheet modeling practice feel that models defined on a single worksheet are easier to understand and maintain. In Excel 2007, a single worksheet can have up to 16,384 columns and 1,048,576 rows. So you may want to keep the core of your Solver model – the formulas (i) that are used to compute your objective and constraints and (ii) that depend on the decision variables – on a *single worksheet*. If you find that you can better structure your model by placing decision variables and constraints on different worksheets, it is highly recommended that you adopt a consistent scheme for choosing blocks of variable and constraint (and other formula) cells, and referencing these cells across worksheets.

Some of the data you need may be available in relational databases, OLAP databases or data warehouses. Microsoft Excel provides rich facilities, such as external data ranges and PivotTables, to bring such data into an Excel worksheet. The raw data, even if partially summarized from database records or transactional data, often needs to be further transformed and summarized on your worksheet(s). This is usually easy to do with Excel formulas. But for clarity in your model, we recommend that you use separate worksheet areas, with distinct colors, borders or shading, for formulas that simply massage the data and do not participate in the solution process (i.e. do not depend on the variables). The Solver can determine which formulas depend on the variables, but *you* or your colleagues may find it difficult to do so if the formulas are intermixed.

---

## Quick Steps Towards Better Performance

The rest of this chapter focuses on steps you can take to obtain *faster and more reliable solutions* for large models from the Premium Solver and Premium Solver Platform. This section describes steps that are easy to apply in most situations.

For users of the Premium Solver, the best recommendation we can make to improve performance is to *upgrade to the Premium Solver Platform*. This is more than just a “sales pitch” – every step you take costs something, either in terms of money or your effort. For most professionals, the cost of upgrading will be repaid if it saves just a few hours of time. And you can find out *at no cost* whether the upgrade will be worthwhile – just download the Premium Solver Platform Setup program, request a free 15-day evaluation license, and try solving your model with the actual software.

For users of the Premium Solver Platform, we highly recommend that you *try solving your model with our field-installable Solver Engines* – especially the Large-Scale SQP Solver, KNITRO Solver, MOSEK Solver Engine, and XPRESS Solver Engine. While the difference in cost may be greater, the same rationale applies: If you can solve your model more quickly or more reliably by upgrading the software, this is almost always cheaper (and yields results sooner) than spending many hours or days of valuable professional time.



## ***Ensure that You Have Enough Memory***

If the Solver seems unusually slow, check whether the hard disk activity LED (present on most PCs) is flickering during the solution process. If it is, memory demands may be causing Windows to swap data between main memory and disk, which greatly slows down the Solver. If you're investing money and, especially, hours of your time to develop an optimization model, consider that RAM is very cheap, and relatively easy to install. We recommend *at least* 512MB RAM if you are working with large Solver models – 1 GB or more is certainly desirable.

## ***Analyze Your Model for Scaling Problems***

Poorly scaled calculations are a frequent cause of long solution times and unreliable solution results, for both linear and nonlinear problems. For a further discussion, see “Problem s w ith Poorly Scaled M odel s” in the chapter “D iagnosing Solver R esult s.” In the Premium Solver Platform, use the Scaling Report to automatically diagnose scaling problem s, as described in the chapter “Solver R eports.”

## ***Add Constraints to Your Model***

Frequently, you can improve solution time by adding constraints to your model which may not be *essential* in defining the problem, but which do *further constrain* the search space that the Solver m ust explore. It s true that the Solver m ust do m ore work to handle the additional constraints, but this extra work usually has an excellent payoff if the constraints are “binding” (i.e. satisfied w ith equality) at som e point during the solution process.

The greatest payoff often comes from additional constraints that are simple bounds on the decision variables. This is because (i) it s usually easier for you to determ ine realistic lower and upper bounds on the variables than to formulate new general constraints, (ii) it s easy to enter bounds on the variables in the Constraints list box, and (iii) each of the Solver engines is able to handle bounds on the variables more efficiently than general constraints.

Users often omit upper bounds on their decision variables, and sometimes omit lower bounds as well. A first step towards improving performance is to enter the tightest bounds on the variables that you can, without eliminating possible good solutions.

Since bounds on the variables are especially important for the performance of the Evolutionary Solver and for multistart methods for global optimization used with the nonlinear Solver engines, the Options dialogs for these Solver engines include a check box “R equire B ounds on V ariables,” w hich is *checked* by default. When this box is checked, the Solver will stop with an error message if some variables do not have lower or upper bounds at the time you click Solve. If you are using the Interval Global Solver or the OptQuest Solver, bounds on all variables are *required* – the Solver will always stop with an error message if bounds on the variables are missing.

---

# **Improving the Formulation of Your Model**

The type of problem you are trying to solve, and the solution method or Solver engine that must be used, has a major impact on solution time:

Linear programming problems can be solved most quickly.

Quadratic programming problems take somewhat more time.

Nonlinear optimization problems take considerably more time.

Non-smooth problems take by far the greatest amount of time.

This section discusses techniques you can use to replace nonlinear functions, and even non-smooth functions, with equivalent (or nearly equivalent) linear or quadratic functions, or with linear functions and binary integer variables. As explained in the chapter “Solver Models and Optimization,” a problem with integer variables can take much longer to solve than a problem without such variables. However, an integer linear problem formulated using the techniques described in this section may still take significantly *less* time to solve than the equivalent nonlinear or non-smooth problem. Moreover, if your problem is integer linear, you can find a *guaranteed* optimal solution, or a solution that is guaranteed to be within at least x% of optimal, whereas with a nonlinear or non-smooth problem you will have no such guarantees. As a rough guide, non-smooth models with more than 1,000 variables may be difficult or impossible to solve in a reasonable amount of time – but equivalent models formulated with linear functions and binary integer variables can often be solved efficiently with the LP/Quadratic Solver. And with the Large-Scale LP/QP Solver Engine or the XPRESS Solver Engine, you can often solve linear integer problems of 10,000, 100,000 or more variables in a reasonable amount of time.

A caveat: If you currently have a model with many nonlinear or non-smooth functions, and you decide to implement some of these techniques to speed up solution of your model, bear in mind that you can use the LP/Quadratic Solver, Large-Scale LP/QP Solver, or XPRESS Solver only for models where *all of the problem functions* are linear (except for the objective function, which may be quadratic). If you create a model with a *mix* of nonlinear or non-smooth functions and linear functions using binary integer variables, it may still take a long time to solve.

## Techniques Using Linear and Quadratic Functions

Below are three common situations where you might at first expect that a nonlinear function is required to express the desired relationship – but with a simple transformation or approximation, you can use a linear or quadratic function instead.

### Ratio Constraints

You may want to express a relationship that seems to require dividing one or more variables by other variables. Suppose that you have a portfolio of 1-month, 3-month and 6-month CDs, with the amounts of each CD in cells C1, D1 and E1, and you wish to limit the average maturity to 3 months. You might write a constraint such as:

$$(1*C1 + 3*D1 + 6*E1) / (C1 + D1 + E1) \leq 3$$

This constraint left hand side is a nonlinear function of the variables, so you would have to use the GRG Solver to find a solution. However, the same constraint can be rewritten (multiplying both sides by the denominator, then collecting terms) as:

$$(1*C1 + 3*D1 + 6*E1) \leq 3*(C1 + D1 + E1), \text{ i.e. } -2*C1 + 3*E1 \leq 0$$

This constraint is a linear function of the variables, so you would be able to use the much faster Simplex or LP/Quadratic Solver to find a solution. (This transformation above relies on the fact that  $C1 + D1 + E1 \geq 0$ .)

### Mini-Max and Maxi-Min

You may want to minimize the maximum of a group of cells such as C1:C5 (or maximize the minimum of a group of cells). It is tempting to use an objective function such as MAX(C1:C5) – but as explained in the chapter “Solver Models and Optimization,” MAX (and MIN) are non-smooth functions, so you do need to use at

least the GRG Solver, and perhaps the Evolutionary Solver to find a solution. Instead, you can introduce another variable D1, make D1 the objective to be minimized, and add the constraint:

C1:C5 <= D1

The effect of this constraint is to make D1 equal to the maximum of C1:C5 at the optimal solution. And if the rest of your model is linear, you can use the much faster Simplex or LP/Quadratic Solver to find a solution.

## Quadratic Approximations

If you cannot represent the entire problem using linear functions of the variables, try to formulate it as a quadratic (QP) or quadratically constrained (QCP) problem, with a quadratic objective and/or constraints. You may be able to use a local, quadratic approximation to a smooth nonlinear function  $f$  near a point  $a$ :

$$f(x) \approx f(a) + f'(a)(x - a) + \frac{1}{2}f''(a)(x - a)^2$$

where  $f'(a)$  denotes the first derivative, and  $f''(a)$  denotes the second derivative of the function  $f$  at the point  $a$ . Several Solver engines offer excellent performance on QP problems, and the SOCP Barrier Solver and the MOSEK Solver Engine offer good to excellent performance on QCP problems.

Even if you cannot eliminate nonlinear functions from your problem altogether, you can improve solution time by making an effort to ensure that as many variables as possible occur linearly in the objective and all of the constraints. You can select the “Recognize Linear Variables” check box in the GRG Solver Options dialog to save time during the solution process. And the Large-Scale GRG Solver and Large-Scale SQP Solver engines also recognize both *linearly occurring variables* and *linear constraints* automatically, for still faster solutions. The Large-Scale SQP Solver is particularly effective when used with the Premium Solver Platform, because it uses the Interpreter’s Structure analysis to *break down each function into linear and nonlinear terms*, which it handles as efficiently as possible.

You can use the Solver Model dialog in the Premium Solver Platform to easily determine the number of linear variables, functions, and occurrences of variables in functions, as described in the chapter “Analyzing and Solving Models.”

## Techniques Using Linear Functions and Binary Integer Variables

Below are three common situations where you might at first expect that a non-smooth function such as IF is required to express the desired relationship – but you can instead use a binary integer variable and one or two linear functions to define an equivalent relationship. The techniques described here are similar to those used when the Premium Solver Platform *automatically transforms* your model (see the chapter “Analyzing and Solving Models”), but you can apply these techniques yourself to handle situations where the automatic transformation is not available.

### Fixed-Charge Constraints

You may have a quantity  $x$  in your model that must “jump” from zero to some (fixed or variable) non-zero value, under certain conditions. For example, a machine on a production line may have a fixed setup time or cost if it is used at all, plus a time or cost per unit produced. You can avoid creating a non-smooth function for  $x$  by introducing a binary integer variable  $y$  (which is 1 if  $x$  is used and 0 if it isn’t), and

adding a constraint  $x \leq My$ , where  $M$  is a constant that is larger than any possible value for  $x$ .

For example, suppose you have a machine that has a setup time of 10 minutes, but once set up will process a widget every 30 seconds. Let cell C1 hold the number of widgets you are producing on this machine, and use cell E1 for a binary integer variable  $y$  that is 1 if you produce *any* widgets on this machine. Then the total production time can be computed as  $=0.5*C1+10*E1$ . Assuming that C1 can be at most 10,000, let  $M1 = 10000$  and add a constraint:

$$C1 \leq M1*E1 \quad (\text{or } C1 - M1*E1 \leq 0)$$

If variable C1 is nonnegative ( $C1 \geq 0$ ) and variable E1 is binary integer ( $E1 = \text{binary}$ ), then C1 is forced to be 0 whenever E1 is 0, or equivalently E1 is forced to be 1 whenever C1 is greater than 0. Since the production time calculation and the constraint are both linear functions, you can solve the problem with the Simplex (or LP/Quadratic) Solver and the Branch & Bound method. This is called a *fixed-charge* constraint.

### Either-Or Constraints

Constraints in an optimization problem are implicitly connected by the logical operator AND – all of them must be satisfied. Sometimes, however, your model may call for either one constraint (say  $f(x) \leq F$ ) or another constraint (say  $g(x) \leq G$ ) to be satisfied. You might consider using the OR function in Excel, but as noted in the chapter “Solver Models and Optimization,” this function is non-smooth. Instead, you can introduce a binary integer variable  $y$  and a constant  $M$ , where  $M$  is greater than any possible value for  $f(x)$  or  $g(x)$ , and add the constraints  $f(x) - F \leq My$  and  $g(x) - G \leq M(1-y)$ . Now, when  $y=0$ ,  $g(x)$  is unrestricted and  $f(x) \leq F$ ; but when  $y=1$ ,  $f(x)$  is unrestricted and  $g(x) \leq G$ .

For example, imagine you want to allocate your purchases among several suppliers in different geographic regions, each of whom has imposed certain conditions on their price bids. Suppose that one supplier's bid requires that you either purchase at least 400 units from their Chicago warehouse or else purchase at least 600 units from their Phoenix warehouse, in order to obtain their most favorable pricing. Let cell C1 hold the number of units you would purchase from Chicago, and cell D1 hold the number of units you would purchase from Phoenix. Assume that cell M1 contains 10,000 which is more than the maximum number of units you intend to purchase. You can model the supplier's either-or requirement with a binary integer variable in cell E1 and the following constraints:

$$\begin{aligned} 400 - C1 &\leq M1*E1 \\ 600 - D1 &\leq M1*(1-E1) \end{aligned}$$

Notice that we have reversed the sense of the constraint left hand sides to reflect the “at least” ( $\geq$ ) requirement. If  $E1=0$ , then C1 (units purchased from Chicago) must be at least 400, and the second constraint has no effect. If  $E1=1$ , then D1 (units purchased from Phoenix) must be at least 600, and the first constraint has no effect.

### IF Functions

In the chapter “Solver Models and Optimization,” we used  $=IF(C1>10,D1,2*D1)$ , where C1 depends on the decision variables, as an example of a non-smooth function: Its value “jumps” from D1 to  $2*D1$  at  $C1=10$ . If you use this IF function directly in your model, you'll either have to try the Transformation tab in the Solver Model dialog, or else solve the model with the Evolutionary Solver. Instead, you can avoid the IF function and solve the problem with the nonlinear GRG Solver – or even

the linear Simplex Solver – by introducing a binary integer variable (say E1) that is 1 if the conditional argument of the IF is TRUE, and 0 otherwise. Add the constraints:

$$C1 - 10 \leq M1 * E1$$

$$10 - C1 \leq M1 * (1 - E1)$$

When E1 is 0, the first constraint forces  $C1 \leq 10$ , and the second constraint has no effect. When E1 is 1, the first constraint has no effect, and the second constraint forces  $C1 \geq 10$ . (If  $C1=10$  exactly, this formulation allows either  $E1=0$  or  $E1=1$ , whichever one yields the better objective.) The value of the IF function can then be calculated as  $D1 * E1 + 2 * D1 * (1 - E1)$ , which simplifies to  $D1 * (2 - E1)$  in this example. If D1 is constant in the problem, this is a linear function; if D1 depends linearly on the variables, it is a quadratic; otherwise, it is a smooth nonlinear function. In all cases, the non-smooth behavior has been eliminated.

Depending on how you use the result of the IF function in the rest of your model, you may be able to take this strategy further. Suppose, for example, that if  $f(x) \geq F$  then you want to impose the constraint  $g(x) \leq G$ ; if  $f(x) < F$  then you don't need this constraint. You can then use a binary variable y (cell E1 in the example above), and impose constraints like the pair above plus an additional constraint on  $g(x)$ :

$$f(x) - F \leq My$$

$$F - f(x) \leq M(1 - y)$$

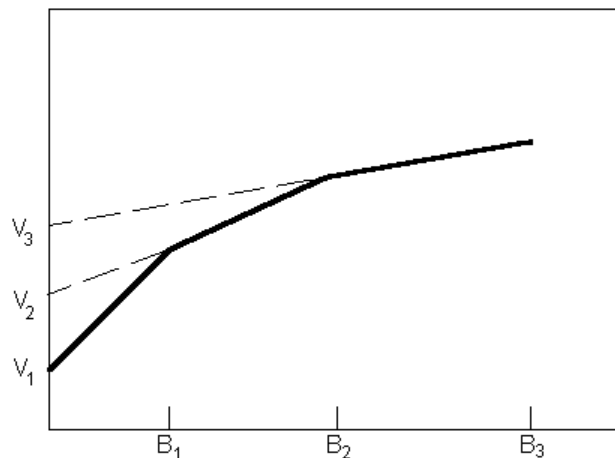
$$g(x) - G \leq M(1 - y)$$

If y is 0,  $f(x) \leq F$  is enforced, and the second and third constraints have no effect. If y is 1,  $f(x) \geq F$  and  $g(x) \leq G$  are enforced, and the first constraint has no effect. If  $f(x)$  and  $g(x)$  are linear functions of the variables, the constraints involving y remain linear, and the problem can be solved with Branch & Bound and the Simplex Solver.

## Using Piecewise-Linear Functions

Many problems involve “stepped” price schedules or quantity discounts, where you might at first expect that a non-smooth function such as CHOOSE or LOOKUP is required to express the relationship. You might be surprised to learn that you can instead use linear functions and binary integer variables to express the relationship.

For example, you might be purchasing parts from a vendor who offers discounts at various quantity levels. The graph below represents such a discount schedule, with three prices and three “breakpoints.” You have a decision variable x representing the quantity to order.



The three prices (slopes of the line segments) are  $c_1$ ,  $c_2$  and  $c_3$ .  $V_1$  represents a fixed initial cost;  $V_2$  and  $V_3$  are also constant in the problem and can be computed from:

$$V_2 = V_1 + c_1 * B_1 - c_2 * B_1$$

$$V_3 = V_2 + c_2 * B_2 - c_3 * B_2$$

In the model, the variable  $x$  is replaced by three variables  $x_1$ ,  $x_2$  and  $x_3$ , representing the quantity ordered or shipped at each possible price. Also included are three 0-1 or binary integer variables  $y_1$ ,  $y_2$  and  $y_3$ . Since you want to minimize costs, the objective and constraints are:

Minimize  $V_1 * y_1 + V_2 * y_2 + V_3 * y_3 + c_1 * x_1 + c_2 * x_2 + c_3 * x_3$

Subject to  $x_1 \leq B_1 * y_1$ ,  $x_2 \leq B_2 * y_2$ ,  $x_3 \leq B_3 * y_3$

If the cost curve is concave as shown above, this is sufficient; but if the function is non-concave (it may vary up and down), additional “fill constraints” are needed:

$$y_1 + y_2 + y_3 = 1$$

$$x_1 \leq B_1 * y_2$$

$$x_2 \leq B_2 * y_3$$

This approach is called a “piecewise-linear” function. It can be used in place of a CHOOSE or LOOKUP function, and it results in a linear integer model instead of a difficult-to-solve non-smooth model. Piecewise-linear functions can also be used to approximate a smooth nonlinear function, by using line segments with slopes matching the gradient of the nonlinear function at various intermediate points.

---

## Organizing Your Model for Fast Solution

This section describes ways you can organize your model so that the Premium Solver and Premium Solver Platform can analyze it more efficiently. Most of this section is devoted to an in-depth discussion of “fast problem setup” for *linear* and *quadratic* models (possibly with integer variables); it is not applicable to *nonlinear* and *non-smooth* models. Because the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform has largely superseded this form of fast problem setup, this section is relevant primarily for (i) the Premium Solver, where the Interpreter is not available and (ii) very large LP models (100,000 variables or more), where memory required for the Interpreter may be greater than available RAM and cause swapping to disk.

### Fast Problem Setup

In the Premium Solver, if your model is linear or quadratic, you may find that the Solver spends most of its time with “Setting Up Problem ...” on the Excel status bar, then speeds through the “Trial Solutions” very quickly. The setup time is required to extract the LP coefficients of the problem functions by recalculating the worksheet. (As described in more depth in the last section of the chapter “Analyzing and Solving Models,” the LP coefficients are the first partial derivatives of the problem functions with respect to the variables, and they are obtained by the method of *finite differencing*, which requires  $n + 1$  recalculations if there are  $n$  decision variables.)

Much of this setup time can be avoided if you write the formulas for the objective function and *all* of the constraint left hand sides using the functions recognized for *fast problem setup*: SUM, SUMPRODUCT, DOTPRODUCT, QUADPRODUCT and MMULT. This may require some work on your part to revise a model you have already constructed, but you will be rewarded with a 5- to 100-fold speed improvement

in setup time, compared to the time taken by the Premium Solver when finite differencing is used.

You can always express a linear or quadratic programming problem using these functions for the objective and all of the constraints, although you may need to introduce new sets of cells to hold the calculated coefficients so that these cells can be referenced by one of the fast problem setup functions. For more information on linear and quadratic functions, see the sections on “Linear Functions” and “Quadratic Functions” in the chapter “Solver Models and Optimization.” In the Support section of Frontline Systems Website at [www.solver.com](http://www.solver.com), you can find an example of the process of converting an existing LP model into fast problem setup form.

The Polymorphic Spreadsheet Interpreter in the Premium Solver Platform uses the techniques of *automatic differentiation* to obtain the LP coefficients faster and more accurately than they can be obtained via finite differencing. Because the Interpreter handles almost every kind of Excel formula and built-in function, you don't have to do the work of designing your model – or revising an existing model – to use only the small set of functions recognized for fast problem setup.

Fast problem setup is still available in the Premium Solver Platform as a specialized method of extracting the constant Jacobian matrix (the LP coefficients) of a linear or quadratic problem, and the constant Hessian matrix (the QP coefficients) of a quadratic objective function – but it is used only if you check the Fast Setup box or choose Solve With = Excel Interpreter in the Solver Model dialog. If your LP or QP model is very large, defining it in fast problem setup format may still save time compared to use of the Polymorphic Spreadsheet Interpreter – but the advantage is not nearly as great as the 5- to 100-fold speed improvement mentioned above.

The following subsections describe the functions supported for fast problem setup, and the form of the function arguments that you must use to ensure that they are recognized for fast setup purposes.

### ***The SUM Function***

The simplest case of a function recognized for fast problem setup is a formula such as =SUM(C1:C10) where C1 through C10 are decision variables. An example of the use of SUM to define constraints can be found in the “Shipping Routes” sheet in the SOLVSAMP.XLS workbook included with Microsoft Excel. Note that a SUM of decision variables is a linear function where all of the coefficients are 1. To be recognized in fast problem setup, your formula must consist *only* of =SUM(cells) (with no constants) where every cell referenced is a decision variable. You may use absolute or relative references or defined names in the arguments to SUM.

### ***The SUMPRODUCT Function***

The SUMPRODUCT function is documented in Microsoft Excel online Help. It returns the sum of the products of corresponding elements in its two arguments, which is exactly the form of a linear function:

$$a_1x_1 + a_2x_2 + \dots + a_nx_n$$

SUMPRODUCT requires that its two arguments refer to the same *number* of cells in the same *orientation* (either a row, a column or a rectangular area of cells). Only single selections, not multiple selections, are permitted in the arguments. If SUMPRODUCT is used in an array formula (see below), it will return the same value in every resulting array element, which is usually not the result you want. (The DOTPRODUCT function, described below, has more flexible arguments and is far more useful in array formulas.) To be recognized in fast problem setup, your formula

must consist *only* of =SUMPRODUCT(*cell range, cell range*) where one of the cell ranges consists *entirely* of decision variables, and the other cell range consists *entirely* of cells that are constant in the Solver problem. You may list the arguments in either order, using absolute or relative references or defined names.

### Other Functions for Fast Problem Setup

Use of the MMULT function is illustrated below under “Using Array Formulas.” To be recognized in fast problem setup, your formula must follow the same rules as for SUMPRODUCT: It must consist *only* of =MMULT(*cell range, cell range*) where one cell range specifies the decision variables, and the other cell range specifies the corresponding coefficients.

The DOTPRODUCT and QUADPRODUCT functions are described in their own sections below. To be recognized in fast problem setup, your usage of these two functions must follow the same rules as for SUMPRODUCT.

To qualify as a quadratic programming (QP) problem – which can be solved efficiently by the LP/Quadratic Solver engine – only the *objective function* (not any of the constraints) may use QUADPRODUCT, or any other quadratic form.

## Using Array Formulas

Optimization models in algebraic form can often be expressed more compactly using indexing and summation notation. For example, the five constraints in EXAMPLE1 could be written as:

$$\sum_{j=1}^3 a_{ij} x_j, i=1, \dots, 5$$

The SUMPRODUCT function corresponds to the summation expression above for one constraint. The *entire set* of five constraint formulas could be defined with the array form of the DOTPRODUCT function (described in detail later in this section). In EXAMPLE1, you would select cells C11 to C15 and “array-enter” the following formula:

{=DOTPRODUCT(D9:F9,D11:F15)}

The braces above are not typed, but they appear when you “array-enter” the formula by pressing CTRL-SHIFT-ENTER instead of just ENTER. If you aren’t familiar with array formulas in Microsoft Excel, you can read about them in Excel’s online Help. They are one of the most useful features of Microsoft Excel.

If your LP model is “dense” and regular in form rather than “sparse,” you may wish to consider use of Microsoft Excel’s matrix built-in functions, such as MMULT which (when array-entered into a group of cells) yields the matrix product of its two operands. For example, the five constraints in EXAMPLE1 could be written in vector-matrix form as:

$$\mathbf{Ax} \leq \mathbf{b}$$

where **A** is the matrix of coefficients, **x** is the vector of decision variables and **b** is the vector of constraint right hand sides. In the Microsoft Excel formula language, the left hand side of this expression could be written as:

{=MMULT(\_A,TRANSPOSE(\_X))}

(The TRANSPOSE function is needed only to “match up” the orientation of the matrix \_A with the row vector \_X.) In worksheet EXAMPLE1, if you insert defined



names `_A` for the coefficients D11:F15 and `_X` for the variables D9:F9, then select cells C11:C15 and array-enter the above formula, it will compute the values of all five constraints.

The Polymorphic Spreadsheet Interpreter in the Premium Solver Platform recognizes most kinds of array formulas supported by Microsoft Excel. But (for rather technical reasons) the use of array formulas actually involves a speed *disadvantage* in the Premium Solver Platform when the coefficients are extracted via automatic differentiation.

If you're using the Premium Solver Platform, we recommend that you use array formulas where they make sense, and focus on making your model simple and easy to maintain. In a large model, you'll probably find that you want or need to use multiple tabular areas for the formulas that define your constraints, and it may be inconvenient or impractical to define entire constraint left hand sides with functions like MMULT and TRANSPOSE.

## Using the Add-in Functions

The Premium Solver and Premium Solver Platform define three new Excel functions: DOTPRODUCT, QUADPRODUCT and QUADTERM. These functions behave just like Excel built-in functions: You can use them in formulas in any spreadsheet (not only in Solver models). When you use the Insert Function... menu option, these functions will appear in the "Select a Function" or "Paste Function" list (classified as Math & Trig functions), and you'll be prompted with named edit fields for their arguments.

In addition, DOTPRODUCT and QUADPRODUCT are recognized for purposes of fast problem setup as described earlier in this chapter. They are also recognized by the Interpreter in the Premium Solver Platform. QUADTERM is used to define a term of a quadratic function; it isn't recognized for fast problem setup, but it is recognized by the Interpreter in the Premium Solver Platform.

### Using DOTPRODUCT

DOTPRODUCT is a generalized version of the built-in function SUMPRODUCT, and it is very useful for defining the objective function and constraints of linear programming problems. DOTPRODUCT is also recognized for fast problem setup as described above, provided that you follow the rules outlined earlier: Your formula must consist *only* of `=DOTPRODUCT(cell reference, cell reference)` where all of the cells in one of the cell references are decision variables, and all of the cells in the other cell reference are constant in the Solver problem. Each *cell reference* must be either an *individual selection* or a *defined name*, but the cell ranges specified by the two arguments need not have the same "shape" (row, column, or rectangular area).

For use in Excel and for purposes of fast problem setup, DOTPRODUCT will accept defined names that specify *multiple selections* for either of its arguments. For example, if you had designed a model like the "Maximizing Income" worksheet pictured in the chapter "Building Solver Models," where the decision variables consisted of *several* rectangular cell selections, you could still calculate the objective function for your model with *one* call to DOTPRODUCT. You would first select the cells (B14:G14, B15:E15, B16), then choose Insert Name Define..., type a name such as CertDepos, and click OK. Next, you might put the corresponding interest rate figures (1%, 1%, 1%, 1%, 1%, 1%, 4%, 4%, 9%) in cells I1 to I9. Then the formula: `=DOTPRODUCT(CertDepos,I1:I9)`

would calculate the objective function (total interest earned), and would be recognized for purposes of fast problem setup.

You could skip the defined name step and use the multiple cell selection directly, as in `=DOTPRODUCT((B14:G14,B15,E15,B16),I1:I9)`. But this syntax is recognized for purposes of fast problem setup only if the argument string is less than about 180 characters. The Polymorphic Spreadsheet Interpreter in the Premium Solver Platform accepts *only* single selections such as `I1:I9`, for both function arguments and defined names – so you cannot define this objective function with a single call to `DOTPRODUCT`. However, with the Interpreter you don't need to – you can define the objective as a sum of several `DOTPRODUCT` functions, or define it with `+` and `*` operators as in the original “Maximizing Income” worksheet.

`DOTPRODUCT` always processes its arguments in *column, row, area* order – in an individual selection it reads cells across columns, wrapping around to subsequent rows, and in a multiple selection it reads the individual cell selections in the order in which they are listed. For example, the formula:

`=DOTPRODUCT(A1:C2,D1:D6)`

will calculate as `=A1*D1+B1*D2+C1*D3+A2*D4+B2*D5+C2*D6`.

### ***The Array Form of DOTPRODUCT***

If `SUMPRODUCT` is used in an array formula, it returns a scalar (single number) result, which is returned in every cell of the array. However, if `DOTPRODUCT` is used (with the proper arguments) in an array formula, it returns an *array result*. You can use this capability to calculate the left hand sides of several constraints with a single array formula. In a sparse model where you'd like to use the built-in function `MMULT` to compute the constraint values, but the variables and constraints aren't laid out in a single matrix, you can use the array form of `DOTPRODUCT` instead.

Further, when you use the array form of `DOTPRODUCT`, the Premium Solver products will recognize this form and use it to process many constraints at once in problem setup. (The array form is recognized for fast problem setup, and it's also recognized by the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform.) If you can't use the array form, even the simple form of `DOTPRODUCT` will save time in problem setup.

`DOTPRODUCT` will return an array value when the number of cells in one of its arguments is an *even multiple* of the number of cells in its other argument. As an example, consider the calculation of parts used in the LP model `EXAMPLE1`. The decision variables are in cells `D9` to `F9` (3 cells), and the coefficients of the constraint left hand sides – the number of parts used for each product – are in cells `D11` to `F15` (`15=3*5` cells). We want to calculate the left hand sides of the constraints in cells `C11` to `C15`. To do this, we would first select the group of five cells `C11:C15` with the mouse. Then we would type:

`=DOTPRODUCT(D9:F9,D11:F15)`

completing the entry with `CTRL+SHIFT+ENTER` instead of just `ENTER`. The formula will display as `{=DOTPRODUCT(D9:F9,D11:F15)}` – the braces are added by Microsoft Excel when the formula is array-entered. With the cell values shown in `EXAMPLE1` prior to solution (e.g. 100 for each of the decision variables), this array formula will calculate 200 in `C11`, 100 in `C12`, 500 in `C13`, 200 in `C14` and 400 in `C15`. Hence, it will compute the same set of values as the array expression shown earlier: `{=MMULT(_A, TRANSPOSE(_X))}`.

Whether it is used in the simple form or the array form, `DOTPRODUCT` always processes its arguments in *column, row, area* order. In the array form, when the cells

in the “shorter” argument have all been processed and cells remain to be processed in the “longer” argument, DOTPRODUCT “wraps around” to the beginning of the “shorter” argument. In the example above, cell C11 calculates the value  $=D9*D11+E9*E11+F9*F11$ ; cell C12 computes  $=D9*D12+E9*E12+F9*F12$ ; and so on. Keep this rule in mind when you use the array form of DOTPRODUCT, and keep your spreadsheet layouts as simple as possible!

## Using QUADPRODUCT

The QUADPRODUCT function can be used to define the objective for quadratic programming problems in a single function call, as required for fast problem setup.

Recall from the section “Quadratic Functions” in the chapter “Solver Models and Optimization” that a quadratic function is a sum of terms, where each term is a (positive or negative) constant (called a *coefficient*) multiplied by a *single variable* or the *product of two variables*. This means that in order to represent the most general quadratic function, we might have a coefficient for each instance of a single variable, and a coefficient for each possible *pair* of two variables. The QUADPRODUCT function is designed to supply coefficients for each single variable and each pair of variables, in a manner similar to SUMPRODUCT and DOTPRODUCT.

You supply the arguments of QUADPRODUCT as shown below:

$=\text{QUADPRODUCT}(\text{variable cells}, \text{single coefficients}, \text{pair coefficients})$

The first argument must consist entirely of decision variable cells. The second and third arguments must consist entirely of cells whose values are *constant* in the optimization problem; if these cells contain formulas, those formulas must not refer to any of the decision variables. The second argument supplies the coefficients to be multiplied by each single variable in the first argument, using an element-by-element correspondence. The third argument supplies the coefficients to be multiplied by each *pair* of variables drawn from the first argument. Hence, if there are  $n$  cells in the first argument, there must be  $n^2$  cells in the third argument. If the variables are represented by  $x_1, x_2, \dots, x_n$ , the single coefficients by  $a_1, a_2, \dots, a_n$ , and the pair coefficients by  $c_1, c_2, \dots, c_N$  where  $N = n^2$ , QUADPRODUCT computes the function:

$$\sum_{i=1}^n \sum_{j=1}^n c_{n(i-1)+j} x_i x_j + \sum_{j=1}^n a_j x_j$$

The pairs are enumerated starting with the first cell paired with itself, then the first cell paired with the second cell, and so on. For example, if the first argument consisted of the cells A1:A3, there should be nine cells in the third argument, and the values in those cells will be multiplied by the following pairs in order: A1\*A1, A1\*A2, A1\*A3, A2\*A1, A2\*A2, A2\*A3, A3\*A1, A3\*A2, and A3\*A3. The value returned by QUADPRODUCT is the sum of all of the coefficients multiplied by their corresponding single variables or pairs of variables.

Multiple selections can be used for each argument of QUADPRODUCT, subject to the same considerations outlined above for DOTPRODUCT: You can use the general syntax for multiple selections in Microsoft Excel, but defined names are needed for purposes of fast problem setup, and multiple selections are not accepted by the Polymorphic Spreadsheet Interpreter. (For an easier way to define a quadratic objective that *is* accepted by the Interpreter in the Premium Solver Platform, see the discussion of the QUADTERM function below.)

A common application of quadratic programming is to find an “efficient portfolio” of securities – often called *portfolio optimization*. The worksheet EXAMPLE4 in the Examples.xls workbook, which is copied to your installation folder when you

Premium Solver product is installed, illustrates portfolio optimization using the Markowitz method. This model, shown below, uses the QUADPRODUCT function to compute the variance of a portfolio of five securities, and uses this quantity as the objective to be minimized, subject to a constraint that gives a lower limit on the portfolio return. Because the *objective* is a quadratic function, and the *constraints* (including the lower bound on return) are all linear functions, this Solver model is a quadratic programming (QP) problem which can be handled by the LP/Quadratic Solver in the Premium Solver Platform.

	A	B	C	D	E	F	G	H	I
1	<b>Portfolio Optimization - Markowitz Method</b>								
2	This model finds the optimal allocation of funds to stocks that minimizes the portfolio risk, measured by								
3	portfolio Variance (a quadratic function) at cell I17 -- computed via a custom QUADPRODUCT function.								
4	If you see #NAME? in I17, use <b>Tools Add-Ins...</b> and check the box next to Frontline's Mathematical								
5	Functions. This quadratic programming (QP) model can be solved with the GRG Nonlinear Solver, or								
6	more efficiently in the Premium Solver Platform with the LP/Quadratic Solver or the SOCP Barrier Solver.								
8		<b>Stock 1</b>	<b>Stock 2</b>	<b>Stock 3</b>	<b>Stock 4</b>	<b>Stock 5</b>	<b>Total</b>		
9	<b>Portfolio %</b>	20.00%	20.00%	20.00%	20.00%	20.00%	100.00%		
10	<b>Expected Return</b>	7.00%	8.00%	9.50%	6.50%	14.00%			
11	<b>Linear QP Terms</b>	0	0	0	0	0			
13	<b>Variance/Covariance Matrix</b>								
14		<b>Stock 1</b>	<b>Stock 2</b>	<b>Stock 3</b>	<b>Stock 4</b>	<b>Stock 5</b>			
15	<b>Stock 1</b>	2.50%	0.10%	1.00%	-0.50%	1.00%			
16	<b>Stock 2</b>	0.10%	4.00%	-0.10%	1.20%	-0.85%			
17	<b>Stock 3</b>	1.00%	-0.10%	1.20%	0.65%	0.75%	<b>Variance</b>	1.25%	
18	<b>Stock 4</b>	-0.50%	1.20%	0.65%	8.00%	1.00%	<b>Std. Dev.</b>	11.17%	
19	<b>Stock 5</b>	1.00%	-0.85%	0.75%	1.00%	7.00%	<b>Return</b>	9.00%	

The formula using QUADPRODUCT at cell I17 reads as follows:

=QUADPRODUCT(Allocations,B11:F11,Stock\_Covariances)

If you select Premium Solver... from the Tools menu, the Solver Parameters dialog will appear with the entries shown below. (We've checked the "Assume Non-Negative" box in the Solver Options dialog.)

Notice the use of defined names in this model: "Allocations" is defined as the cell range B9:F9, and "Stock\_Covariances" is defined as the rectangle B15:F19. Since in this problem the objective function consists only of the quadratic terms (pairs of variables) with no linear terms (single variables), we supply a row of zeroes for the second argument to QUADPRODUCT.

**Solver Parameters V7.0**

Set Cell: Portfolio\_Variance

Equal To: ☐ Max ☒ Min ☐ Value Of: 0

By Changing Variable Cells: Allocations

Subject to the Constraints:

Portfolio\_Return >= 0.095  
Total\_Portfolio = 1

Standard LP/Quadratic

Buttons: Solve, Close, Model, Options, Add, Variables, Change, Reset All, Delete, Help

## Using QUADTERM

The QUADTERM function can be used to define a term of an overall quadratic function. Its purpose is to make it easier for you to define a quadratic objective or constraint of more than 256 decision variables. You are unlikely to need this function in Excel 2007, which allows 16,384 columns per worksheet. But since Microsoft Excel 2000-2003 allows a maximum of 256 columns, you cannot easily define a “square matrix” of coefficients with more than 256 rows and columns. Instead, you can break up the matrix into four, nine or more submatrices, each one up to 256 rows and columns. You then use the QUADTERM function to compute the contribution to the overall quadratic function from each submatrix. In your objective function cell, you sum up the QUADTERM functions, plus a SUMPRODUCT or DOTPRODUCT function if there is a linear term in the objective.

You supply the arguments of QUADTERM as shown below:

`=QUADTERM(variables1, variables2, pair coefficients)`

The first and second arguments should each be a range of decision variable cells (both arguments may define the same range, or they may define different ranges of cells). The third argument defines coefficients that will be multiplied by each pair of cells in the ranges *variables1* and *variables2*. If there are  $n$  cells in the first argument, there must be  $n$  cells in the second argument, and  $n^2$  cells in the third argument. If the first argument is represented by  $x_1, x_2, \dots, x_n$ , the second argument by  $y_1, y_2, \dots, y_n$ , and the coefficients by  $c_1, c_2, \dots, c_N$  where  $N = n^2$ , QUADTERM computes the function:

$$\sum_{i=1}^n \sum_{j=1}^n c_{n(i-1)+j} x_i y_j$$

As with QUADPRODUCT, the pairs are enumerated starting with the first cell in the first argument paired with the first cell in the second argument, then the first cell in the first argument paired with the second cell in the second argument, and so on. The value returned by QUADTERM is the sum of all of the coefficients multiplied by their corresponding pairs of variables.

As described earlier, the QUADTERM function can be used in any Microsoft Excel worksheet, with or without the Solver. It is not recognized for purposes of fast problem setup, but it is recognized by the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform. As an example, you might define the objective of a 512-variable problem, with decision variables X1:X512, using four submatrices and a formula `=QUADTERM(X1:X256, X1:X256, A601:IV856) + QUADTERM(X1:X256, X257:X512, A901:IV1156) + QUADTERM(X257:X512, X1:X256, A1201:IV1456) + QUADTERM(X257:X512, X257:X512, A1501:IV1756)`.



# Simulation Optimization with Risk Solver Engine

---

## Monte Carlo Simulation and Optimization

With the Premium Solver Platform and **Risk Solver Engine**, you can define and solve *simulation optimization* problems, where the objective and constraints in your model may depend on both your *decision variables* and on *uncertain variables* – sometimes called „random variables“ or „assumptions“ – that represent uncertain factors not under your control.

Risk Solver Engine is licensed separately from the Premium Solver Platform. Used alone, it enables you to create and run risk analysis models using Monte Carlo simulation. You can try Risk Solver Engine free of charge with a 15-day trial license: Just visit [www.solver.com](http://www.solver.com), login and download the RSESetup program.

A Risk Solver Engine model can include uncertain variable cells, whose values are defined by probability distributions. You specify the distributions via function calls such as `=PsiNormal(10,1)`. Each time such a function is called, it returns a different value – much like the `Rand()` function in Excel. You can use the cells containing these random variables in any formula in your model.

On your command, or (if desired) *each time you change a number* on the worksheet, Risk Solver Engine will perform a **Monte Carlo simulation** with thousands of trials. On each trial, the random variable cells will have different values (a “sample” from each variable’s probability distribution), and all the formula cells in your model will be recalculated with these values.

Trials and summary statistics, such as the mean value and standard deviation across all the trials, are accumulated for the formula cells you designate (sometimes called „forecasts“). You can access these statistics in regular Excel formulas. And in the Premium Solver Platform, you can use these summary statistics in the objective and constraints of your optimization model.

When you define a simulation optimization model and click **Solve**, the Premium Solver Platform performs a Monte Carlo simulation through Risk Solver Engine **on each Trial Solution** of the optimization. Doing this is usually an order of magnitude faster than alternative products for simulation optimization. You can for example maximize the expected (mean) value of Net Profit – which may be uncertain, i.e. dependent on random variables – subject to constraints that may also depend on random variables.

---

# Defining a Simulation Optimization Model

Creating a simulation optimization model using the Premium Solver Platform and Risk Solver Engine is straightforward. You follow these steps:

1. Define decision variable cells (such as A1), using either the Solver Parameters dialog or the `PsiVar()` function. These are factors that *are* under your control – you (or the Solver) will decide what values they should have.
2. Define uncertain variable cells (such as A2), that contain formulas calling the PSI Distribution functions supplied by Risk Solver Engine – for example `PsiUniform()` and `PsiNormal()`. These are factors that *are not* under your control.
3. Build your model, using cell formulas that may depend on the decision variables, uncertain variables, or both.
4. Each cell (such as B1) containing a formula that depends on uncertain variables (say  $=A1+2*A2$ ) represents thousands of trial values, generated during each Monte Carlo simulation by *sampling* different values for A2 and computing  $=A1+2*A2$ .
5. In other cells (such as C1), define the summary statistics you want, using functions such as `PsiMean(B1)` or `PsiStdDev(B1)`. You may use formulas to compute further values based on these summary statistics.
6. Define your objective and constraints for optimization. These may be cell formulas that depend only on the decision variables, depend on the uncertain variables through PSI Statistics functions, or depend on both.

An objective or constraint can depend on a cell containing a PSI Statistic function such as C1 above, but not directly on a cell such as B1 above. The cell containing a PSI Statistic function has a single value; the cell B1 represents many different values.

## Using PSI Functions for Simulation

With Risk Solver Engine, the *uncertain variables* you define may have values drawn from your choice of more than 40 PSI Distribution functions – from **PsiBernoulli()** to **PsiWeibull()**.

You can also draw values from predefined *Certified Distributions*, by simply referring to these distributions in **PsiSip()** and **PsiSlurp()** functions. You can easily shift, truncate and correlate probability distributions using PSI Property functions such as **PsiShift()**, **PsiTruncate()** and **PsiCorrMatrix()**.

To define results that depend on the uncertainties in your model, you create ordinary Excel formulas that compute functions of your uncertain variables. You can optionally designate the cells containing computed results that you want to use (in charts or summary statistics) with the function **PsiOutput()**.

You obtain summary statistics across all simulation trials for such cells by referring to them in formulas using **PsiMean()**, **PsiVariance()**, **PsiPercentile()**, **PsiCVaR()**, and other PSI Statistics functions.

See the chapter “PSI Function Reference” in the Risk Solver Engine User Guide for a complete list of PSI functions you can use. There are more than 70 functions in all.



## PSI Functions in Objectives and Constraints

Once you have cell formulas using PSI Statistics functions like the ones just mentioned, you can use these cells – or other formulas that depend on them – as your objective (Set Cell) or as the left hand sides of constraints.


For example, if you have a worksheet that defines uncertain market demand for your products, and computes sales, inventory levels and Net Profit, you could define and solve an optimization problem that maximizes Net Profit, subject to constraints on maximum (say, 90<sup>th</sup> percentile) or minimum (10<sup>th</sup> percentile) inventory levels.

Such problems are typically *non-smooth* optimization problems that may be very difficult to solve. But the combination of the Premium Solver Platform and Risk Solver Engine gives you unprecedented power and speed to tackle these problems.

---

## Solving a Simulation Optimization Model

Solving a simulation optimization model using the Premium Solver Platform and Risk Solver Engine is also straightforward. Follow these steps:

1. Activate **Interactive Simulation**  by clicking the **light bulb** button on the Risk Solver Engine toolbar.
2. Select **Tools Premium Solver** to display the Solver Parameters dialog, and click the **Solve** button.

In Version 7.0, when you solve a problem with Interactive Simulation active, the **Solve With** option is effectively set to **Excel Interpreter**, regardless of the setting in the Solver Model dialog. This means *only* that Excel is used to calculate the objective and constraints for each Trial Solution of the optimization. But in order to calculate the objective and constraints, a Monte Carlo simulation must be performed – which accounts for *most* of the computation time – and this is done at high speed by the PSI Interpreter in Risk Solver Engine.

## Solver Engines and Options

A simulation optimization model is almost always *nonlinear*, and is usually *non-smooth*, because your objective and/or constraints depend on PSI Statistics functions, that in turn depend on PSI Distribution functions.

Hence you should use the **Evolutionary Solver**, or the **GRG Nonlinear Solver** with the **Multistart Search** box checked, to solve a simulation optimization model. You can also use plug-in Solver engines such as the OptQuest Solver, and the Large-Scale GRG, Large-Scale SQP, and KNITRO Solvers with their Multistart Search options.

## A Project Selection Example

An example problem where simulation optimization may be helpful is in capital budgeting, where we must select among various project proposals seeking funding, within a limited budget. Suppose that each project has a given probability of success or failure, and a range of uncertain future cash flows if it is successful. What is known with certainty is the amount needed to fund each project this year. We have \$2.5 million worth of first-year project funding requests, but a budget of only \$1.5 million – so we must select a subset of the projects. This is illustrated in the ProjectSelect.xls workbook, shown on the next page.

ProjectSelect.xls							
A	B	C	D	E	F	G	H
1	Project Selection with a Budget Constraint						
2							
3	Project Number	Cash Flow if Successful	Chance of Success	Expected Cash Flow	Initial Investment	Expected Cash - Invest	Decision to Include
4	1	\$576,040	100%	\$576,040	\$325,000	\$251,040	1
5	2	\$654,697	0%	\$0	\$450,000	-\$450,000	1
6	3	\$1,217,050	100%	\$1,217,050	\$550,000	\$667,050	1
7	4	\$603,576	100%	\$603,576	\$300,000	\$303,576	1
8	5	\$489,660	100%	\$489,660	\$150,000	\$339,660	1
9	6	\$585,031	100%	\$585,031	\$250,000	\$335,031	1
10	7	\$355,769	0%	\$0	\$150,000	-\$150,000	1
11	8	\$514,673	100%	\$514,673	\$325,000	\$189,673	1
12							
13			Total Investment		2,500,000		
14			Budget Constraint		1,500,000		
15							
16			Total Cash Flow		1,486,029		
17			Expected Total CF		2,000,254		
18							

## The Simulation Model

To model the “Cash Flow if Successful,” where only a minimum, most likely, and maximum cash flow are known, we’ve used the `PsiTriangular()` function. For example, the formula in C4 is `=PsiTriangular(400000,500000,900000)`.

To model the “Chance of Success,” we’ve used the function `=PsiBinomial(1,9)` for each project. On each trial, this distribution returns 1 with probability 90% and 0 with probability 10%. We multiply the “Cash Flow if Successful” and the “Chance of Success” to obtain the “Expected Cash Flow.”

Cell F16 computes the total cash flow as the `SUMPRODUCT` of the selected projects with their (uncertain) individual cash flows multiplied by their chance of success. This changes on each simulation trial, as you can see by pressing F9.

Cell F17 contains `=PsiMean(F16)`: It computes the *mean* (i.e. average or expected) value of the total expected cash flow. Click the **light bulb** on the Risk Solver Engine toolbar to see this mean value for 1,000 trials, each time you press F9.

## The Optimization Model

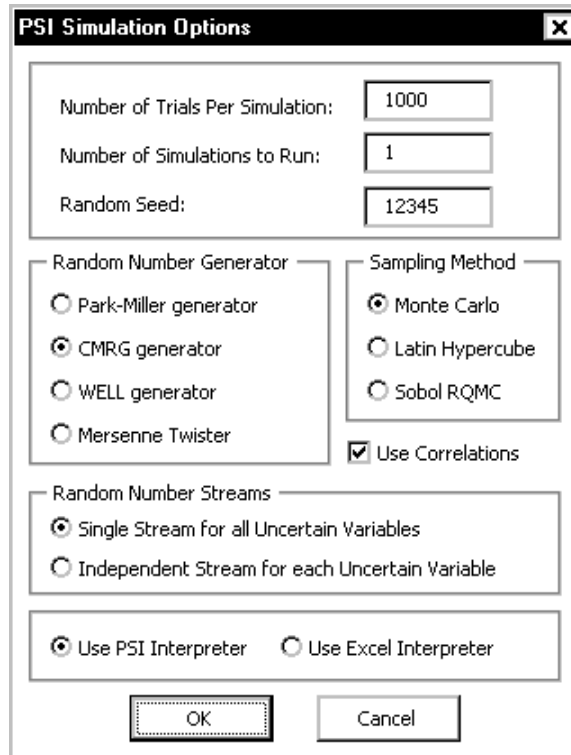
Cells H4:H11 are our **decision variables**, which should be 1 to fund the project or 0 to reject it. These will be binary integer variables.

Cell F13 computes our total investment as the `SUMPRODUCT` of the selected projects with their (certain) initial costs. This will be **constrained** to be less than or equal to our budget of \$1.5 million at cell F14.

Cell F17 – the mean or expected value of the total cash flow – is our **objective**, which we wish to maximize.

## Setting the Random Seed

Before running the optimization, we’ll set a fixed random number seed for each simulation. This means that the same random sample will be used by Risk Solver Engine on each Trial Solution of the optimization process; hence, the Solver can adjust the decision variables and observe different results for the problem functions without the “noise” of constantly changing total values due only to the random sample used in the simulation. To do this, click the **tool** icon on the Risk Solver Engine toolbar, which displays the dialog on the next page.



**PSI Simulation Options**

Number of Trials Per Simulation: 1000

Number of Simulations to Run: 1

Random Seed: 12345

Random Number Generator:

- ☐ Park-Miller generator
- ☒ CMRG generator
- ☐ WELL generator
- ☐ Mersenne Twister

Sampling Method:

- ☒ Monte Carlo
- ☐ Latin Hypercube
- ☐ Sobol RQMC

☒ Use Correlations

Random Number Streams:

- ☒ Single Stream for all Uncertain Variables
- ☐ Independent Stream for each Uncertain Variable

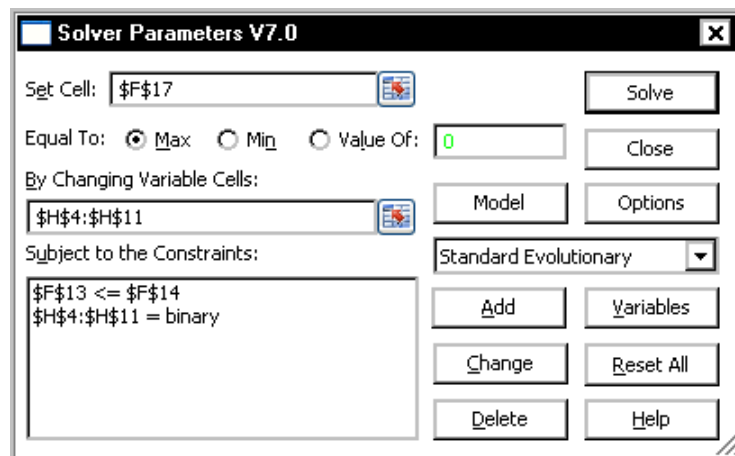
☒ Use PSI Interpreter ☐ Use Excel Interpreter

OK Cancel

In the Random Seed edit box, type a number such as 12345, then click OK. Any nonzero number can be used; each number will yield different simulation trials.

### ***Using the Evolutionary Solver***

Next, we set up the model so that the Solver Parameters dialog looks like this:



**Solver Parameters V7.0**

Set Cell: \$F\$17

Equal To: ☒ Max ☐ Min ☐ Value Of: 0

By Changing Variable Cells: \$H\$4:\$H\$11

Subject to the Constraints:

- \$F\$13 <= \$F\$14
- \$H\$4:\$H\$11 = binary

Standard Evolutionary

Add Change Delete Variables Reset All Help

Solve Close Model Options

As described above, we want to maximize the mean or expected value of the total cash flow at F17, subject to the constraint that the total initial investment at F13 is less than or equal to our budget of \$1.5 million at F14. The decision variables are constrained to be binary integer – either 0 or 1.

When you click **Solve**, the Evolutionary Solver will find a solution like the one shown on the next page – usually in few seconds – where we select projects 1, 2 and 3 for an expected value of total cash flow of about \$1 million.

Project Selection with a Budget Constraint							
Project Number	Cash Flow if Successful	Chance of Success	Expected Cash Flow	Initial Investment	Expected Cash - Invest	Decision to Include	
1	\$731,121	100%	\$731,121	\$325,000	\$406,121	1	
2	\$1,058,968	100%	\$1,058,968	\$450,000	\$608,968	1	
3	\$810,750	100%	\$810,750	\$550,000	\$260,750	1	
4	\$462,177	0%	\$0	\$300,000	-\$300,000	0	
5	\$617,109	100%	\$617,109	\$150,000	\$467,109	0	
6	\$451,513	100%	\$451,513	\$250,000	\$201,513	0	
7	\$547,922	100%	\$547,922	\$150,000	\$397,922	0	
8	\$552,663	0%	\$0	\$325,000	-\$325,000	0	
Total Investment				1,325,000			
Budget Constraint				1,500,000			
Total Cash Flow				1,275,839			
Expected Total CF				1,070,446			

If you use the **tool** icon and the Simulation Options dialog to set the **Random Seed** back to 0, and click F9 a few times, you will see that the objective at F17 varies from run to run (since a different random sample is drawn each time), and is often a little less than the objective found during the optimization.

This illustrates the tradeoff of a fixed vs. variable random number seed – a fixed seed enables the Solver to more rapidly find improved solutions, since it is not dealing with random “noise,” but the solution it finds is optimized for the set of 1,000 trials determined by the initial seed, and may not be as good for a different set of 1,000 trials. But because the Solver is maximizing the mean value, the solution is usually quite acceptable.

You can instead run the Solver with the seed set at 0, which means that a different random sample is used on each simulation run. This makes it harder for the Solver to find improved solutions, because the “random noise” makes it hard to compare successive Trial Solutions. Usually, the Solver will require more time to find a good solution, but this solution may be more “robust” on different sets of simulation trials.

Another option is to use a fixed seed and increase the number of trials in the Simulation Options dialog – thanks to the speed of Risk Solver Engine, you can often afford to run 2,000, 5,000, or more trials per simulation.

Version 7.0 focuses on the high-performance simulation optimization capability provided by the combination of the Premium Solver Platform and Risk Solver Engine. Future Frontline Systems products will go much further to support your ability to solve problems that call for good or optimal decisions in situations involving uncertainty, where you can quantify the decisions and the uncertainty in an Excel model.

# Diagnosing Solver Results

---

## If You Aren't Getting the Solution You Expect

This chapter will help you understand the results reported by the Solver and diagnose problems with your Solver models. *The most important step you can take* to deal with potential Solver problems is to start out with a clear idea of the type of optimization model you are creating, how it relates to well-known problem types, and whether yours is a linear, quadratic, smooth nonlinear or non-smooth optimization problem – as discussed in previous chapters. If you then build your model in a *well-structured, readable and efficient form* – as outlined at the beginning of the chapter “Building Large-Scale Models” – diagnosing problems should be relatively easy. But at times you may be “surprised” by the results you get from the Solver.

If the Solver stops with a solution (set of values for the decision variables or Changing Cells) that is different from what you expect, or what you believe is correct, follow the suggestions below. You can usually narrow down the problem to one of a few possibilities.

**Check the Solver Result Message shown in the Solver Results dialog.** Users sometimes contact Frontline Systems about “wrong solutions”, but they don't know which Solver Result Message they received – this is crucial to diagnosing the problem. Read carefully the discussion of your Solver Result Message in the following sections.

Consider carefully the possibility that the solution found by the Solver is correct, and that your expectation is wrong. This may mean that what your model actually says is different from what you intended.

In the Premium Solver Platform, many Solver Result Messages from the Polymorphic Spreadsheet Interpreter refer to a specific problem at a specific cell address in your worksheet. You may have to modify the formula in this cell to use the Interpreter, or else you'll have to set the Solve With option to Excel Interpreter in the Solver Model dialog.

Check the “Show Iteration Results” box in the Solver Options dialog and re-solve. The Solver will pause with the message “Solver paused, current solution values displayed on worksheet.” Click Continue to see the path towards the solution taken by the Solver.

If you receive the message “Solver could not find a feasible solution,” select and examine the Feasibility Report to determine which subset of the constraints is making the problem infeasible.

If you receive the message “The linearity conditions required by this Solver engine are not satisfied,” select and examine the Linearity Report to determine which functions or variables in your model are not linear. Even better – if you have the Premium Solver Platform – follow the steps in “Diagnosis Tab: Analyzing Model Exceptions” in the chapter “Analyzing and Solving Models” to pinpoint the exact cell formulas that aren’t linear.

Read the section in this chapter on “Problem with Poorly Scaled Models.” In the Premium Solver Platform, if you see the Scaling Report listed in the Reports list box of the Solver Results dialog, select this report and examine its contents for strong clues about poor scaling in your model.

Later sections of this chapter discuss characteristics and limitations of the GRG Solver for smooth nonlinear problems, the Interval Global Solver for global optimization of smooth nonlinear problems, and the Evolutionary Solver for non-smooth problems. Read the section(s) most relevant for the type of problem you are solving.

---

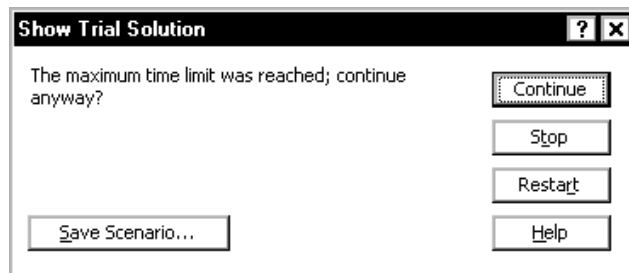
## During the Solution Process

When you click the Solve button in the Solver Parameters dialog, the solution process is started. When it completes, the Solver Results dialog appears, displaying a Solver Result Message, and giving you the option to save or discard the results and generate reports. If you are controlling the Solver via a VBA program, you will call the Problem.Solver.Optimize method or the SolverSolve function, and examine the integer result (OptimizeStatus value) that corresponds to a Solver Result Message.

You can interrupt the solution process at any time by pressing the ESC key. This will display the Show Trial Solution dialog, pictured below. The Show Trial Solution dialog also appears if you have checked the box Show Iteration Results in the Solver Options dialog, or if the Solver reaches a limit on the solution process (such as maximum time, iterations, subproblems, or integer or feasible solutions) that you have set via the Solver Options dialog.

### Choosing to Continue, Stop or Restart

At the time the Show Trial Solution dialog appears, your worksheet will contain the current values of the decision variables, objection function and constraints. (If you are using a VBA program to control the Solver, you can specify a function in your code to be called in lieu of displaying this dialog.) In this dialog (or via your VBA function) you can choose to *continue*, *stop*, or *restart* the solution process.



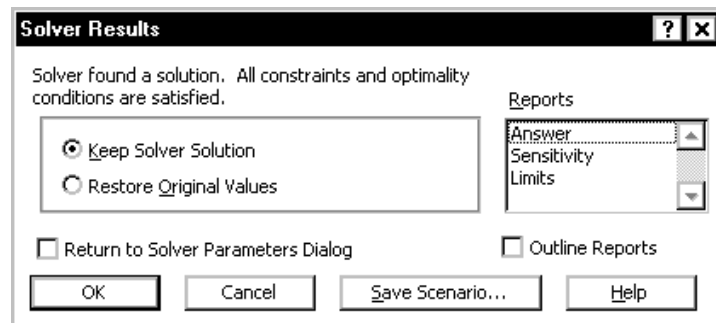
If you choose to *continue* the solution process, the limit (if any) that was reached is ignored, and the Solver continues to run. If you choose to *stop* the solution process, the Solver “cleans up” and then displays the Solver Results dialog, or returns from your VBA method or function call. If you choose to *restart* the solution process, the

Solver will “re-initialize” in a manner specific to the Solver engine being used, then continue to run. The LP/Quadratic Solver and the SOCP Barrier Solver treat the *restart* option as equivalent to the *continue* option. The nonlinear GRG Solver re-initializes its “curvature” information (approximate Hessian of the objective) and convergence test, and continues from the current Trial Solution. The Evolutionary Solver re-initializes its population of candidate solutions, replacing the worst half of the population with newly sampled points for greater diversity, and continues the solution process. The Interval Global Solver treats the *restart* option as equivalent to the *continue* option. Field-installable Solver engines handle this option in manners similar to those just described for linear, smooth nonlinear and non-smooth problems.

---

## When the Solver Finishes

When the solution process completes, the Solver Results dialog appears, displaying a Solver Result Message, as shown below. If you are controlling the Solver via a VBA program and you’ve called the `Problem.Solver.Optimize` method, the Solver object `OptimizeStatus` property holds a result code that corresponds to a Solver Result Message. If you’ve called the “traditional” `SolverSolve` function, this function will return the integer result code. At this point, your worksheet contains the *best* solution found – values for the decision variable cells, and calculated values for the objection function and constraints.



In this dialog (or by calling the `Problem.Solver.Report` method or the “traditional” `SolverFinish` function), you can select one or more reports, choose one of the options “Keep Solver Solution” or “Restore Original Values,” and optionally save the decision variable values in a named scenario by clicking on the `Save Scenario...` button. When you click OK, the reports will be produced. Clicking Cancel instead will discard the solution and cancel generation of the reports. The reports are further described in the chapter “Solver Reports.”

After the reports (if any) are produced, the Solver will return to worksheet Ready mode unless you’ve checked the box “Return to Solver Parameters Dialog.” When you check the “Return to Solver Parameters Dialog” box, it remains checked (until you change it) for the duration of your Excel session. To return to worksheet Ready mode, you can click the Close button in the Solver Parameters dialog, uncheck this box, or click Cancel in the Solver Results dialog.

## Standard Solver Result Messages

The LP/Quadratic Solver, SOCP Barrier Solver, nonlinear GRG Solver, Interval Global Solver, and Evolutionary Solver, and the Branch & Bound and multistart methods bundled with the Premium Solver Platform return the integer result codes and display the Solver Result Messages described in this section. Some of these

messages have a slightly different interpretation depending on which Solver engine you are using; see the explanations of each message, particularly for return code 0, “Solver found a solution.” Please note that the Branch & Bound and Multistart methods usually return result codes 14 through 17, documented later in this section.

Field-installable Solver engines are designed to return the same codes and display the same messages as the built-in Solver engines whenever possible, but they can also return custom result codes (starting with 1000) and display custom messages, as described in their individual documentation. The Interval Global Solver can return three of these custom result codes and messages, described at the end of this section.

#### **-1. A licensing problem was detected, or your trial license has expired.**

This message appears if a Premium Solver product cannot find its licensing information, if the licensing information is invalid, or if you have a time-limited evaluation license that has expired. *Click the Help button* for further information about the licensing problem. Please call Frontline Systems at (775) 831-0300, or send email to us at [info@solver.com](mailto:info@solver.com) for further assistance.

#### **0. Solver found a solution. All constraints and optimality conditions are satisfied.**

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 message, because they have no way of testing the solution for true optimality.

If you are solving a *linear* programming problem or a *convex quadratic* programming problem with the LP/Quadratic Solver, the Solver has found the *globally* optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective (Set Cell). 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, the Solver has found the *globally* optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective (Set Cell). 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 *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 (Set Cell). 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 using the Interval Global Solver for *global optimization* of a *smooth nonlinear* problem with no integer constraints, this means that the Solver has found the globally optimal solution: There is *no* other solution satisfying the constraints that has a better value for the objective. But this is *subject to limitations* due to the finite precision of computer arithmetic – discussed below in “Limitations on Global Optimization” – that can, in rare cases, cause the Solver to “miss” a feasible solution with an even better objective value.

If you are solving a *mixed-integer* programming problem (any problem with integer constraints) using a Premium Solver product, this message means that the Branch & Bound method has found a solution satisfying the constraints (including the integer



constraints) with the “best possible” objective value (but see the next paragraph). 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.

In the standard Microsoft Excel Solver, this message *also* appears for mixed-integer problems where the Solver stopped because the solution was within the range of the true integer optimal solution allowed by the Tolerance value in the Solver Options dialog (5% by default). In the Premium Solver products, when the Branch & Bound process stops due to a nonzero Tolerance without “proving optimality,” the message “Solver found an integer solution within tolerance. All constraints are satisfied” (result code 14) is displayed to distinguish this condition (see below).

## 1. Solver has converged to the current solution. All constraints are satisfied.

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 the Interval Global Solver, or a non-smooth problem with the Evolutionary Solver.

When the GRG Solver or the Interval Global Solver is being used, this message 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 in the Solver Options dialog. A *poorly scaled* model is more likely to trigger this stopping condition, even if the Use Automatic Scaling box in the Solver Options dialog is checked. 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 message 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 in the Solver Options dialog. 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. Solver cannot improve the current solution. All constraints are satisfied.

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 (“Solver found a solution”) and convergence (“Solver converged to the current solution”) have not yet been satisfied. The exact meaning depends on whether you are solving a smooth nonlinear problem with the GRG Solver, a global optimization problem with the Interval Global Solver, or a non-smooth problem with the Evolutionary Solver.

When the GRG Solver is being used, this message occurs very rarely. 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 Interval Global Solver is being used, this message is more common. It means that the Solver has not found an “improved global solution” (a feasible solution with an objective value better than the currently best known solution), in the amount of time specified by the Max Time without Improvement option in the Solver Options dialog. The reported solution is the best one found so far, but the search space has not been fully explored. For more information, see the discussion of “Interval Global Solver Stopping Conditions” below. If you receive this message, and you are willing to spend more solution time to have a better chance of “proving” global optimality, increase the value of the Max Time without Improvement option.

When the Evolutionary Solver is being used, this message 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 “Solver converged to the current solution” or “Solver cannot improve the current solution” if you let it run for long enough. If you believe that this message 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. Stop chosen when the maximum iteration limit was reached.

This message appears when (i) the Solver has completed the maximum number of iterations, or trial solutions, allowed in the Iterations box in the Solver Options dialog *and* (ii) you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value in the Iterations box, or click on the Continue button instead of the Stop button in the Show Trial Solution dialog. 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 message is relatively unlikely to appear. The Evolutionary Solver uses the Max Subproblems and Max Feasible Solutions options on the Limit Options dialog tab, and the Branch & Bound method (employed by the other Solver engines on problems with integer constraints) uses the Max Subproblems and Max Integer Solutions options on the Integer Options dialog tab, 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. The Set Cell values do not converge.

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 Set Cell may take on negative values without limit unless this is prevented by the constraints or bounds on the variables. Check the Assume Non-Negative box in the Solver Options dialog to impose  $\geq 0$  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 message, 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 message, 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. Solver could not find a feasible solution.

This message appears 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 Interval Global Solver, this message means that the Solver could find no feasible solutions after a systematic exploration of the search space. The Interval Global Solver is designed to “prove feasibility” as well as global optimality, but this is *subject to limitations* due to the finite precision of computer arithmetic – discussed below in “Limitations on Global Optimization” – that can, in rare cases, cause the Solver to “miss” a feasible solution.

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 value in the Solver Options dialog (which reduces the infeasibility penalty, thereby allowing the evolutionary algorithm to explore more “nearly feasible” points).

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 select the Feasibility Report, shown in the Reports list box when this message appears, and click OK. (This report can take time for the LP/Quadratic Solver or SOCP Barrier Solver, *more* time for the GRG Solver, and *much more* time for the Interval Global Solver; it is not available for the Evolutionary Solver.) For an example of using the Feasibility Report to diagnose an infeasible solution, see “The Feasibility Report” in the chapter “Solver Reports.”

## 6. Solver stopped at user's request.

This message appears only if you press ESC to display the Show Trial Solution dialog, and then click on the Stop button. If you are controlling the Solver from a VBA program, remember that the user may press ESC while your VBA program is running. You can write VBA code to disable the ESC key, or – better – define a VBA function that the Solver will call instead of displaying the Show Trial Solution dialog (an Evaluator in the object-oriented API, or a ShowRef function argument in the “traditional” SolverSolve function.). Be sure to test for this OptimizeStatus or integer result (6) in your VBA code, and take action appropriate for your application.

## 7. The linearity conditions required by this Solver engine are not satisfied.

In the standard Excel Solver, this message is worded “The conditions for Assume Linear Model are not satisfied,” and it can appear only when the Assume Linear Model box in the Solver Options dialog is checked. In the Premium Solver, this message appears if you’ve selected the linear Simplex Solver engine, but the Solver’s numeric tests to ensure that the objective and constraints are indeed linear functions of the decision variables were not satisfied.

In the Premium Solver Platform, this message appears if you’ve selected the LP/Quadratic Solver 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 and the Solver’s tests determine that the constraints or the objective are not linear or convex quadratic functions of the variables. To understand exactly what is meant by a linear or quadratic function, read the section “Functions of the Variables” in the chapter “Solver Models and Optimization.”

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 for nonlinear or non-smooth functions or operators applied to the decision variables. If you have the Premium Solver Platform, simply follow the steps in “Analyzing Model Exceptions” in the chapter “Analyzing and Solving Models” to pinpoint the exact cell formulas that aren’t linear. If you have the Premium Solver, select and examine the Linearity Report to determine which functions or variables in your model are not linear; see “The Linearity Report” in the chapter “Solver Reports.”

## 8. The problem is too large for Solver to handle.

This message – or the more specific message **Too many adjustable cells, Too many constraints, or Too many integer adjustable cells** – appears when the Solver determines that your model is too large for the Solver engine that is selected (in the Solver engine dropdown list) at the time you click Solve. 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 in order to proceed.

You can check the size (the number of variables, constraints, bounds, and integers) of the problem you have defined, and compare it to the size limits of the Solver engine you are using, by displaying the Problem tab in the Solver Options dialog for that Solver engine. The problem size is also displayed in the Solver Model dialog.

## 9. Solver encountered an error value in a target or constraint cell.

This message appears when the Solver recalculates your worksheet using a new set of values for the decision variables (Changing Cells), and discovers an error value such as #VALUE!, #NUM!, #DIV/0! or #NAME? in the cell calculating the objective (Set Cell) or one of the constraints. Inspecting the worksheet for error values like these will usually indicate the source of the problem. If you’ve entered formulas for the right hand sides of certain constraints, the error might have occurred in one of these formulas rather than in a cell on the worksheet. For this and other reasons, it’s better to use only constants and cell references on the right hand sides of constraints.

If you see #VALUE!, #N/A or #NAME?, look for names or cell references to rows or columns that you have deleted. If you see #NUM! or #DIV/0!, look for unanticipated values of the decision variables which lead to arguments outside the domains of your functions – such as a negative value supplied to SQRT. You can often add constraints to avoid such domain errors; if you have trouble with a constraint such as \$A\$1 >= 0, try a constraint such as \$A\$1 >= 0.0001 instead.

In the Premium Solver Platform, when the Polymorphic Spreadsheet Interpreter is used (Solve With = PSI Interpreter), a more specific message usually appears instead of “Solver encountered an error value in a (nonspecific) target or constraint cell.” At a minimum, the message will say “Excel error value returned at cell *address*,” where *address* (e.g. Sheet1!\$A\$1) tells you exactly where the error was encountered. Other messages may tell you more about the error. The general form of the message is:

**Error condition at cell *address*. Edit your formulas, or use Excel Interpreter in the Solver Model dialog.** Error condition is one of the following:

Floating point overflow	Invalid token
Runtime stack overflow	Decision variable with formula
Runtime stack empty	Decision variable defined more than once
String overflow	Missing Diagnostic/Memory evaluation
Division by zero	Unknown function
Unfeasible argument	Unsupported Excel function
Type mismatch	Excel error value returned
Invalid operation	Non-smooth special function

See also result code 21, “Solver encountered an error computing derivatives,” and result code 12, with messages that can appear when the Interpreter first analyzes the formulas in your model (when you click the Check Model or Solve button).

“Floating point overflow” indicates that the computed value is too large to represent with computer arithmetic; “String overflow” indicates that a string is too long to be stored in a cell. “Division by zero” would yield #DIV/0! on the worksheet, and “Unfeasible argument” means that an argument is outside the domain of a function, such as =SQRT(A1) where A1 is negative.

“Unknown function” appears for functions whose names are not recognized by the Interpreter, such as user-written functions in VBA. “Unsupported Excel function” appears for the few functions that the Interpreter recognizes but does not support (see the list in the section “More on the Polymorphic Spreadsheet Interpreter” in the chapter “Analyzing and Solving Models”). “Non-smooth special function” appears if your model uses functions ABS, IF, MAX, MIN or SIGN, and the Require Smooth box is checked in the Solver Model dialog (see “Analyzing and Solving Models”).

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. Stop chosen when the maximum time limit was reached.

This message appears when (i) the Solver has run for the maximum time (number of seconds) allowed in the Max Time box in the Solver Options dialog *and* (ii) you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value in the Max Time box or click on the Continue button

instead of the Stop button in the Show Trial Solution dialog. But you should also consider whether re-scaling your model or adding constraints might reduce the total solution time required.

## 11. There is not enough memory available to solve the problem.

This message appears when the Solver could not 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, or even when “Analyzing Solver Model,” “Diagnosing Problem Function” or “Setting Up Problem” appears on the Excel status bar.

The Polymorphic Spreadsheet Interpreter in the Premium Solver Platform can use a considerable amount of memory, when you solve a problem by clicking the Solve button, and when you click the Check Model button in the Solver Model dialog. You can progressively reduce the memory used by the Interpreter by taking the following actions in order, using the Solver Model dialog:

1. Check the Sparse box in the Advanced options group.
2. Set the Check For option to Gradients.
3. Set the Solve With option to Excel Interpreter.

When Solve With = Excel Interpreter, the PSI Interpreter is not used and does not use any memory; any further problems are due to memory demands of the Solver engines, Microsoft Excel and Windows. You can save some memory by closing any Windows applications other than Excel, closing programs that run in the System Tray, and closing any Excel workbooks not needed to solve the problem.

## 12. Error condition at cell address. Edit your formulas, or use Excel Interpreter in the Solver Model dialog.

This message appears when the Polymorphic Spreadsheet Interpreter first analyzes the formulas in your model after you click the Solve button or the Check Model button in the Solver Model dialog. *Address* is the worksheet address of the cell (in Sheet1!\$A\$1 form) where the error was encountered, and *Error condition* is one of the following:

OLE error	Missing (
Invalid token	Missing )
Unexpected end of formula	Wrong number of parameters
Invalid array	Type mismatch
Invalid number	Code segment overflow
Invalid fraction	Expression too long
Invalid exponent	Symbol table full
Too many digits	Circular reference
Real constant out of range	External name
Integer constant out of range	Multi-area not supported
Invalid expression	Non-smooth function
Undefined identifier	Unknown function
Range failure	Loss of significance

Many of these messages will never appear as long as you entered your formulas in the normal way through Microsoft Excel, because Excel “validates” your formulas and displays its own error messages as soon as you complete formula entry. Some of the messages you may encounter are described in the following paragraphs.

**Undefined identifier** appears if you've used a name or identifier (instead of a cell reference such as A1) in a formula, and that name was not defined using the Insert Name Define... or Insert Name Create... menu commands in Excel. If you've used "labels in formulas" and checked the box "Accept labels in formulas" on the Calculation tab of the Tools Options... dialog in Excel, this message will appear. The Interpreter does not support this use of labels in formulas – you'll have to define these labels with the Insert Name Define... or Insert Name Create... commands, or else set Solve With = Excel Interpreter to avoid using the PSI Interpreter.

**Circular reference** appears if Excel has already warned you about a circular reference in your formulas, and it can also appear if you've used array formulas in a "potentially circular" way. (For example, if cells A1:A2 contain {=1+B1:B4} and cells B3:B4 contain {=1+A1:A4}, Excel doesn't consider this a circular reference, but the PSI Interpreter does.) If you must use circular references in your model, you'll have to set Solve With = Excel Interpreter to avoid using the PSI Interpreter.

**External name** appears if your formulas use references to cells in other *workbooks* (not just other worksheets), and the Interpreter is unable to open those workbooks. You should ensure that the external workbooks are in the same folder as the Solver workbook, or for better performance, move or copy the worksheets you need into the workbook containing the Solver model.

**Multi-area not supported or Missing )** appears if your formulas or defined names use multiple selections such as (A1:A5,C1:H1). While the Interpreter does accept *argument lists* consisting of single selections, such as =SUM(A1:A5,C1:H1), it does not accept multiple selections for defined names, or for single arguments such as =SUMSQ((A1:A5,C1:H1), (B1:B5,C2:H2)). If you must use such multiple selections, you'll have to set Solve With = Excel Interpreter.

*Note:* Result code 12 was formerly associated with the message "Another Excel instance is using SOLVER32.DLL. Try again later," which does not occur in the modern versions of Excel and Windows supported by the Premium Solver and Premium Solver Platform.

### 13. Error in model. Please verify that all cells and constraints are valid.

This message means that the internal "model" (information about the variable cells, Set Cell, constraints, Solver options, etc.) is not in a valid form. An "empty" or incomplete Solver model, perhaps one with no objective in the Set Cell edit box and no constraints other than bounds on the variables in the Constraints list box, can cause this message to appear. You might also receive this message if you are using the *wrong version* of either Solver.xla or Solver32.dll, or if you've modified the values of certain hidden defined names used by the Solver, either interactively or in a VBA program. ***To guard against this possibility, you should avoid using any defined names beginning with "solver" in your own application.***

### 14. Solver found an integer solution within tolerance. All constraints are satisfied.

If you are solving a mixed-integer programming problem (any problem with integer constraints) using one of the Premium Solver products, with a *non-zero value* for the integer Tolerance setting on the Integer tab of the Solver Options dialog, 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. (For more information, see "Options for Mixed-Integer Problems" in the chapter "Solver Options.") 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 Premium Solver products will produce the message “Solver found a solution. All constraints are satisfied” (result code 0, shown earlier in this section).

#### **15. Stop chosen when the maximum number of feasible [integer] solutions was reached.**

If you are using the Evolutionary Solver, this message appears when (i) the Solver has found the maximum number of feasible solutions (values for the variables that satisfy all constraints) allowed by the Max Feasible Sols box on the Limits tab of the Solver Options dialog *and* (ii) you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value in the Max Feasible Sols box, or click on the Continue button instead of the Stop button in the Show Trial Solution dialog to continue the solution process.

If you are using one of the other Solver engines on a problem with integer constraints, this message appears when (i) 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 box on the Integer tab of the Solver Options dialog *and* (ii) you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value in the Max Integer Sols box, or click on the Continue button instead of the Stop button in the Show Trial Solution dialog. 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 in the Premium Solver Platform, try activating more Cuts and Heuristics on the Integer tab of the Solver Options dialog.

#### **16. Stop chosen when the max number of feasible [integer] subproblems was reached.**

If you are using the Evolutionary Solver, this message appears when (i) the Solver has explored the maximum number of subproblems allowed in the Max Subproblems box on the Limits tab of the Solver Options dialog *and* (ii) you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value in the Max Subproblems box, or click on the Continue button instead of the Stop button in the Show Trial Solution dialog to continue the solution process.

If you are using one of the other Solver engines on a problem with integer constraints, this message appears when (i) the Solver has explored the maximum number of integer subproblems (each one is a “regular” Solver problem with additional bounds on the variables) allowed in the Max Subproblems box on the Integer tab of the Solver Options dialog *and* (ii) you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value in the Max Subproblems box, or click on the Continue button instead of the Stop button in the Show Trial Solution dialog. 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 in the Premium Solver Platform, try activating more Cuts and Heuristics on the Integer tab of the Solver Options dialog.

#### **17. Solver converged in probability to a global solution.**

If you are using the multistart methods for global optimization, with either the nonlinear GRG Solver or a field-installable nonlinear Solver engine (by checking the Global Optimization options in the appropriate Solver Options dialog), this message appears when the multistart method's Bayesian test has determined that all of the locally optimal solutions have *probably* been found; the solution displayed on the worksheet 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. All variables must have both upper and lower bounds.**

If you are using the Interval Global Solver or the OptQuest Solver, this message appears if you have not defined lower and upper bounds on all of the decision variables in the problem. If you are using the Evolutionary Solver or the multistart methods for global optimization, and you have checked the box “Require Bounds on Variables” in the Solver Options dialog (it is checked by default), this message will also appear. You should add the missing bounds and try again. Upper bounds must be entered in the Constraints list box. Lower bounds of zero can be applied to all variables by checking the “Assume Non-Negative” box in the Solver Options dialog; non-zero lower bounds must be entered in the Constraints list box. You *must* define bounds on all variables in order to use the Interval Global Solver or the OptQuest Solver. For the Evolutionary Solver or the multistart methods, such bounds are not absolutely required (you can uncheck the box “Require Bounds on Variables”), but they are a practical necessity if you want the Solver to find good solutions in a reasonable amount of time.

#### **19. Variable bounds conflict in binary or alldifferent constraint.**

This message appears 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 and try to solve again.

#### **20. Lower and upper bounds on variables allow no feasible solution.**

This message appears if you have defined lower and upper bounds on a decision variable, where the lower bound is greater than the upper bound. This (obviously) means there can be no feasible solution, but most Solver engines will detect this condition before even starting the solution process, and display this message instead of “Solver could not find a feasible solution” to help you more quickly identify the source of the problem. If you have defined your bounds and other constraints in uniform blocks, the lower and upper bounds on a given range of cells will appear consecutively in the Constraints list box (where they are sorted), making it easy to spot the inconsistent bounds.

#### **21. Solver encountered an error computing derivatives. Consult Help on Derivatives, or use Excel Interpreter in the Solver Model dialog.**

This message appears when the Polymorphic Spreadsheet Interpreter in the Premium Solver Platform is being used (Solve With = PSI Interpreter), and the Interpreter encounters an error when computing derivatives via automatic differentiation. (For more information, see “More on the Polymorphic Spreadsheet Interpreter” in the chapter “Analyzing and Solving Models.”) The most common cause of this message is a non-smooth function in your objective or constraints, for which the derivative is

undefined. But in general, automatic differentiation is somewhat more strict than finite differencing: As a simple example, `=SQRT(A1)` evaluated at `A1=0` will yield this error message when the Solver is using automatic differentiation (since the derivative of the `SQRT` function is algebraically undefined at zero), but it won't yield an error when Solve With = Excel Interpreter and the Solver is using finite differencing.

If you receive this message, follow the steps in “Analyzing Model Exceptions” in the chapter “Analyzing and Solving Models” to pinpoint the exact cell formulas that are non-smooth. If you cannot modify your formulas to eliminate the non-smooth functions, your options are to (i) use a Solver engine, such as the Evolutionary Solver or the OptQuest Solver, that doesn't require derivatives, or (ii) set Solve With = Excel Interpreter and solve the problem using finite differencing.

## Interval Global Solver Result Messages

The Interval Global Solver can return many of the standard result codes and Solver Result Messages described above, but it can also return one of three custom result codes and messages, as described below.

### 1000. Interval Solver requires PSI Interpreter and strictly smooth functions.

This message appears if you select the Interval Global Solver engine and click Solve, and the Solve With option is set to Excel Interpreter *or* the Check For option is set to Gradients, *or* if your model contains *any* non-smooth functions. The Interval Global Solver considers the „special functions ABS, IF, MAX, MIN or SIGN non-smooth, whether or not the Require Smooth box is checked in the Solver Model dialog. If you receive this message, follow the steps in “Analyzing Model Exceptions” in the chapter “Analyzing and Solving Models” to pinpoint the exact cell formulas that are non-smooth; you must modify these formulas to use the Interval Global Solver.

### 1001. Function cannot be evaluated for given real or interval arguments.

This message may appear (instead of “Solver encountered an error value...”) if the Interval Global Solver encounters an arithmetic operation or function that it cannot evaluate for the current values of the decision variables. Recall that the Interval Global Solver evaluates Excel formulas over intervals such as `[1, 2]` as well as real numbers. In the course of seeking a solution, the Solver may have to evaluate a formula that (for example) involves division by an *interval containing zero*, or the square root of an *interval containing negative values*, which yield errors. If you receive this message, try adding constraints, or adjusting the right hand sides of existing constraints to eliminate the problem. For example, if you have trouble with a constraint such as `$A$1 >= 0`, try a constraint such as `$A$1 >= 0.0001` instead.

### 1002. Solution found, but not proven globally optimal.

This message indicates that the Interval Global Solver has systematically explored the solution space and has found a solution that is very probably the global optimum, but it has not been able to “prove global optimality.” Most often, this means that there is more than a tiny difference between this solution's objective value and the *best bound* on the global optimum's objective value that the Solver has been able to find. For more information, see the discussion of “Interval Global Solver Stopping Conditions” in the section “Limitations on Global Optimization.”

---

# 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 messages such as “Solver could not find a feasible solution,” “Solver could not improve the current solution,” or even “The linearity conditions required by this Solver engine are not satisfied,” 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 box in their Solver Options dialogs. When this box is checked, 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 your Excel formulas*.

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 worksheet may range from perhaps 1 to 1,000.

If you have the Premium Solver Platform, and you're experiencing results that may be due to poor scaling, you can check your model for scaling problems that arise *in the middle of your Excel formulas* by selecting the Scaling Report when it appears in the Solver Results dialog, and examining the results of this report, as described in the chapter “Solver Reports.” If you have the Premium Solver, you'll have to go through each of your formulas and play “what-if” manually to identify such problems.

## Historical Note on Scaling and Linearity Tests

Poor scaling is an ever-present issue for the Solver, and for almost any kind of mathematical software. Successive versions of the Solver have used increasingly sophisticated methods to deal with poor scaling, culminating in the Premium Solver Platform's tools for analyzing your model for scaling problems.

The Use Automatic Scaling option has been available in the standard Microsoft Excel Solver since Excel 5.0, but in Excel 5.0 and 7.0, this option was effective only for nonlinear problems solved with the GRG Solver. Because of this, the Solver's linearity test (used when the “Assume Linear Model” box was checked) could be “fooled” by an all-linear, but poorly scaled model – yielding the error message “The conditions for Assume Linear Model are not satisfied.”

In Excel 97, 2000, XP, 2003, and Excel 2007 and the Premium Solver products, the Use Automatic Scaling option is effective for all types of models, and the Solver also

uses a more robust test for linearity. Since no automatic scaling method will work in *all* situations, it is still **good practice to ensure that the model on your worksheet is well scaled** – even if you do take advantage of the Use Automatic Scaling option.

---

## The Tolerance Option and Integer Constraints

Users who solve problems with integer constraints using the standard Excel Solver occasionally report that “Solver claims it found an optimal solution, but I manually found an even better solution.” What happens in such cases is that the Solver stops with the message “Solver found a solution” because it found a solution *within the range* of the true integer optimal solution *allowed by the Tolerance* option in the Solver Options dialog. In similar cases, the Premium Solver products display a message “Solver found an integer solution within tolerance,” to avoid confusion.

When you solve a mixed-integer programming problem (any problem with integer constraints) using the Simplex, LP/Quadratic, SOCP Barrier, GRG or Interval Global Solver, all of which employ the Branch & Bound method, the solution process is governed by the integer Tolerance option on the Solver Options or Integer Options dialog tab. Since the *default setting of the Tolerance option* is 0.05, the Solver stops when it has found a solution satisfying the integer constraints whose objective is within 5% of the true integer optimal solution. Therefore, you may know of or be able to discover an integer solution that is better than the one found by the Solver.

The reason that the default setting of the integer Tolerance option is 0.05 is that the solution process for integer problems – which can take a great deal of time in any case – often finds a near-optimal solution (sometimes *the* optimal solution) relatively quickly, and then spends far more time exhaustively checking other possibilities to find (or verify that it has found) the very best integer solution. The integer Tolerance default setting is a compromise value that often saves a great deal of time, and still ensures that a solution found by the Solver is within 5% of the true optimal solution.

To ensure that the Solver finds the true integer optimal solution – possibly at the expense of far more solution time – set the integer Tolerance option to zero. In the Premium Solver products, look for the Tolerance edit box on the Integer tab of the Solver Options dialog.

---

## Limitations on Smooth Nonlinear Optimization

As discussed in the chapter “Solver Models and 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.

As discussed in the chapter “Analyzing and Solving Models,” in the Premium Solver Platform you can easily check whether your model is **convex** or **non-convex**, by clicking the Model button, selecting Check For Convexity, and clicking the Check Model button. The result of the convexity test may be *conclusive* (the Solver has proven that the model is convex or non-convex) or *inconclusive* (the Solver was unable to prove either condition). If the test is inconclusive, you are best advised to assume that your model is non-convex, unless you can prove through your own mathematical analysis that it is convex.

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.” In the Premium Solver Platform, you can check the Global Optimization options in the Solver Options dialog for the nonlinear GRG Solver, and use the **multistart** method to *automatically* run the Solver from multiple starting points.

Note that, when the GRG Nonlinear Solver is selected in the dropdown list in the Solver Parameters dialog, the Generalized Reduced Gradient algorithm is used to solve the problem – *even if it is actually a linear model* that could be solved by the (faster and more reliable) Simplex or Barrier method. The GRG method will *usually* find the optimal solution to a linear problem, but occasionally you will receive a Solver Result Message indicating some uncertainty about the status of the solution – especially if the model is poorly scaled, as discussed above. So you should always ensure that you have selected the right Solver engine for your problem.

## 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 (“Solver found a solution”) 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 setting in the Solver Options dialog).

### ***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 message (“Solver has converged to the current solution”) appears. 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 in the Convergence box in the Solver Options dialog 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 box in the Solver Options dialog is checked. 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 message “Solver cannot improve the current solution,” 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 User Guide, as well as most of the books recommended in the Introduction. 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 Interval Global Solver or 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 the Premium Solver Platform 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 method 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.

## **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 for the Premium Solver Platform.

---

## Limitations on Global Optimization

With the Premium Solver Platform, you have several choices available for solving global optimization problems: You can use the nonlinear GRG Solver (or a field-installable nonlinear Solver engine) with multistart methods; you can use the Interval Global Solver; or you can use the Evolutionary Solver or OptQuest Solver to seek global solutions to smooth nonlinear problems, though they are designed primarily for non-smooth problems. Overall, the Premium Solver Platform is very likely the world's best platform for global optimization.

Which choice should you use? Perhaps the best answer is “try them all, and use the one that performs best on your model.” But you may favor the new Interval Global Solver if it is important to you to find the *true global optimum* – not just a “better” local optimum, or a “good” solution that is better than what you are using now. For example, if you are seeking the minimum energy configuration of atoms in a molecule, you will want the true global optimum because in nature, the molecule will be in that configuration most of the time. The Interval Global Solver may take longer to run, but it has a better chance than other methods of finding the *true global optimum*, and it is the only Solver engine that can “prove” that it has found the global optimum.

This section describes the characteristics and limitations of global optimization with the Interval Global Solver. For more information on the multistart methods, see “GRG Solver with Multistart Methods” in the previous section, “Limitations on Smooth Nonlinear Optimization.” For more information on the Evolutionary Solver, see the following section, “Limitations on Non-Smooth Optimization.”

## Rounding and Possible Loss of Solutions

The Interval Global Solver uses *deterministic* methods to search for the global optimum, whereas the Evolutionary Solver and the multistart methods for global optimization use *nondeterministic* methods, which involve an element of random chance. Given time, the Interval Global Solver will find a “proven” global optimum, and it will find *all* real solutions to a system of nonlinear equations (subject to the limitations described here). The Evolutionary Solver and multistart methods have no way to “prove” that the global optimum has been found.

But in rare cases, *when its advanced options are used*, the Interval Global Solver can “lose track of” potential solutions that might prove to be the global optimum, fail to find a feasible solution, or “lose” real solutions of a system of nonlinear equations, because of roundoff errors that can arise from the use of finite precision computer arithmetic. This is more likely to occur for solution(s) found at or very close to the boundaries of constraints.

The Interval Global Solver uses the Polymorphic Spreadsheet Interpreter's *interval arithmetic* methods, which use “directed rounding” of floating point arithmetic operations at the machine level, to *eliminate* the possibility of roundoff error. If you use only the “Classic Interval” option in the Methods option group of the Solver Options dialog, you can be confident that the Interval Global Solver will (eventually, given enough time) find a “proven” global optimum. But in its advanced methods,

the Interval Global Solver uses both interval arithmetic and ordinary real arithmetic, for the sake of performance; it avoids using directed rounding for *all* arithmetic operations, even those involving ordinary real numbers (which would have to be enclosed in narrow intervals), and it seeks to avoid spending a great deal of time processing large numbers of very small “boxes” to rigorously verify that they don’t contain possible solutions.

The ideal of finding globally optimal solutions with rigorous guarantees is no doubt achievable for problems of low dimension (with a small number of variables). But it has been shown that the simplest possible global optimization problem – a quadratic programming problem in the general case (where the objective is **non-convex**, and there may be many locally optimal points at constraint boundaries) – is *NP-hard*, meaning that the solution time is very likely to grow *exponentially* with the number of decision variables.

The Interval Global Solver trades off rigorous guarantees of finding the globally optimal solution in favor of fast solution times on realistic size problems. And its methods, while not rigorous, are *very effective* at finding the true global optimum. In fact, Frontline Systems has not yet seen or constructed an example problem where the Interval Global Solver actually “loses” a solution that turns out to be the global optimum. As a practical matter, you are likely to receive a Solver Result Message such as “Solution found, but not proven globally optimal” or “Solver cannot improve the current solution” in situations where, for various reasons, the Solver has not been able to verify that it has found the true global optimum.

## Interval Global Solver Stopping Conditions

It is helpful to understand what the Interval Global Solver can and cannot do, and what each of the possible Solver Result Messages means for this Solver engine. At best, the Interval Global Solver will find a “proven” globally optimal solution to a reasonably *well-scaled* smooth nonlinear optimization problem – in a reasonable amount of time. But at times, the Solver will be unable to “prove” that the solution is globally optimal, unable to improve the current solution in a reasonable amount of time, or unable to find a feasible solution. And the words “proven” and “prove” are in quotes because they are subject to limitations due to roundoff error, as discussed above under “Rounding and Possible Loss of Solutions.”

### ***When Solver Cannot Find a Feasible Solution***

When the Interval Global Solver reports that “Solver could not find a feasible solution,” and you have allowed the Solver to run without interruption until this message appears, it is very likely – though not 100% certain – that no feasible solution exists. The Interval Global Solver is designed to “prove feasibility” as well as global optimality, but this is subject to limitations due to roundoff error.

### ***When Solver Cannot Improve the Current Solution***

When the Interval Global Solver reports that “Solver cannot improve the current solution,” it means that the Solver has not found an “improved global solution” (a feasible solution with an objective value better than the currently best known solution), in the amount of time specified by the Max Time without Improvement option in the Solver Options dialog. The reported solution is the best one found so far, but the search space has not been fully explored. If you receive this message, and you are willing to spend more solution time to have a better chance of “proving” global optimality, increase the value of the Max Time without Improvement option.



### ***When Solver Cannot Prove Global Optimality***

As described in “Global Optimization” in the chapter “Solver Models and Optimization,” the Interval Global Solver processes a list of “boxes” that consist of bounded intervals for each decision variable, progressively subdividing and “shrinking” them, and improving a known bound on the globally optimal objective function value. Eventually, the boxes that remain each enclose a locally optimal solution, and the best of these is chosen as the globally optimal solution. The Interval Global Solver returns “Solver found a solution” (result code 0) when it determines that, in the box enclosing the best solution, (i) the bounded intervals for each decision variable are smaller than the Accuracy value in the Solver Options dialog, and (ii) the objective value in this box differs from the best known bound on the globally optimal objective by no more than the Accuracy value. When the Interval Solver finishes processing the list of boxes, but the above two conditions are not met, it returns the message “Solution found, but not proven globally optimal.”

### **Interval Global Solver and Integer Constraints**

As with the nonlinear GRG Solver, the performance of the Branch & Bound method on nonlinear global optimization problems with integer constraints is limited by the performance of the Interval Global Solver on the subproblems. If the Interval Global Solver should fail to find the globally optimal solution, or fail to find a feasible point when one exists on a given run, this may prevent the Branch & Bound method from finding the true integer optimal solution. Since a single global optimization run can take a great deal of time, and the Branch & Bound process may require thousands of such runs, the Interval Global Solver makes further tradeoffs in favor of fast solutions rather than guarantees of finding the global optimum on each run, when it is solving a problem with integer constraints. In most cases, however, the combination of the Branch & Bound method and the Interval Global Solver will at least yield a relatively good integer solution.

---

## **Limitations on Non-Smooth Optimization**

As discussed in the chapter “Solver Models and 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 in Excel is the IF function where the conditional test is dependent on the decision variables. Other common discontinuous functions are CHOOSE, the LOOKUP functions, and COUNT. Common non-smooth functions in Excel are ABS, MIN and MAX, INT and ROUND, and CEILING and FLOOR. Functions such as SUMIF and the database functions are discontinuous if the criterion or conditional argument depends on the decision variables.

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.

You *can* use discontinuous functions such as IF and CHOOSE in calculations on the worksheet that are *not dependent on the decision variables*, and are therefore constant in the optimization problem. But any discontinuous functions that do depend on the variables make the overall Solver model non-smooth. Users sometimes fail to realize that certain functions, such as ABS and ROUND, are non-smooth. For more information on this subject, read the section “Non-Smooth Functions” in the chapter “Solver Models and Optimization.”

## Effect on the 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 are using the Premium Solver Platform with default settings, the Interpreter will compute derivatives of the problem functions using *automatic differentiation*. (For further information, see “More on the Polymorphic Spreadsheet Interpreter” in the chapter “Analyzing and Solving Models.”) If you try to solve a problem with non-smooth or discontinuous functions (other than the „special functions” ABS, IF, MAX, MIN or SIGN) using the GRG Solver, you will likely receive the message “Solver encountered an error computing derivatives.” If you check the Require Smooth box in the Solver Model dialog, you will also receive this message for models that use the „special functions.” You can set the Solve With option to Excel Interpreter and solve your model – but only with the caveats noted above. The Premium Solver always uses the Excel Interpreter, so these caveats apply whenever you try to solve a non-smooth problem.

If you try to solve a problem with non-smooth or discontinuous functions with the linear Simplex Solver (using Solve With = Excel Interpreter in the Premium Solver Platform), 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 has Converged to the Current Solution***

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 in the Solver Options dialog. 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. In this latter case, it may help to interrupt the Solver with the ESC key and click the Restart button (which replaces the worst half of the population with newly sampled points), or to run the Evolutionary Solver again with a larger Population Size and/or an increased Mutation Rate, which increases the chances of a diverse population.

## ***When Solver Cannot Improve the Current Solution***

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 value on the Limits tab of the Solver Options dialog, in the amount of time specified by the Max Time without Improvement option in the same dialog. 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 values, increase the Max Subproblems and Max Feasible Sols values, 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 displays the message “Solver found a solution,” you may have found at least a *locally optimal* point (but remember that this test depends on smoothness of the problem functions).
5. Select and examine the Population Report. If the Best Values are similar from run to run of the Evolutionary Solver, and if the Standard Deviations are small, this may be reason for confidence that your solution is close to the global optimum. Since optimization tends to drive the variable values to extremes, if the solution is feasible and the Best Values are close to the Maximum or Minimum Values listed in the Population Report, this may indicate that you have found an optimal solution.

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 we suggested in the Introduction, 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. The chapter “Building Large-Scale Models” describes 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 Options

This chapter describes the options available in the Solver Options dialog for the standard Microsoft Excel Solver, and in the Solver Options dialogs for each of the bundled Solver engines in the Premium Solver and Premium Solver Platform. It also briefly describes how these options may be examined or set programmatically.

In the Premium Solver Platform, options may be examined or set interactively via the Solver Options dialogs shown in this chapter, or programmatically using either the **new object-oriented API** described below and in the chapter “Using the Object-Oriented API,” or the **traditional VBA functions** described in the later chapter “Using Traditional VBA Functions.”

In the Solver Platform SDK, options may be examined or set via its **object-oriented API**, just like the Premium Solver Platform’s API, or via the **SDK procedural API**. The object-oriented API is described below; the SDK procedural API is described in the SDK API Reference Guide. The string names of most options are the same for both platforms, and are shown for each option below.

Bear in mind that the options that control numerical tolerances and solution strategies are pre-set to the choices that are most appropriate for the majority of problems; you should change these settings only when necessary, after carefully reading this chapter. The options you will use most often are common to all the Solver products, and control features like the display of iteration results, or the upper limits on solution time or Solver iterations.

---

## Setting Options Programmatically

In the Premium Solver Platform, you can examine or set Solver Engine options in VBA using the object-oriented API described in this section. In the Solver Platform SDK, you can examine or set Solver Engine options, in a variety of programming languages, using the same object-oriented API. In both cases, all option values are of type double, though for some options only integer values, or values 0 and 1 are used.

### Object-Oriented API

In the object-oriented API, each Solver engine option or parameter is represented by an **EngineParam** object instance. This object has properties Name, Value, Default (the initial or default value), MinValue, and MaxValue (the minimum and maximum allowed values). All the options or parameters for a Solver engine belong to a collection, which is an **EngineParamCollection** object.

To access an option or parameter, you start with a reference to the Solver engine object, say `myEngine` or `myProb.Engine`. The engine object's `Params` property refers to the `EngineParamCollection` object. As with all collections, you can access an individual `EngineParam` in the collection by name or by index. For example, to refer to the `MaxTime` limit for the problem's currently selected Solver engine, you'd write `myProb.Engine.Params("MaxTime")`.

Once you have a reference to the `EngineParam` object (as above), you can get or set its properties using simple assignment statements. For example, you can set the `MaxTime` limit for the currently selected Solver engine to 1000 seconds by writing:

```
VBA: myProb.Engine.Params("MaxTime").Value = 1000
```

To get the current `MaxTime` parameter value, put the property reference on the right hand side of an assignment statement (declaring **Dim maxTime As Double**):

```
VBA: maxTime = myProb.Engine.Params("MaxTime").Value
```

In the Solver Platform SDK, the same kinds of assignment statements can be used, with just slight differences due to the syntax of various programming languages:

```
VB6: myProb.Engine.Params("MaxTime").Value = 1000
```

```
VB.NET: myProb.Engine.Params("MaxTime").Value = 1000
```

```
C++: myProb.Engine.Params(L"MaxTime").Value = 1000;
```

```
C#: myProb.Engine.Params("MaxTime").Value = 1000;
```

```
Matlab: myProb.Engine.Params('MaxTime').Value = 1000;
```

```
Java: myProb.Engine().Params().Item("MaxTime").Value(1000);
```

(Since Java currently lacks properties, the syntax used by the Solver Platform SDK is somewhat different.) To get the current `MaxTime` parameter value, put the property reference on the right hand side of an assignment statement, for example in C#:

```
double maxTime = myProb.Engine.Params("MaxTime").Value;
```

You can access all of the options and parameters supported by a Solver engine by indexing its `EngineParamCollection`. For example, `myProb.Engine.Params(0)` refers to the first parameter in the collection. In all the object-oriented languages, you can write a for-loop to index through all of the parameters like the following example in VBA, VB6 or VB.NET:

```
For i = 0 to myProb.Engine.Params.Count - 1
    MsgBox myProb.Engine.Params(i).Name & " = " &
        myProb.Engine.Params(i).Value
Next i
```

In VBA, VB6, VB.NET, and C#, you can also iterate through a collection using a “for each” loop:

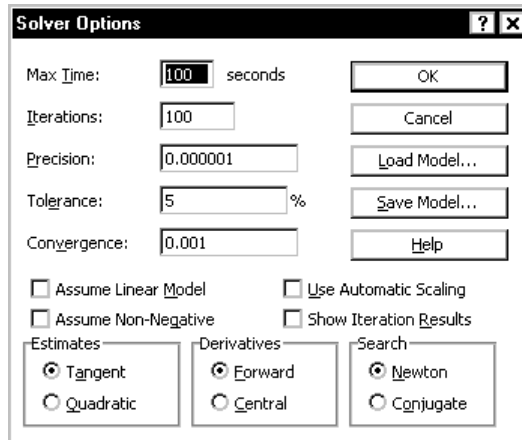
```
Dim myParam as EngineParam
For Each myParam in myProb.Engine.Params
    MsgBox myParam.Name & " = " & myParam.Value
Next
```

---

## The Standard Microsoft Excel Solver

There is just one Solver Options dialog displayed by the standard Microsoft Excel Solver, containing options for both the linear Simplex Solver and the nonlinear GRG Solver engines. This dialog box is depicted on the next page, as it appears in Excel

2000, XP, 2003 and 2007. All of the options in this dialog – with the exception of the Assume Linear Model check box, discussed later in this chapter – are also present in the options dialogs for the Premium Solver and Premium Solver Platform.



We will first discuss the options common to all Solver engines. We will include the Assume Linear Model option, used only in the standard Excel Solver. Next, we will describe the options specific to the linear Simplex Solver, the LP/Quadratic Solver, the SOCP Barrier Solver, the nonlinear GRG Solver (including the multistart methods), the Interval Global Solver, and the Evolutionary Solver. Then we will discuss the options on the Limits and Integer tabs in the Solver Options dialogs, used to control the Evolutionary Solver and advanced methods for integer programming problems in each of the Solver engines. A special section will discuss the extensive options for integer programming in the LP/Quadratic Solver. Finally, we will discuss loading, saving, and merging Solver models, including the new PSI function format.

## Common Solver Options

### Max Time and Iterations

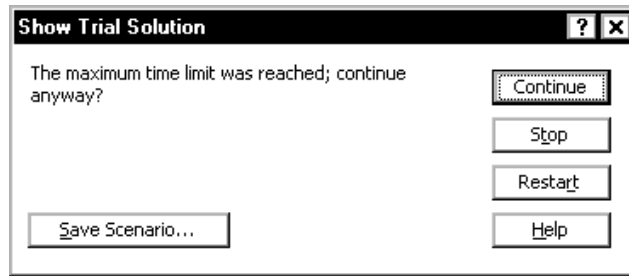
*VBA / SDK:* Parameter Names "MaxTime", "Iterations", integer value > 0

The value in the Max Time edit box determines the maximum time in seconds that the Solver will run before it stops, including problem setup time and time to find the optimal solution. For problems with integer constraints, this is the total time taken to solve all subproblems explored by the Branch & Bound method. The default value is 100 seconds.

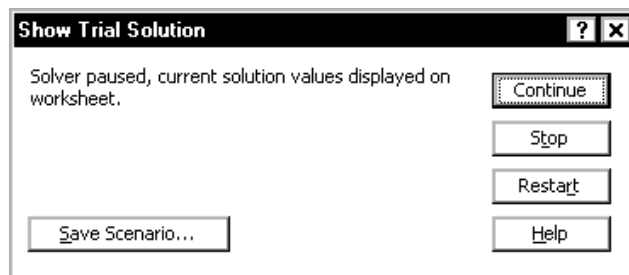
The value in the Iterations edit box determines the maximum number of iterations ("pivots" for the Simplex Solver, or major iterations for the GRG Solver) that the Solver may perform on one problem. A new "Trial Solution" is generated on each iteration; the most recent Trial Solution is reported on the Excel status bar. For problems with integer constraints, the Iterations setting determines the maximum number of iterations for any *one* subproblem. The default value is 100 iterations.

Users of the Premium Solver and Premium Solver Platform will most likely want to increase the Max Time and Iterations settings from their default values, in order to solve larger problems. Bear in mind that if the maximum time or maximum number of iterations is exceeded, the Solver will stop and display a dialog like the one shown below: You will have the option to stop at that point or to continue the solution

process. If you click on the Continue button, the time or iteration limit is removed, and you will not be prompted again.



There is really no downside to specifying quite a large value for the Max Time or Iterations setting: If you ever get impatient, you can simply press the ESC key while the Solver is running. If your model is large enough to take some time to recalculate even once, you should hold down the ESC key for a second or two. After a momentary delay, the dialog box shown below will appear, and you will have the option to stop at that point or continue or restart the solution process.



## Precision

*VBA / SDK:* Parameter Name "Precision",  $0 < \text{value} < 1$

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. Recall from "Elements of Solver Models" in the chapter "Solver Models and Optimization" that a constraint is satisfied if the relation it represents is true *within a small tolerance*; the Precision value is that tolerance. With the default setting of 1.0E-6 (0.000001), a calculated left hand side of -1.0E-7 would satisfy a constraint such as  $A1 \geq 0$ .

### ***Precision and Regular Constraints***

Use caution in making this number much smaller, since the finite precision of computer arithmetic virtually ensures that the values calculated by Microsoft Excel and the Solver 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 input data to work in *units of millions*, or checking the Use Automatic Scaling box 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 = integer, A1:A5 = binary or A1:A5 = 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.

## Tolerance and Convergence

VBA / SDK: Parameter Names "IntTolerance", "Convergence",  $0 \leq \text{value} \leq 1$

The Tolerance option determines how close a “candidate” integer solution must be to the true integer optimal solution before the Solver stops. In the Premium Solver and Premium Solver Platform, this option appears on the Integer tab of the Solver Options dialog; it is described more fully in a later section focusing on options for integer programming problems.

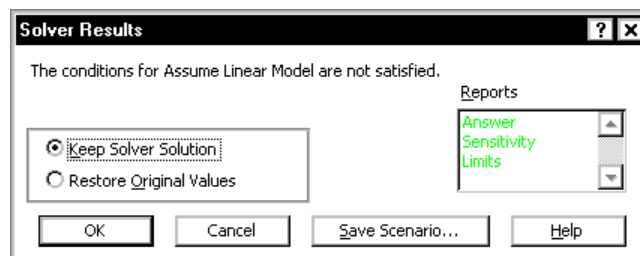
The Convergence option controls the stopping conditions used by the GRG Solver and the Evolutionary Solver that lead to the message “Solver has converged to the current solution.” In the Premium Solver and Premium Solver Platform, it appears in the Options dialogs for the Evolutionary Solver and the GRG Solver. It is described more fully in later sections focusing on options for these Solver engines.

## Assume Linear Model

VBA: Parameter Name "AssumeLinear"

SDK: Not applicable

In the standard Microsoft Excel Solver, the Assume Linear Model check box controls the choice of the linear Simplex Solver or the nonlinear GRG Solver: When it is checked, the Simplex Solver is used; otherwise the GRG Solver is used. The box is labeled “Assume Linear Model” because the Solver initially *assumes* that your model is made up entirely of *linear* functions for the objective and constraints. It solves the problem using the Simplex Solver, but as it recalculates the worksheet, it performs tests to verify that the objective and constraints are, in fact, behaving as linear functions of the variables. If these tests are not satisfied to within a close tolerance, the Solver stops (usually before starting any Trial Solutions, in modern versions of Microsoft Excel and in the Premium Solver products) and displays the Solver Results dialog with the message “The conditions for Assume Linear Model are not satisfied.”



In the Premium Solver and Premium Solver Platform, the Solver engine to be used is selected from a dropdown list in the main Solver Parameters dialog, and the Assume Linear Model check box is not used. If you choose the Premium Solver's LP Simplex Solver, the Solver will test your objective and constraint functions for linearity as described above, and will display the Solver Results dialog with the message “**The linearity conditions required by this Solver engine are not satisfied.**”

In the Premium Solver Platform, if you choose the LP/Quadratic Solver or the SOCP Barrier Solver, the Solver will check whether your objective and constraint functions are linear or *convex quadratic*, as supported by the Solver engine. If any functions are general nonlinear, or if any constraint is quadratic for the LP/Quadratic Solver, it will display the Solver Results dialog with the message “**The linearity conditions**

**required by this Solver engine are not satisfied.”** (In this context, “linearity conditions” is meant to include the allowed convex quadratic functions).

## Assume Non-Negative

**VBA:** Parameter Name "AssumeNonneg", value 1/True or 0/False

**SDK:** Use Variable object NonNegative method or SolverVarNonNegative function

When this box is checked, any decision variables that are not given explicit lower bounds via  $\geq$ , binary, or alldifferent constraints in the Constraints list box of the Solver Parameters dialog will be given a lower bound of zero when the problem is solved. This option has no effect for decision variables that *do* have explicit  $\geq$  constraints, even if those constraints allow the variables to assume negative values.

## Use Automatic Scaling

**VBA / SDK:** Parameter Name "Scaling", value 1/True or 0/False

When this box is checked, 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. For more information, see “Problems with Poorly Scaled Models” in the chapter “Diagnosing Solver Results,” and “The Scaling Report” in the chapter “Solver Reports.”

Note: In older versions of Microsoft Excel (prior to Excel 97), *this option is effective only for nonlinear optimization problems* solved with the GRG Solver.

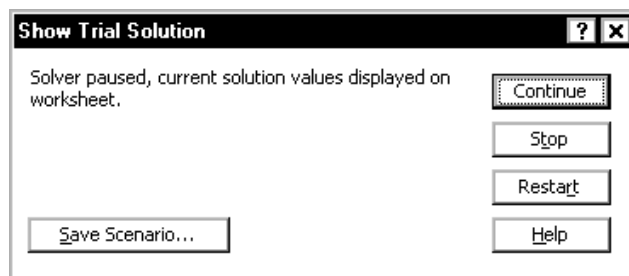
If your model is nonlinear and you check the Use Automatic Scaling box, *make sure that the initial values for the decision variables are “reasonable,”* i.e. of roughly the same magnitudes that you expect for those variables at the optimal solution. The effectiveness of the Automatic Scaling option depends on how well these starting values reflect the values encountered during the solution process.

## Show Iteration Results

**VBA:** Parameter Name "StepThru", value 1/True or 0/False

**SDK:** Define an Evaluator for Eval\_Type\_Iteration

When this box is checked, a dialog like the one below will appear on every iteration during the solution process:



This is the same dialog that appears when you press ESC at any time during the solution process, but when the Show Iteration Results box is checked it appears automatically on every iteration. When this dialog appears, the best values so far for the decision variables appear on the worksheet, which is recalculated to show the

values of the objective function and the constraints. You may click the Continue button to go on with the solution process, the Stop button to stop immediately, or the Restart button to restart (and then continue) the solution process. You may also click on the Save Scenario... button to save the current decision variable values in a named scenario, which may be displayed later with the Microsoft Excel Scenario Manager. For more information on this dialog and the effect of the Restart button, see the section “During the Solution Process” in the chapter “Diagnosing Solver Results.”

## Bypass Solver Reports

**VBA:** Parameter Name "BypassReports", value 1/True or 0/False

**SDK:** Not Applicable

This check box is a “Common Solver Option” in the Premium Solver products; it appears in the Solver Options dialogs shown below for the Simplex or LP/Quadratic Solver, SOCP Barrier Solver, GRG Nonlinear Solver, Interval Global Solver, and Evolutionary Solver. You can use it to save time during the solution process if you do not need the reports for the current solution run. The reports are selected from the Solver Results dialog at the end of the solution process; unless this box is checked, the Solver always performs extra computations to prepare for the possibility that you will select one or more reports from the Solver Results dialog. When this box is checked, the extra computations are skipped; the Reports list box will be grayed out in the Solver Results dialog, and you won't be able to select any reports for this run.

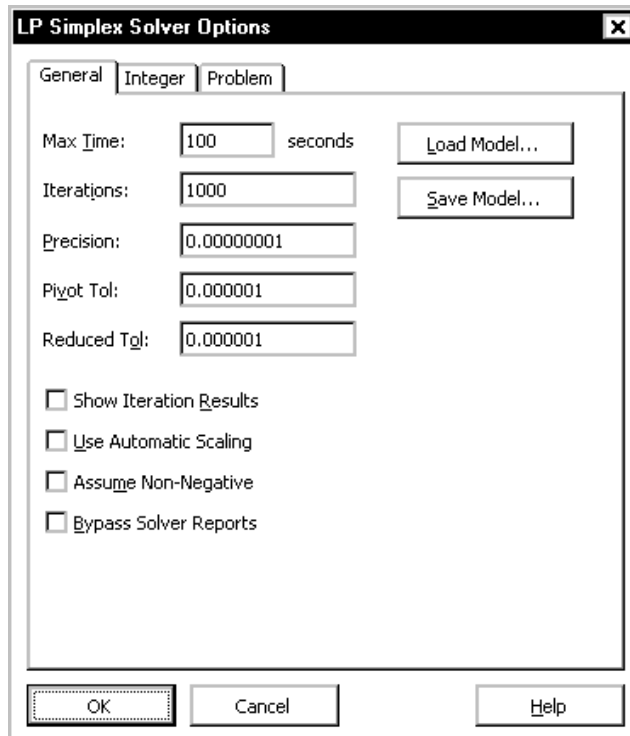
Even though report generation in the Premium Solver and Premium Solver Platform is very fast, the Bypass Solver Reports option can make a real difference in your total solution time, especially when you are solving larger models. In the Premium Solver Platform, it is also supported by many field-installable Solver engines that can solve problems with hundreds of thousands of variables and constraints. It is not unusual for the extra report-related computations to take as much time as the *entire solution process*, especially if you have taken other steps (such as using the functions recognized for fast problem setup) to ensure the best possible solution times.

---

## LP Simplex Solver Options

In the Premium Solver, if the linear Simplex Solver is selected from the Solver engine dropdown list, and the Options button is clicked, the Solver Options dialog shown on the next page is displayed.

The General tab of this dialog contains all of the Common Solver Options discussed earlier, plus the Pivot Tolerance and Reduced Cost Tolerance options, which are used by the Simplex Solver.



## Pivot Tolerance

*VBA / SDK:* Parameter Name "PivotTol",  $0 < \text{value} < 1$

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

*VBA / SDK:* Parameter Name "ReducedTol",  $0 < \text{value} < 1$

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.

## Options for Mixed-Integer Problems

Clicking the Integer tab in the LP Simplex Solver Options dialog displays a set of options for mixed-integer linear programming problems, as described in the major section "Options for Mixed-Integer Problems" later in this chapter. These options control the Branch and Bound method for mixed-integer problems, described in that section.

---

## LP/Quadratic Solver Options

In the Premium Solver Platform, if the LP/Quadratic Solver is selected from the Solver engine dropdown list, and the Options button is clicked, the Solver Options dialog shown on the next page is displayed.

The General tab of this dialog contains all of the Common Solver Options discussed earlier except the Precision edit box, plus the Primal Tolerance, Dual Tolerance, Do Presolve, and Derivatives options, which are specific to the LP/Quadratic Solver. The Bypass Solver Reports check box is worth noting here, since it can have a large impact on solution time. 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.

The screenshot shows the 'LP/Quadratic Solver Options' dialog box with the 'General' tab selected. The dialog has three tabs: 'General', 'Integer', and 'Problem'. The 'General' tab contains the following settings:

- Max Time: 1000
- Iterations: 1000
- Primal Tolerance: 0.0000001
- Dual Tolerance: 0.0000001
- ☐ Show Iteration Results
- ☐ Use Automatic Scaling
- ☐ Assume Non-Negative
- ☐ Bypass Solver Reports
- ☒ Do Presolve
- Buttons: Load Model..., Save Model...
- Derivatives section:
  - ☒ Forward
  - ☐ Central

At the bottom are 'OK', 'Cancel', and 'Help' buttons.

## Primal Tolerance and Dual Tolerance

*VBA / SDK:* Parameter Names "PrimalTolerance", "DualTolerance",  $0 < \text{value} < 1$

The Primal Tolerance is the maximum amount by which the primal constraints can be violated and still be considered feasible. The Dual Tolerance is the maximum amount by which the dual constraints can be violated and still be considered feasible. The default values of  $1.0\text{E-}7$  for both tolerances are suitable for most problems.

## Do Presolve

*VBA / SDK:* Parameter Name "Presolve", value 1/True or 0/False

When this box is checked (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

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 Premium Solver Platform, these derivatives may be computed via *automatic differentiation* or via *finite differencing*. For more information, see the section “More on the Polymorphic Spreadsheet Interpreter” in the chapter “Analyzing and Solving Models.”

When you are using the Interpreter (Solve With = PSI Interpreter in the Solver Model dialog), automatic differentiation is used, exact derivative values are computed, and the setting of the Derivatives choice is ignored. When Solve With = Excel Interpreter, the method used for finite differencing is determined by the setting of the Derivatives choice. *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. (Bear in mind that automatic differentiation is much faster than either Forward or Central differencing.)

### Options for Mixed-Integer Problems

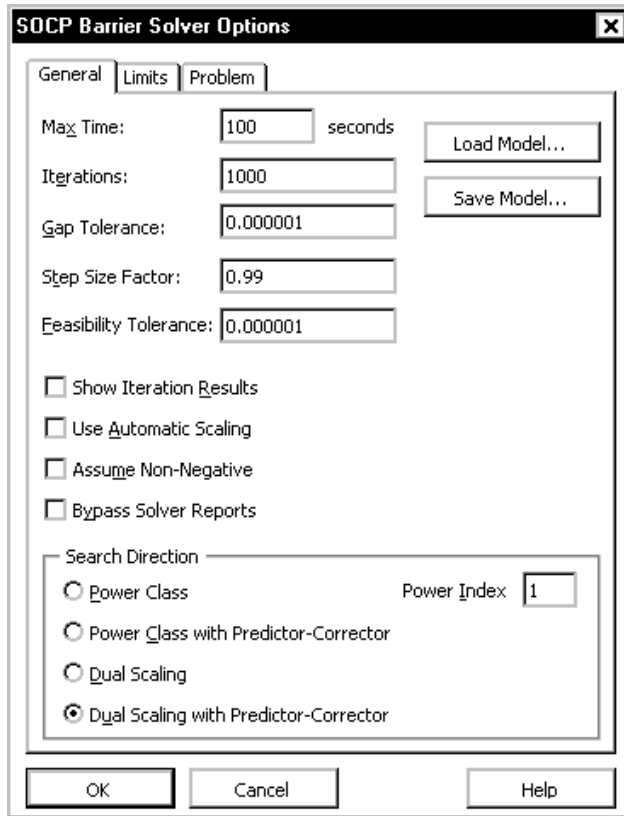
Clicking the Integer tab in the LP/Quadratic Solver Options dialog displays an extensive set of options for mixed-integer linear programming problems, as described in the section “LP/Quadratic Solver Integer Tab Options” later in this chapter. These options control the Branch and Cut method for mixed-integer problems, described in that section.

---

## SOCP Barrier Solver Options

In the Premium Solver Platform, if the SOCP Barrier Solver is selected from the Solver engine dropdown list, and the Options button is clicked, the Solver Options dialog shown on the next page is displayed.

The General tab of this dialog contains all of the Common Solver Options discussed earlier except the Precision edit box, plus the Gap Tolerance, Step Size Factor, and Feasibility Tolerance options and the Search Direction option group, which are specific to the SOCP Barrier Solver.



The image shows a dialog box titled "SOCP Barrier Solver Options" with three tabs: "General", "Limits", and "Problem". The "General" tab is selected. It contains several input fields and checkboxes. The "Max Time" field is set to 100 seconds. The "Iterations" field is set to 1000. The "Gap Tolerance" field is set to 0.000001. The "Step Size Factor" field is set to 0.99. The "Feasibility Tolerance" field is set to 0.000001. There are two buttons on the right: "Load Model..." and "Save Model...". Below these are four checkboxes: "Show Iteration Results", "Use Automatic Scaling", "Assume Non-Negative", and "Bypass Solver Reports". At the bottom, there is a "Search Direction" section with four radio buttons: "Power Class", "Power Class with Predictor-Corrector", "Dual Scaling", and "Dual Scaling with Predictor-Corrector". The "Power Index" field is set to 1. At the very bottom are three buttons: "OK", "Cancel", and "Help".

## Gap Tolerance

*VBA / SDK*: Parameter Name "GapTolerance",  $0 < \text{value} < 1$

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

*VBA / SDK*: Parameter Name "StepSizeFactor",  $0 < \text{value} < .99$

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

*VBA / SDK*: Parameter Name "FeasibilityTolerance",  $0 < \text{value} < 1$

The SOCP Barrier Solver considers a solution feasible if the constraints are satisfied to within this tolerance.

## Search Direction

*VBA / SDK*: Parameter Name "SearchDirection", value 1-Power Class, 2-Power Class with Predictor-Corrector, 3-Dual Scaling, 4- Dual Scaling with Predictor-Corrector

The SOCP Barrier Solver offers four options for computing the search direction on each iteration.

### ***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

*VBA / SDK*: Parameter Name "PowerIndex", integer value  $\geq 0$

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.

### ***Options for Mixed-Integer Problems***

Clicking the Integer tab in the SOCP Barrier Solver Options dialog displays a set of options for mixed-integer linear programming problems, as described in the major section "Options for Mixed-Integer Problems" later in this chapter. These options control the Branch and Bound method for mixed-integer problems, described in that section.

---

## GRG Nonlinear Solver Options

In the Premium Solver and Premium Solver Platform, if the GRG Nonlinear Solver is selected from the Solver engine dropdown list, and the Options button is clicked, the Solver Options dialog shown on the next page is displayed.

The General tab of this dialog contains all of the Common Solver Options discussed earlier, plus several options specific to the GRG Solver, which are described in this section. The Global Optimization options group and the Population Size and



Random Seed edit boxes also appear in this dialog; these options 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.

The screenshot shows the "GRG Nonlinear Solver Options" dialog box. It has three tabs: "General", "Integer", and "Problem". The "General" tab is selected. The dialog contains the following elements:

- Max Time:** 100 seconds
- Iterations:** 1000
- Precision:** 0.000001
- Convergence:** 0.0001
- Population Size:** 0
- Random Seed:** (empty text box)
- Buttons:** "Load Model...", "Save Model..."
- Global Optimization:** A group box containing:
  - ☐ Multistart Search
  - ☐ Topographic Search
  - ☒ Require Bounds on Vars
- Estimates:** A group box containing:
  - ☒ Tangent
  - ☐ Quadratic
- Derivatives:** A group box containing:
  - ☒ Forward
  - ☐ Central
- Search:** A group box containing:
  - ☒ Newton
  - ☐ Conjugate
- Bottom Buttons:** "OK", "Cancel", "Help"

## Convergence

VBA / SDK: Parameter Name "Convergence",  $0 \leq \text{value} \leq 1$

As discussed in the chapter “Diagnosing Solver Results,” the GRG Solver will stop and display the message “Solver has converged to the current solution” 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  $1.0\text{E-}4$  (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 setting in the Convergence box to a smaller value such as  $1.0\text{E-}5$  or  $1.0\text{E-}6$ ; 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.

## Recognize Linear Variables

*VBA / SDK:* Parameter Name "RecognizeLinear", value 1/True or 0/False

This check box activates an “aggressive” strategy to speed the solution of nonlinear problems that may be useful in the Premium Solver, and in the Premium Solver Platform when the Polymorphic Spreadsheet Interpreter is not used (Solve With = Excel Interpreter). As explained in the chapter “Solver Models and Optimization,” a Solver problem is nonlinear (and must be solved with the GRG Solver engine) if the objective or any of the constraints is a nonlinear function of even one decision variable. But in many such 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.

If you check the Recognize Linear Variables box, the 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 display the message “The linearity conditions required by this Solver engine are not satisfied.” If you receive this message, you should uncheck the Recognize Linear Variables box and re-solve the problem.

In the Premium Solver Platform, when the Interpreter is used (which is the default), checking this box will not save any time, because partial derivatives are computed via *automatic differentiation* rather than *finite differencing*. The field-installable Large-Scale GRG Solver, Large-Scale SQP Solver, and KNITRO Solver engines are designed to take advantage of information provided by the Interpreter, and will exploit partial linearity in the problem functions much more effectively than the GRG Solver with the Recognize Linear Variables option.

## 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 GRG Solver. For more information, consult the academic papers on the GRG method listed at the end of the Introduction.

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 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

VBA / SDK: Parameter Name "Estimates", value 1-Tangent or 2-Quadratic

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

VBA / SDK: Parameter Name "Derivatives", value 1-Forward or 2-Central

On each major iteration, the GRG Solver requires values for the gradients of the objective and constraints (i.e. the Jacobian matrix). In the Premium Solver Platform, these derivatives may be computed via *automatic differentiation* or via *finite differencing*. In the Premium Solver, only finite differencing is available. For more information, see the section "More on the Polymorphic Spreadsheet Interpreter" in the chapter "Analyzing and Solving Models."

In the Premium Solver Platform, when you are using the Interpreter (Solve With = PSI Interpreter), automatic differentiation is used, highly accurate derivative values are computed, and the Derivatives setting is ignored. In the Premium Solver, and in the Premium Solver Platform when Solve With = Excel Interpreter, 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. (Bear in mind that automatic differentiation is much faster than either Forward or Central differencing.)

## Search

VBA / SDK: Parameter Name "SearchOption", value 1-Newton or 2-Conjugate

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. (In the Premium Solver, this would roughly square the number of worksheet recalculations required to solve the problem.) 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.

## Options for Mixed-Integer Problems

Clicking the Integer tab in the GRG Solver Options dialog displays a set of options for mixed-integer linear programming problems, as described in the major section “Options for Mixed-Integer Problems” later in this chapter. These options control the Branch and Bound method for mixed-integer problems, described in that section.

---

## Multistart Search Options

This section discusses the Global Optimization options group and the Population Size and Random Seed edit boxes that appear in the Solver Options dialog for the GRG Solver, as shown below:

The screenshot shows the "GRG Nonlinear Solver Options" dialog box with the "Integer" tab selected. The dialog has three tabs: "General", "Integer", and "Problem". The "Integer" tab contains the following options:

- Max Time: 100 seconds
- Iterations: 1000
- Precision: 0.000001
- Convergence: 0.0001
- Population Size: 0
- Random Seed: (empty box)
- Buttons: Load Model..., Save Model...
- Checkboxes:
  - ☐ Show Iteration Results
  - ☐ Use Automatic Scaling
  - ☐ Assume Non-Negative
  - ☐ Bypass Solver Reports
  - ☐ Recognize Linear Vars
- Global Optimization group:
  - ☐ Multistart Search
  - ☐ Topographic Search
  - ☒ Require Bounds on Vars
- Estimates group:
  - ☒ Tangent
  - ☐ Quadratic
- Derivatives group:
  - ☒ Forward
  - ☐ Central
- Search group:
  - ☒ Newton
  - ☐ Conjugate
- Buttons: OK, Cancel, Help

These options 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. The multistart methods are described under “Global Optimization” in the chapter “Solver Models and Optimization,” and their behavior and stopping rules are further described under “GRG Solver with Multistart Methods” in the chapter “Diagnosing Solver Results.”

## Multistart Search

**VBA / SDK:** Parameter Name "MultiStart", value 1/True or 0/False

If this box is checked, the multistart methods are used to seek a globally optimal solution. If this box is unchecked, 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

*VBA / SDK:* Parameter Name "TopoSearch", value 1/True or 0/False

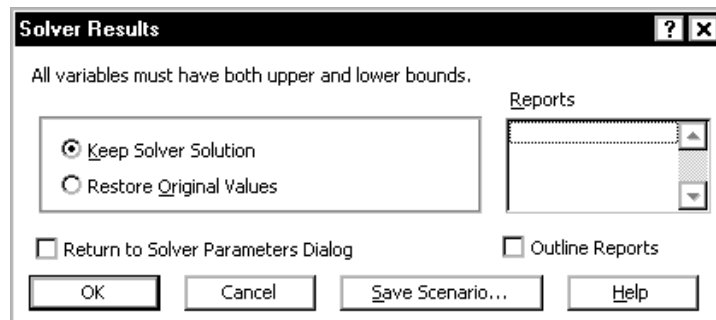
If this box (and the Multistart Search box) are checked, the multistart methods will make use of a “topographic” search method. This method uses the objective value computed for the randomly sampled starting points to determine a “topography” of overall “hills” and “valleys” in the search space, in an effort to find better clusters and start the GRG Solver from an improved point (already in a “hill” or “valley”) in each cluster. Determining the topography takes extra computing 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

*VBA / SDK:* Parameter Name "RequireBounds", value 1/True or 0/False

This box is checked 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 checked 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 boxes are checked, but you have not defined upper and lower bounds on all of the variables, a Solver Results dialog like the one below will be displayed.



When this message appears, you must either add constraints to the Constraints list box (or check the Assume Non-Negative box, to add lower bounds on the variables), or else uncheck the Require Bounds on Variables box, then click Solve again to allow the Solver to proceed with the solution process.

## Population Size

*VBA / SDK*: Parameter Name "PopulationSize", integer value > 0

The multistart methods generate a number of candidate starting points for the GRG Solver equal to the value that you enter in this box. 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 in this box, or leave it blank, 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

*VBA / SDK*: Parameter Name "RandomSeed", integer value > 0

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.

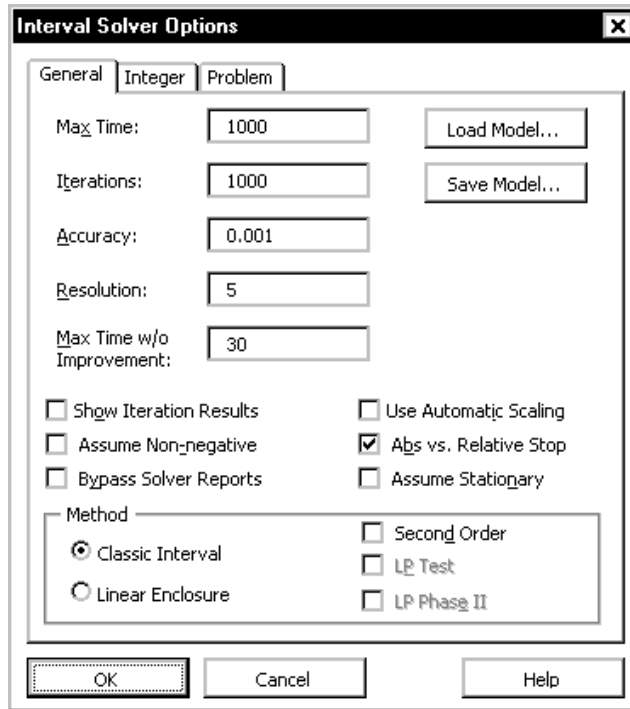
---

## Interval Global Solver Options

In the Premium Solver Platform, when the Interval Global Solver is selected from the Solver engine dropdown list in the Solver Parameters dialog, and the Options button is clicked, a Solver Options dialog like the one shown on the next page will be displayed.

The General tab of this dialog contains all of the Common Solver Options discussed earlier except the Precision edit box, plus several options specific to the Interval Global Solver, which are described in this section.

The default choices for these options are suitable for most problems, but you should experiment with the Method options group to see which methods are *fastest* on your problem. By default, the Interval Global Solver uses only “first order” methods; significant speed gains may be achieved via use of the Second Order and Linear Enclosure methods.



## Accuracy

*VBA / SDK:* Parameter Name "Accuracy",  $0 < \text{value} < 1$

This option plays a role conceptually similar to the Precision option in the other Solver engines. It is used as a tolerance in the Interval Global Solver to determine whether a “box” has been reduced in size to approximately a “point solution,” whether the “distance” between two intervals is sufficiently small, and – in conjunction with the Resolution option below – whether a proposed solution to a system of equations should be treated as distinct from all other known solutions.

## Resolution

*VBA / SDK:* Parameter Name "Resolution",  $0 < \text{value} < 100$

When the Interval Global Solver is seeking all real solutions of a system of nonlinear equations, the Accuracy and Resolution values are used to distinguish one solution from another. A proposed new solution – a “box” consisting of intervals enclosing the decision variables – is compared to all other known solutions. It is considered the same as an existing solution if either the absolute distance between intervals is less than or equal to the Accuracy value or the relative distance between intervals (taking into account their magnitudes) is less than or equal to the Resolution value, for all decision variables. If it is not the same as any existing solution, the new point is accepted as a distinct solution.

## Max Time w/o Improvement

*VBA / SDK:* Parameter Name "MaxTimeNoImp", integer value  $> 0$

The value in this edit box (measured in seconds) is the maximum time that the Interval Global Solver will spend in its search without finding an “improved global solution” (a feasible solution with an objective value better than the currently best

known solution). If this time limit is exceeded, the Solver will stop and display the Solver Results dialog with the message “Solver cannot improve the current solution.” For more information, see “Interval Global Solver Stopping Conditions” in the chapter “Diagnosing Solver Results.”

## Absolute vs. Relative Stop

*VBA / SDK:* Parameter Name "AbsRelStop", value 1/True or 0/False

This option affects the test used to decide whether the globally optimal solution has been found. As described in “Global Optimization” in the chapter “Solver Models and Optimization,” the Interval Global Solver uses an Interval Branch & Bound method that isolates locally optimal solutions and also updates a known best bound on the globally optimal objective function value. The Solver stops when it has found a feasible solution whose objective function value is very close to this best known bound. If the Abs vs. Relative box is checked (the default), the Solver compares the absolute difference between the objective value and the best bound to the Accuracy value. If this box is unchecked, the Solver compares the relative difference (dividing by the objective magnitude) to the Accuracy value.

## Assume Stationary

*VBA / SDK:* Parameter Name "AssumeStationary", value 1/True or 0/False

This option can be used to speed up the Interval Global Solver in situations where you know that the globally optimal solution is a “stationary point” and not a point where a decision variable is equal to its lower or upper bound. The Solver can save a significant amount of time if it does not have to check for possible solutions on the “edges of boxes” where a variable equals one of its bounds. If the Assume Stationary box is checked, the Solver will skip such checks for possible solutions. Of course, this means that if the true global optimum *is* at a point where a decision variable equals a bound, the Solver will probably “miss” this solution and return another point that is not the true global optimum.

## Method Options Group

The Method options group plays a critical role in determining the performance of the Interval Global Solver. As described in “Global Optimization” in the chapter “Solver Models and Optimization,” the Interval Branch & Bound algorithm processes a list of “boxes” that consist of bounded intervals for each decision variable, starting with a single box determined by the bounds that you specify. On each iteration, it seeks lower and upper bounds for the objective and the constraints in a given box that will allow it to discard all or a portion of the box (narrowing the intervals for some of the variables), by proving that the box can contain no feasible solutions, or that it can contain no objective function values better than a known best bound on the globally optimal objective. Boxes that cannot be discarded are subdivided into smaller boxes, and the process is repeated. Eventually, the boxes that remain each enclose a locally optimal solution, and the best of these is chosen as the globally optimal solution.

To obtain good bounds on function values in a box, the Interval Global Solver uses a first-order approximation to the problem functions, using interval values and interval gradients computed by the Interpreter in the Premium Solver Platform. Two rather different first-order approximations can be used: The “classic interval” or *mean value form* and the *linear enclosure form*. With each form, different advanced methods can be used, as discussed below. The “classic interval” form is the default,



but it *pays to experiment* with both forms to see which one performs best on your model. In addition to these first-order approximations, the Solver always uses local constraint propagation methods (also known as *hull consistency* methods) that narrow intervals at each stage of evaluation of the problem functions.

## **Classic Interval vs. Linear Enclosure**

VBA / SDK: Parameter Name "Method", value 1-Classic Interval or 2-Linear Enclosure

The Classic Interval and Linear Enclosure options form a “radio button group,” so that only one of these two options is selected. Classic Interval (the default) uses methods described in the research literature for a number of years; Linear Enclosure uses recently published methods that are implemented for the first time, to Frontline Systems knowledge, in the Interval Global Solver.

When **Classic Interval** is selected, the Solver uses the *mean value form* (based on the interval gradient) as a first-order approximation of the problem functions. This form is especially useful in tests that enable the Solver to rapidly “shrink” a box. With this form, second order methods can be used as described below, but these are optional since they require evaluation of the interval Hessian.

When **Linear Enclosure** is selected, the Solver uses the *linear enclosure form* as a first-order approximation of the problem functions. The linear enclosure doesn't use the interval gradient directly, but it computes similar information to completely enclose the function within linearized boundaries. This form does not readily lend itself to the classic interval second order methods, but because it *completely encloses* the function, it can be used to enable the Solver to rapidly discard many boxes.

## **Second Order**

VBA / SDK: Parameter Name "SecondOrder", value 1/True or 0/False

When this box is checked (and Classic Interval is selected – otherwise it is grayed out), the Interval Global Solver uses a variant of the Interval Newton method (analogous to Newton's method for real numbers, but operating over intervals), employing the Krawczyk operator at its key step, to rapidly find an interval minimum for the objective and shrink or discard the current box, or to rapidly determine whether a solution to a system of equations exists in the current box. Use of the Krawczyk operator requires the interval Hessian, which is computed by the Interpreter via reverse automatic differentiation.

## **LP Test**

VBA / SDK: Parameter Name "LPTest", value 1/True or 0/False

When this box is checked (and Linear Enclosure is selected – otherwise it is grayed out), the Solver internally creates a series of linear programming problems, using the linear enclosures of the original problem's constraints and the bounds on the current box, and applies Phase I of the Simplex method to this problem. If the Simplex method finds no feasible solutions, then the original problem's constraints also have no feasible solutions in the current box, and this box or region can be discarded in the overall Interval Branch & Bound algorithm.

## **LP Phase II**

VBA / SDK: Parameter Name "LPPhaseII", value 1/True or 0/False

When this box is checked (and Linear Enclosure is selected – otherwise it is grayed out), the Solver proceeds as just described for the LP Test option, but it also uses Phase II of the Simplex method to seek an improved bound on the objective function in the current box. If this improved bound is feasible in the original problem, it is used to update the known best bound on the globally optimal objective in the overall Interval Branch & Bound algorithm.

## Evolutionary Solver Options

In the Premium Solver and Premium Solver Platform, when the Evolutionary Solver is selected from the Solver engine dropdown list, and the Options button is clicked, the Solver Options dialog shown below is displayed.

This dialog contains all of the Common Solver Options discussed earlier (the Convergence option has a special meaning for the Evolutionary Solver, as discussed below), plus several options specific to the Evolutionary Solver.

As with the other Solver engines, the Max Time option determines the maximum amount of time the Evolutionary Solver will run before displaying a dialog box asking whether the user wants to continue. The Iterations option rarely comes into play, because the Evolutionary Solver always uses the Max Subproblems and Max Feasible Solutions options on the Limits tab, 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

*VBA / SDK:* Parameter Name "Convergence",  $0 \leq \text{value} \leq 1$

As discussed in the chapter “Diagnosing Solver Results,” the Evolutionary Solver will stop and display the message “Solver has converged to the current solution” 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 message “Solver has converged to the current solution” is appearing prematurely, 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

*VBA / SDK:* Parameter Name "PopulationSize", integer value  $> 0$

As described in the chapter “Solver Models and Optimization,” 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 leave the edit box blank, 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 mutation 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

*VBA / SDK:* Parameter Name "MutationRate",  $0 \leq \text{value} \leq 1$

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 “mutation-preserving” mutation strategies for variables that are members of an “all different” 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

VBA / SDK: Parameter Name "RandomSeed", integer value > 0

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, enter a positive integer value into this box; this value will then be used to “seed” the random number generator each time you click Solve.

## Require Bounds on Variables

VBA / SDK: Parameter Name "RequireBounds", value 1/True or 0/False

If the check box “Require Bounds on Variables” is selected, and some of the decision variables do not have upper or lower bounds specified in the Constraints list box (or via the Assume Non-Negative option) at the time you click Solve, the Solver will stop immediately with the message “All variables must have both upper and lower bounds” – as illustrated in the section “Multistart Search Options” earlier in this chapter. If this option is not selected, the Solver will not require upper and lower bounds on the variables, but will attempt to solve the problem without them. Note that this box is *checked 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, and enter them in the Constraints list box.

## Local Search

VBA / SDK: Parameter Name "LocalSearch", value 1-Randomized Local Search, 2-Deterministic Pattern Search, 3-Gradient Local Search, 4-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 Premium Solver Platform, 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 Premium Solver Platform. In the Premium Solver, this option is equivalent to Randomized Local Search; Deterministic Pattern Search or Gradient Local Search must be selected manually. In the Premium Solver Platform, 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***

*VBA / SDK:* Parameter Name "FixNonSmooth", value 1/True or 0/False

In the Premium Solver Platform, this option determines how non-smooth variable occurrences in the problem will be handled during the local search step. In the Premium Solver, this option is ignored. If this box is checked, 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 box is unchecked, 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 box is unchecked by default; you can experiment with its setting on your problem.

The behavior of the Fix Nonsmooth Variables option on a small group of special functions – currently ABS, IF, MAX, MIN and SIGN – is affected by the setting of the Require Smooth check box in the Solver Model dialog, as described in “Using Analyzer Advanced Options” in the chapter “Analyzing and Solving Models.” When the Require Smooth box is unchecked (the default), checking the Fix Nonsmooth Variables box will *not* fix variables occurring in these special functions.

### ***Filtered Local Search***

In the Premium Solver Platform, the Solver 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 Premium Solver Platform 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.

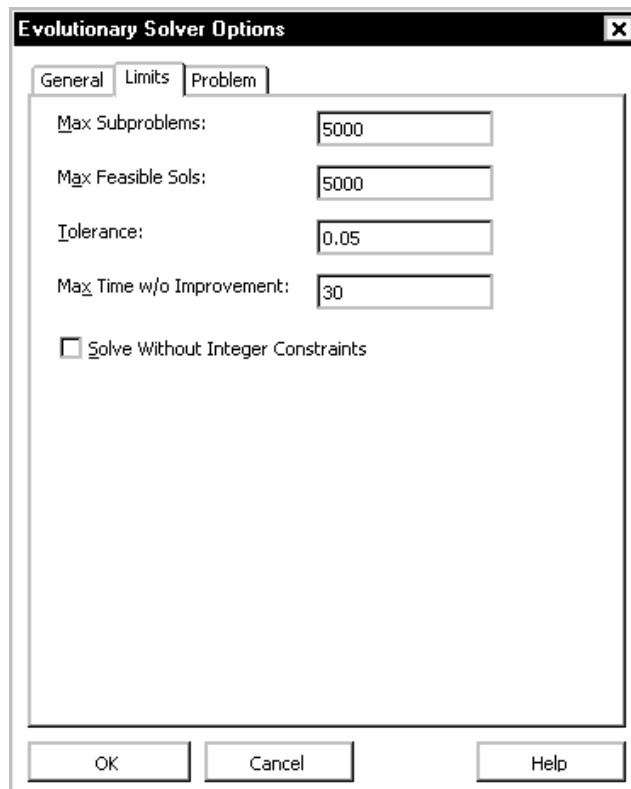
In addition to the Local Search options, the Evolutionary Solver employs a set of methods, corresponding to the four local search methods, to transform infeasible solutions – generated through mutation and crossover – into feasible solutions in new regions of the search space. These methods, which also 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 Premium Solver products is unusually strong in its ability to deal with a combination of constraints and non-smooth functions.

For the reasons described in “Using Analyzer Advanced Options” in the chapter “Analyzing and Solving Models,” **if the Evolutionary Solver stops with the message “Solver encountered an error computing derivatives,” you should check the Analyzer Sparse box in the Model dialog, and click Solve again.**

---

## Limits Tab Options

The Solver Options dialog for the Evolutionary Solver includes a Limits tab. When you click this tab, the options shown below are displayed.



The screenshot shows the 'Evolutionary Solver Options' dialog box with the 'Limits' tab selected. The dialog has three tabs: 'General', 'Limits', and 'Problem'. The 'Limits' tab contains the following settings:

Option	Value
Max Subproblems:	5000
Max Feasible Sols:	5000
Tolerance:	0.05
Max Time w/o Improvement:	30

There is also an unchecked checkbox labeled 'Solve Without Integer Constraints'. At the bottom of the dialog are three buttons: 'OK', 'Cancel', and 'Help'.

Where the other Solver engines use the Branch & Bound method to solve problems with integer constraints, subject to limits set in the Integer Options dialog tab, the Evolutionary Solver handles integer constraints on its own, and is subject to the limits set in this dialog tab. 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

*VBA / SDK:* Parameter Name "MaxSubProblems", integer value > 0

The value in the Max Subproblems edit box places a limit on the number of subproblems that may be explored by the evolutionary algorithm before the Solver pauses and asks you whether to continue, stop or restart the solution process. 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. During the solution process, the number of subproblems considered so far is shown on the Excel status bar, along with the objective of the best feasible solution (if any) found so far. If your model is moderately large or complex, you may need to increase this limit from its default value; any value up to 2,147,483,647 may be used.

## Max Feasible Solutions

VBA / SDK: Parameter Name "MaxIntegerSols", integer value  $> 0$

The value in the Max Feasible Sols edit box places a limit on the number of feasible solutions found by the evolutionary algorithm before the Solver pauses and asks you whether to continue, stop or restart the solution process. A feasible solution is any solution that satisfies all of the constraints, including any integer constraints. As with the Max Subproblems option, if your model is moderately large or complex, you may need to increase this limit; any value up to 2,147,483,647 may be used.

## Tolerance

VBA / SDK: Parameter Name "IntTolerance",  $0 \leq \text{value} \leq 1$

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 value for the number of seconds in the Max Time without Improvement edit box, 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

VBA / SDK: Parameter Name "MaxTimeNoImp", integer value  $> 0$

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 value for the number of seconds in the Max Time without Improvement edit box, the Evolutionary Solver stops and displays the Solver Results dialog. The message is "Solver cannot improve the current solution," unless the evolutionary algorithm has discovered no feasible solutions at all, in which case the message is "Solver could not find a feasible solution." 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

VBA / SDK: Parameter Name "SolveWithout", value 1/True or 0/False

When you click the Solve button (in the Solver Parameters dialog) while this box is checked, the Solver *ignores* integer constraints (including alldifferent constraints) and solves the "relaxation" of the problem. It is often useful to solve the relaxation, and it is much more convenient to check this box than to delete the integer constraints and add them back again later.

This option remains in effect until you uncheck the Solve Without Integer Constraints box. As discussed further under "The Integer Options Dialog Tab" below, when you solve an integer programming problem (without this option) and the Solver finds no feasible integer solution, you are offered the option of solving the relaxation on a "one-time-only" basis in the Solver Results dialog.



---

## Integer Tab Options

In the Premium Solver and Premium Solver Platform, the Solver Options dialogs for all of the bundled Solver engines except the Evolutionary Solver include an Integer tab. When you click this tab, options for mixed-integer programming problems are displayed. This section describes Integer tab options for all Solver engines *except* the LP/Quadratic Solver in the Premium Solver Platform; the LP/Quadratic Solver's Integer tab options are described in the following section.

The screenshot shows the 'LP Simplex Solver Options' dialog box with the 'Integer' tab selected. The dialog has three tabs: 'General', 'Integer', and 'Problem'. The 'Integer' tab contains the following options:

- Max Subproblems: 1000
- Max Feasible Sols: 1000
- Tolerance: 0.05
- Integer Cutoff: (empty)
- ☐ Solve Without Integer Constraints
- ☒ Use Dual Simplex for Subproblems
- Preprocessing and Probing:
  - ☐ Probing / Feasibility
  - ☐ Optimality Fixing
  - ☐ Bounds Improvement
  - ☐ Primal Heuristic
- Gomory Cuts: 20
- Passes: 1
- Knapsack Cuts: 20
- Passes: 1

At the bottom are buttons for 'OK', 'Cancel', and 'Help'.

This dialog collects all of the options that pertain to the solution of problems with integer constraints. The controls below the “Solve Without Integer Constraints” check box appear *only* for the LP Simplex Solver in the Premium Solver.

### Max Subproblems

*VBA / SDK:* Parameter Name "MaxSubProblems", integer value > 0

The value in the Max Subproblems edit box places a limit on the number of subproblems that may be explored by the Branch & Bound algorithm before the Solver pauses and asks you whether to continue or stop the solution process. (The Branch & Bound method is briefly described in the chapter “Solver Models and Optimization.”) Each subproblem is a “regular” Solver problem with additional bounds on the variables.

*Note:* A “Restart” button also appears in the dialog box where the Solver asks you whether you want to continue or stop the solution process; but this button is meaningful only for the GRG Nonlinear Solver, and it affects only restarting of the current subproblem. The Branch & Bound algorithm is never restarted, as this would simply mean discarding the progress that has been made so far.

In a problem with integer constraints, the Max Subproblems limit should be used in preference to the Iterations limit in the Solver Options dialog; the Iterations limit should be set high enough for each of the individual subproblems solved during the Branch & Bound process. For problems with many integer constraints, you may need to increase this limit from its default value; any integer value up to 2,147,483,647 may be used.

## Max Integer Solutions

*VBA / SDK:* Parameter Name "MaxIntegerSols", integer value > 0

The value in the Max Integer Solutions edit box places a limit on the number of “candidate” integer solutions found by the Branch & Bound algorithm before the Solver pauses and asks you whether to continue or stop the solution process. Each candidate 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 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. You can set this limit to any integer value up to 2,147,483,647.

## Integer Tolerance

*VBA / SDK:* Parameter Name "IntTolerance",  $0 \leq \text{value} \leq 1$

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. In the Premium Solver and Premium Solver Platform, the Solver Result Message will be “Solver found an integer solution within tolerance.” 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 may take a great deal of computing time.

## Integer Cutoff

VBA / SDK: Parameter Name "IntCutoff",  $-1E30 < \text{value} < +1E30$

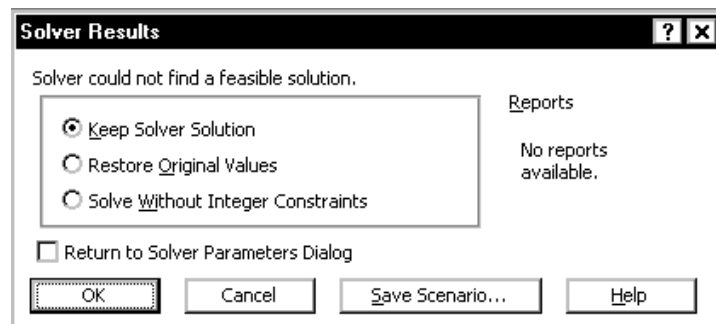
This option provides another way to save time in the solution of mixed-integer programming problems. If you know the objective value of a feasible integer solution to your problem – possibly from a previous run of the same or a very similar problem – you can enter this objective value in the Integer Cutoff edit box. 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 here, 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 Solver to skip solving the subproblem that would yield the optimal integer solution.

## Solve Without Integer Constraints

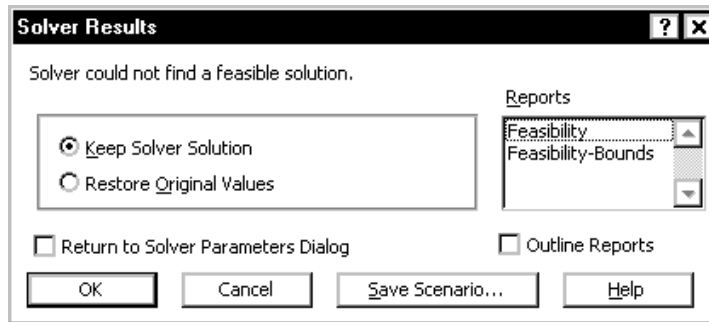
VBA / SDK: Parameter Name "SolveWithout", value 1/True or 0/False

When you click the Solve button (in the Solver Parameters dialog) while this box is checked, the Solver *ignores* integer constraints (including alldifferent constraints) and solves the “relaxation” of the problem. It is often useful to solve the relaxation, and it is much more convenient to check this box than to delete the integer constraints and add them back again later.

This option remains in effect until you uncheck the Solve Without Integer Constraints box. When you solve an integer programming problem (without this option) and the Solver finds no feasible integer solution, you are offered the option of solving the relaxation on a “one-time-only” basis in the Solver Results dialog, as shown below.



When chosen this way, the option is effective for only one solution attempt, when you click OK. It is possible that the relaxation of the problem – ignoring the integer constraints – is still infeasible; in this case, you will next see the Solver Results dialog shown on the following page, which will allow you to produce a Feasibility Report for the relaxation of the problem. Using the Feasibility Report, you can more easily locate and correct the conflicting constraints that make the problem infeasible.



## Use Dual Simplex for Subproblems

*VBA / SDK:* Parameter Name "UseDual", value 1-Primal or 2-Dual (i.e. checked)

This option appears only in the Premium Solver, when the LP Simplex Solver is selected. When this box is checked – note that it is *checked by default* – 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 Premium Solver 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 option group appears only in the Premium Solver, when the LP Simplex Solver 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***

*VBA / SDK:* Parameter Name "ProbingFeasibility", value 1/True or 0/False

The Probing strategy allows the Solver to derive values for certain binary integer variables based on the settings of others, prior to actually solving the problem. When the Branch & Bound method creates a subproblem with an additional (tighter) bound on a binary integer variable, this causes the variable to be fixed at 0 or 1. In many problems, this has implications for the values of other binary integer variables that

can be discovered through Probing. For example, your model may have a constraint such as:

$$x_1 + x_2 + x_3 + x_4 + x_5 = 1$$

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

VBA / SDK: Parameter Name "BoundsImprovement", value 1/True or 0/False

The Bounds Improvement strategy allows the Solver 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

VBA / SDK: Parameter Name "OptimalityFixing", value 1/True or 0/False

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

VBA / SDK: Parameter Name "PrimalHeuristic", value 1/True or 0/False

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.

You may wish to check this box, then watch the Excel status bar while your problem is being solved, and note how long the Solver takes before “Branch” messages appear, and whether an “Incumbent: xxx” appears in these messages. If Branch messages appear without an incumbent value after some delay, it is probably better to uncheck this box, since the primal heuristic is not proving effective on the problem.

## ***Variable Reordering and Pseudocost Branching***

A Variable Reordering option was used in earlier versions of the Premium Solver 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 Premium Solver, which 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 Solver 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 Solver 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 Solver 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 your Changing Cells edit box (from left or right), or in the Variables list box (top to bottom, and left to right). You may be able to improve performance by manually ordering the variables in the Changing Cells box or Variables list, based on your knowledge of the problem. In the example above, you would list the plant-building binary integer variable first in the Changing Cells edit box, 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 Premium Solver. 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

VBA / SDK: Parameter Name "MaxGomoryCuts", integer value > 0

The value in this edit box 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 Solver 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 see the Scaling Report listed as an option in the Solver Results dialog, select it and examine the report contents to help find scaling problems in your model, as described the chapter "Solver Reports." If you have trouble finding the integer optimal solution with the default settings for Gomory cuts, you may want to enter 0 in this edit box, to eliminate Gomory cuts as a possible source of problems due to rounding.

## Gomory Passes

VBA / SDK: Parameter Name "GomoryPasses", integer value > 0

The value in this edit box is the number of "passes" the Solver 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 is possible to solve an LP/MI 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

VBA / SDK: Parameter Name "MaxKnapsackCuts", integer value > 0

The value in this edit box is the maximum number of Knapsack cuts that the Solver should generate for a given subproblem. When this maximum is reached, or if there are no further cut opportunities, the Solver 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

VBA / SDK: Parameter Name "KnapsackPasses", integer value > 0

The value in this edit box 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.

---

## LP/Quadratic Solver Integer Tab Options

This section describes the Integer tab options for the LP/Quadratic Solver in the Premium Solver Platform, which features an extensive set of options to improve performance on mixed-integer programming problems. When the LP/Quadratic Solver is selected from the Solver engine dropdown list, the Options button is clicked, and the Integer tab is clicked, the options shown below will be displayed.

The screenshot shows the 'LP/Quadratic Solver Options' dialog box with the 'Integer' tab selected. The dialog has three tabs: 'General', 'Integer', and 'Problem'. The 'Integer' tab contains the following options:

- Max Subproblems: 5000
- Max Feasible Sols: 5000
- Tolerance: 0.05
- Integer Cutoff: (empty)
- Maximum Cut Passes at Root: -1 in Tree: 10
- ☐ Solve Without Integer Constraints
- ☒ Use Strong Branching
- Cuts & Heuristics section:
  - ☐ Lift and Cover
  - ☐ Rounding
  - ☐ Knapsack
  - ☐ Mixed Integer Rounding
  - ☐ Two Mixed Integer Rounding
  - ☐ Reduce and Split
  - ☐ Local Search Heuristic
  - ☐ Gomory
  - ☒ Probing
  - ☐ Odd Hole
  - ☐ Clique
  - ☐ Flowcover
  - ☐ Special Ordered Sets
  - ☐ Rounding Heuristic

At the bottom are 'OK', 'Cancel', and 'Help' buttons.

### Max Subproblems

VBA / SDK: Parameter Name "MaxSubProblems", integer value > 0

The value in the Max Subproblems edit box places a limit on the number of subproblems that may be explored by the Branch & Bound algorithm before the Solver pauses and asks you whether to continue or stop the solution process. Each subproblem is a "regular" Solver problem with additional bounds on the variables.

In a problem with integer constraints, this limit should be used in preference to the Iterations limit in the Solver Options dialog; the Iterations limit should be set high enough for each of the individual subproblems solved during the Branch & Bound process. For problems with many integer constraints, you may need to increase this limit from its default value; any integer value up to 2,147,483,647 may be used.

### Max Feasible (Integer) Solutions

VBA / SDK: Parameter Name "MaxIntegerSols", integer value > 0



The value in the Max Feasible Sols edit box places a limit on the number of feasible integer solutions found by the Branch & Bound algorithm before the Solver pauses and asks you whether to continue or stop the solution process. 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

VBA / SDK: Parameter Name "IntTolerance",  $0 \leq \text{value} \leq 1$

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 message “Solver found an integer solution within tolerance.” 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 may take a great deal of computing time.

## Integer Cutoff

VBA / SDK: Parameter Name "IntCutoff",  $-1\text{E}30 < \text{value} < +1\text{E}30$

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 in the Integer Cutoff edit box. 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 here, 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 Solver to skip solving the subproblem that would yield the optimal integer solution.

## Maximum Cut Passes

VBA / SDK: Parameter Name "MaxRootCutPasses", integer value  $\geq -1$

Parameter Name "MaxTreeCutPasses", integer value  $\geq -1$

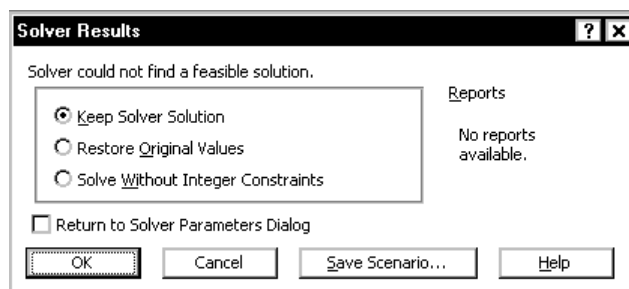
This option determines the maximum number of “passes” carried out to generate cuts, at the root node (immediately after the first LP relaxation is solved), and at nodes deeper in the Branch & Bound tree; it is effective only if one or more of the Cut Generation check box options (see below) are checked. When cuts are added to a problem, the resulting problem may present further opportunities to generate cuts; hence, cut generation “passes” are performed until either no new cuts are found, or the maximum number of passes is reached. A value of -1 in the “at Root” edit box means that the number of passes should be determined automatically. The default value of 10 passes for nodes deeper in the tree is appropriate for many models, but you may wish to try both smaller and larger values in this edit box.

## Solve Without Integer Constraints

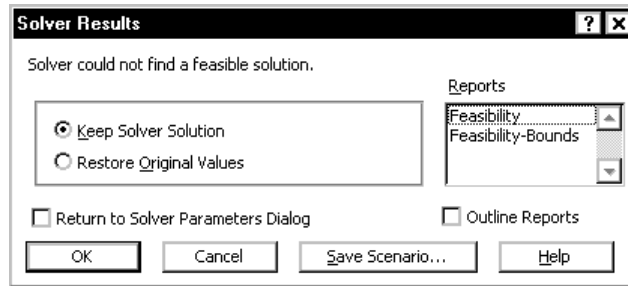
VBA / SDK: Parameter Name "SolveWithout", value 1/True or 0/False

When you click the Solve button in the Solver Parameters dialog while this box is checked, the Solver *ignores* integer constraints (including alldifferent constraints) and solves the “relaxation” of the problem. It is often useful to solve the relaxation, and it is much more convenient to check this box than to delete the integer constraints and add them back again later.

This option remains in effect until you uncheck the Solve Without Integer Constraints box. In the Premium Solver Platform, when you solve an integer programming problem (without this option) and the Solver finds no feasible integer solution, you are offered the option of solving the relaxation on a “one-time-only” basis in the Solver Results dialog, as shown below.



When chosen this way, the option is effective for only one solution attempt, when you click OK. It is possible that the relaxation of the problem – ignoring the integer constraints – is still infeasible; in this case, you will next see the Solver Results dialog on the next page, which will allow you to produce a Feasibility Report for the relaxation of the problem. Using the Feasibility Report, you can more easily locate and correct the conflicting constraints that make the problem infeasible.



## Use Strong Branching

VBA / SDK: Parameter Name "StrongBranching", value 1/True or 0/False

When this box is checked, the Solver performs strong branching at the root node. *Strong Branching* is a method used to estimate the impact of branching on each integer variable on the objective function (its *pseudocost*), by performing a few iterations of the Dual Simplex method. Such pseudocosts are used to guide the choice of the next subproblem to explore, and the next integer variable to branch upon, throughout the Branch and Bound process. Time spent in strong branching is often repaid many times over in a reduction of the number of nodes that must be explored to find the integer optimal solution.

## Cuts & Heuristics

The LP/Quadratic Solver in the Premium Solver Platform supports a wide range of cuts and heuristics. 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. A *heuristic* is a strategy that often – but not always – will find a reasonably good “incumbent” or feasible integer solution early in the search. Cuts and 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.

### Lift and Cover Cuts

VBA / SDK: Parameter Name "LiftAndCoverCuts", value 1/True or 0/False

When this box is checked, Lift and Cover cuts may be generated. These cuts are somewhat expensive to compute, but when they 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.

### Rounding Cuts

VBA / SDK: Parameter Name "RoundingCuts", value 1/True or 0/False

When this box is checked, Rounding cuts may be generated. A Rounding cut is an inequality in all integer variables formed by netting out any continuous variables, dividing through by the greatest common denominator (gcd) of the coefficients, and rounding down the right hand side.

### Knapsack Cuts

VBA / SDK: Parameter Name "KnapsackCuts", value 1/True or 0/False

When this box is checked, Knapsack cuts may be generated. Like Lift and Cover cuts, these cuts are somewhat expensive to compute, but when they 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.

### ***Gomory Cuts***

*VBA / SDK*: Parameter Name "GomoryCuts", value 1/True or 0/False

When this box is checked, Gomory cuts may be generated. Gomory cuts are found 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. The LP/Quadratic Solver has very good methods for minimizing the effects of such errors, but in rare cases, you may want to reduce the Maximum Cut Passes value when using Gomory cuts, to minimize or eliminate possible problems due to rounding.

### ***Probing Cuts***

*VBA / SDK*: Parameter Name "ProbingCuts", value 1/True or 0/False

When this box is checked, Probing cuts may be generated. This process is similar to the Preprocessing and Probing methods used in the LP Simplex Solver. Probing involves setting certain binary integer variables to 0 or 1 and deriving values for other binary integer variables, or tightening bounds on the constraints.

### ***Odd Hole Cuts***

*VBA / SDK*: Parameter Name "OddHoleCuts", value 1/True or 0/False

When this box is checked, Odd Hole cuts (also called odd cycle cuts) may be generated, using a method due to Grotschel, Lovasz and Schrijver. These cuts apply only to constraints that are sums of binary integer variables.

### ***Mixed Integer Rounding Cuts***

*VBA / SDK*: Parameter Name "MirCuts", value 1/True or 0/False

When this box is checked, Mixed Integer Rounding cuts may be generated.

### ***Two Mixed Integer Rounding Cuts***

*VBA / SDK*: Parameter Name "TwoMirCuts", value 1/True or 0/False

When this box is checked, Two Mixed Integer Rounding cuts may be generated.

### ***Clique Cuts***

*VBA / SDK*: Parameter Name "CliqueCuts", value 1/True or 0/False

When this box is checked, Clique cuts may be generated. Cuts for both row cliques and start cliques are generated, using a method due to Hoffman and Padberg.

### ***Flow Cover Cuts***

*VBA / SDK*: Parameter Name "FlowCoverCuts", value 1/True or 0/False

When this box is checked, Flow Cover cuts may be generated.

### ***Reduce and Split Cuts***

VBA / SDK: Parameter Name "RedSplitCuts", value 1/True or 0/False

When this box is checked, Reduce and Split cuts may be generated. These cuts are variants of Gomory cuts.

### ***Special Ordered Sets***

VBA / SDK: Parameter Name "SOSCuts", value 1/True or 0/False

This strategy scans the model for constraints of the form  $x_1 + x_2 + \dots + x_n = 1$  where all of the variables  $x_i$  are binary integer variables. Such constraints often arise in practice, and are sometimes called "special ordered sets." In any feasible solution, exactly one of the variables  $x_i$  must be 1, and all the others must be 0; hence only  $n$  possible permutations of values for the variables (rather than  $2^n$ ) need be considered.

### ***Local Search Heuristic***

VBA / SDK: Parameter Name "LocalHeur", value 1/True or 0/False

When this box is checked, a "local search" heuristic, similar to the Primal Heuristic discussed above for the LP Simplex Solver, is used to seek possible integer solutions (by adjusting the values of individual integer variables) in the "vicinity" of a known integer solution.

### ***Rounding Heuristic***

VBA / SDK: Parameter Name "RoundingHeur", value 1/True or 0/False

When this box is checked, a "rounding" heuristic is used to seek possible integer solutions (by adjusting the values of individual integer variables) in the "vicinity" of a known integer solution.

---

## **The Problem Tab**

In the Premium Solver and Premium Solver Platform, each Solver Options dialog includes a Problem tab. Clicking on this tab displays statistics on the size of the current problem and the corresponding Solver engine size limits, including the number of decision variables, number of constraints, number of bounds on the variables, and number of integer variables. These edit controls are "read-only" – the current problem sizes are computed automatically, and the Solver engine size limits are obtained automatically from both built-in and field-installable Solver engines.

When the LP/Quadratic Solver is selected from the Solver engine dropdown list, the Options button is clicked, and the Problem tab is clicked, the options shown on the next page will be displayed. The LP/Quadratic Solver supports linear and quadratic programming problems of up to 8,000 variables, 8,000 constraints, and 2,000 integer variables.

LP/Quadratic Solver Options			
General Integer Problem			
Current Problem			
Variables	Constraints	Bounds	Integers
0	0	0	0
Solver Engine Size Limits			
Variables	Constraints	Bounds	Integers
8000	8000	16000	2000
OK Cancel Help			

## Loading, Saving and Merging Solver Models

The Solver Options dialogs also include Load Model... and Save Model... buttons, which allow you to save and restore the specifications (variable, constraint and objective cell selections plus option settings) of a Solver model on the worksheet. The model specifications are stored as formulas in cells, which include references to the variable, constraint and objective cells and values for the Solver options.

The “current” Solver model defined for each worksheet is automatically saved “behind the scenes” in that worksheet. So it is not necessary to use this feature to keep track of a single Solver model – the last set of specifications you defined will be saved automatically when the workbook is saved, and restored when it is re-opened. But the Load Model... and Save Model... buttons can be used to save more than one Solver model on the same worksheet, and to merge two models into one.

### Saved Model Formats

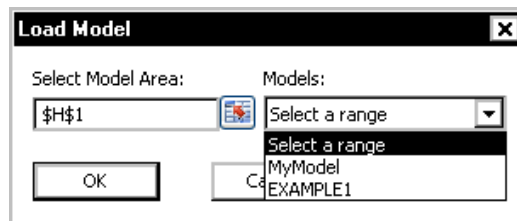
The model specifications can be stored as formulas in two formats: The **Classic** format, which is upward compatible from the standard Excel Solver, uses certain built-in Excel functions and array formulas to store the model; it is not intended for user modification. The **Psi Function** format, introduced in Version 7.0, uses add-in functions such as PsiVar(), PsiCon(), PsiObj() and PsiOption() to store the model; it is described in “Defining Your Model with PSI Functions” in the chapter “Building Solver Models.” Since it is a fully documented format, it may be used to “import” models that are created manually or with other programs.

## Competitive Products

A third alternative is available for users upgrading to the Premium Solver Platform from certain competitive software products: If you click the Load Model button and there *is no* model for the Premium Solver Platform defined in the workbook, but there *is* a model defined for a recognized competitive software product, the Solver will list this model and allow you to select it for loading. When you click OK, the model is loaded and converted – it becomes the “current” Solver model defined for the active worksheet, for the Premium Solver Platform.

## Model Names

When you use **Psi functions** to save a model, you can include an argument in each Psi function call that gives a name such as “MyModel” to a set of model specifications. When you do this, the formula cells containing Psi functions need not be contiguous on the spreadsheet; they are associated via the model name. When you click the Load Model button, the Solver offers you a choice of available models:



The “current” Solver model also has a name, which is the same as the worksheet name. You can choose any of the named models, or you can choose **Select a range** and load a set of (usually Classic format) specifications from a contiguous cell range.

## Using Multiple Solver Models

It is possible – and often useful – to define more than one Solver model based on the same worksheet formulas. An example of this is provided in the “Portfolio of Securities” worksheet in the *SOLV SAM P.XLS* workbook that is included with Microsoft Excel. This worksheet defines a portfolio optimization model, where the Solver must determine what percentage of available funds to invest in four different stocks (A, B, C and D) and Treasury bills. The worksheet formulas calculate the portfolio rate of return, and the portfolio risk as measured by the statistical variance of returns. There are two possible approaches to solving this model: (1) Find the maximum rate of return, subject to an upper limit on the portfolio risk, or (2) Find the minimum risk (variance), subject to a lower limit on the portfolio return.

The “current” Solver problem on this worksheet is the one that maximizes return, subject to a constraint on portfolio risk. But both Solver problems (“Maximize Return” and “Minimize Risk”) have been set up and their specifications saved (in Classic format) in the lower part of this worksheet, starting at row 21. If you click on the Load Model... button in the Solver options dialog, select cells D21:D29, and click OK, you will load the specifications for the problem that minimizes risk subject to a constraint on return.

## Transferring Models Between Spreadsheets

Another application of Load Model... and Save Model... is to transfer Solver model specifications from one worksheet to another. You can do this, not only between Excel worksheets, but between worksheets created in Excel 2000, XP, 2003 or 2007

– or created with a Premium Solver product – and worksheets created in Lotus 1-2-3 97 or Millenium Edition – which features a Solver designed by Frontline Systems.

When you open a Lotus 1-2-3 worksheet in Excel, most or all of the formulas and cell formats will be converted automatically – but information about the “default” Solver model is not transferred or converted. With the Save Model... and Load Model... functions, however, you can transfer the specifications of the Solver model as formulas on the worksheet. You can do this in either direction, as outlined below.

### ***From 1-2-3 to Excel***

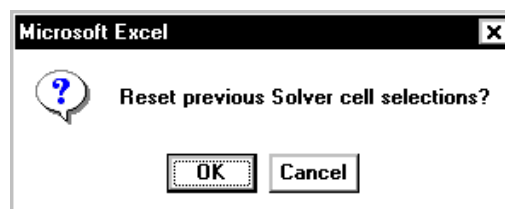
1. Starting from the Solver Options dialog, click the Save Model button and save your Solver model specifications in a cell range.
2. Choose File Save As... and save the workbook with the Solver model specifications in 1-2-3 Release 5 (WK4) format.
3. Open the saved WK4 file in Excel.
4. Choose Tools Premium Solver to display the Solver Parameters dialog. Click the Options button to display the Solver Options dialog.
5. Click Load Model, and select the cell range containing the model specifications.

### ***From Excel to 1-2-3***

1. *First*, choose File Save As... and save your Excel workbook in WK4 format. This step gives a “hint” to the Excel Solver to use a model specification format that can be read by both Excel and 1-2-3.
2. Starting from the Solver Options dialog, click the Save Model button to save your Solver model specifications in a cell range.
3. Choose File Save to save the WK4 file with the Solver model specifications.
4. Open the saved WK4 file in 1-2-3.
5. Choose Range Analyze Solver to display the Solver Parameters dialog. Click the Options button to display the Solver Options dialog.
6. Click Load Model, and select the cell range containing the model specifications.

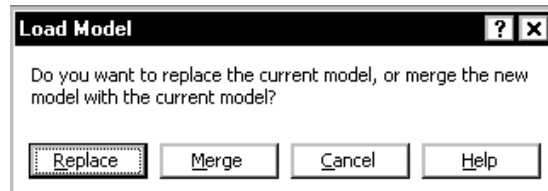
## **Merging Solver Models**

In the standard Microsoft Excel Solver and earlier versions of the Premium Solver products, loading a model's specifications through Load Model... causes any existing specifications for the “current” Solver model to be erased. You are prompted before this happens with the alert box shown below.





In recent versions of the Premium Solver products, you have another choice: You can merge the model specifications being loaded with the current model specifications. Where the specifications necessarily overlap – as in the selection of the objective (and the “m a x i m i z e” or “m i n i m i z e” setting) and in the settings of Solver options – the newly loaded specifications take precedence. But the variable cell selections and the constraint left hand sides, relations and right hand sides being loaded are merged into the current model. You are prompted to choose between replacing the current model specifications and merging in the new specifications:



Merging model specifications can be quite useful, for it allows you to build and test smaller, simple Solver models and then combine them into a larger model. Suppose, for example, that you wanted to create a planning model for a manufacturing firm that would take into account both the mix of products being built and the routes along which they were being shipped. You might create two models on one worksheet, one based on the Product Mix example and the other based on the Shipping Routes example in SOLVSAMP.XLS, and test them individually. Then you could combine them with the Merge function, and test the production-distribution model as a whole.



# Solver Reports

---

## Introduction

This chapter will help you use the information in the Solver Reports, which can be produced when the Solver finds a solution – or when it *fails* to find a solution, and instead reports that the linearity conditions are not satisfied, or that your model is infeasible. We'll explain how to interpret the values in the Answer, Sensitivity and Limits Reports, available in the standard Excel Solver and the Premium Solver products, and how to use the diagnostic Scaling, Linearity and Feasibility Reports and the specialized Solutions and Population Reports, which are unique to the Premium Solver Platform. To illustrate the reports, we'll use EXAMPLE1 through EXAMPLE4 in the workbook **Examples.xls**, which you can examine by clicking Help, then the Examples button in the initial Help dialog. For the Solutions Report, we'll use other examples including a historically interesting nonlinear equation.

### ***Structure and Transformation Reports***

In addition to the eight types of reports described in this chapter, the Premium Solver Platform offers two additional reports that are produced by the new Polymorphic Spreadsheet Interpreter, and requested via the **Solver Model** dialog, as explained in the chapter “Analyzing and Solving Models.”

The **Structure Report**, described and illustrated in “Analyzing Model Exceptions,” analyzes in depth the linear, quadratic, smooth nonlinear, and non-smooth variables and functions in your model, and helps you find and fix “exceptional” formulas if you're having difficulty build a linear or quadratic programming model.

The **Transformation Report**, shown in “Transforming Your Non-Smooth Model,” documents how the Polymorphic Spreadsheet Interpreter can automatically transform your model, replacing non-smooth functions such as IF, MIN, MAX, ABS, AND, OR, and NOT with equivalent expressions using new variables and linear constraints.

### ***Answer, Sensitivity and Limits Reports***

The Answer, Sensitivity and Limits Reports are available when the Solver finds an optimal solution for your model; they give you additional information about the solution and its range of applicability. All three reports can be useful, but we recommend that you focus on the Sensitivity Report. When properly interpreted, this report will tell you a great deal about your model and its optimal solution, which you could not easily determine by simply inspecting the final solution values on the worksheet. Using the Sensitivity Report, you can determine what would happen if

you changed your model in various ways and re-ran the Solver, without your having to actually carry out these steps.

In Excel VBA, you can use the new object-oriented API to access the information in the Answer and Sensitivity Reports via the properties `InitialValue`, `FinalValue`, `DualValue`, `DualUpper`, and `DualLower` of the `Variable` and `Function` objects. These objects and properties can also be used in the Solver Platform SDK, outside of Excel. See the chapter “Using the Object-Oriented API” for further information.

## Scaling Report

The Scaling Report – available only in the Premium Solver Platform, since it uses the Polymorphic Spreadsheet Interpreter – helps you find and fix poorly scaled formulas in your model. It appears in the Reports list box of the Solver Results dialog when you get a result – such as “Solver could not find a feasible solution,” “Solver could not improve the current solution,” or “The linearity conditions required by this Solver engine are not satisfied” – that generally indicate other conditions, but *may be due to a poorly scaled model*. If you are puzzled by a result, and you see that the Scaling Report is available, we highly recommend that you select it, click OK, and then examine the report contents. This takes only a moment, and it may save you hours of time if it reveals a scaling problem. See “The Scaling Report” below for a realistic example, using the EXAMPLE4 Portfolio Optimization model.

## Linearity and Feasibility Reports

The Linearity and Feasibility Reports -- available in both the Premium Solver and the Premium Solver Platform – help you diagnose problems in your models.

With the Linearity Report, you can pinpoint and, if desired, eliminate nonlinear functions from your model, so that it can be solved with a faster and more reliable linear Solver. Using the object-oriented API or the Solver Platform SDK, you can access the information in the Linearity Report by calling the `Model` object `DependTest` method. In the Premium Solver Platform, the `Structure` Report or the `Model` object `DependCheck` method can provide even more information.

With the Feasibility Report, you can pinpoint the constraints that interact to make your model infeasible, and correct them as needed. Using the object-oriented API or the Solver Platform SDK, you can access the information in the Feasibility Report via the `BoundIndex`, `BoundStatus`, `ConstraintIndex` and `ConstraintStatus` properties of the `OptIIS` object, which is a member of each `Variable` and `Function` object.

## Solutions Report

Where the Answer Report gives you detailed information about the single “best solution” that appears on the worksheet when the Solver Results dialog is displayed, the Solutions Report gives you objective function and decision variable values for a number of alternative solutions, found during the optimization process. For mixed-integer problems, the report shows each „incom b e n t or feasible integer solution found by the Branch & Bound method. For global optimization problems solved with the GRG, LSGRG, LSSQP, and KNITRO Solver engines, the report shows each locally optimal solution found by the Multistart method. For the Evolutionary and OptQuest Solvers, the report shows members of the final population of solutions.

Using the object-oriented API or the Solver Platform SDK (after calling the Solver object `Optimize` method), you can access the information in the Solutions Report by setting the Solver object `SolutionIndex` property to a value between 1 and the `NumSolutions` property value, then accessing the `Value` properties of the `Variable` and `Function` objects.

The Solutions Report has a special meaning for the Interval Global Solver. It is available for problems with no objective function to be maximized or minimized, and with all equality constraints (a *system of equations*) or all inequality constraints (a *system of inequalities*). For a system of nonlinear equations, the Answer Report shows only a single solution, but the Solutions Report shows you *all real solutions*. For a system of inequalities, the Answer Report again shows you only a single feasible point, but the Solutions Report shows you an “inner solution” – a *region* or set of points where all of the constraints are satisfied.

### Population Report

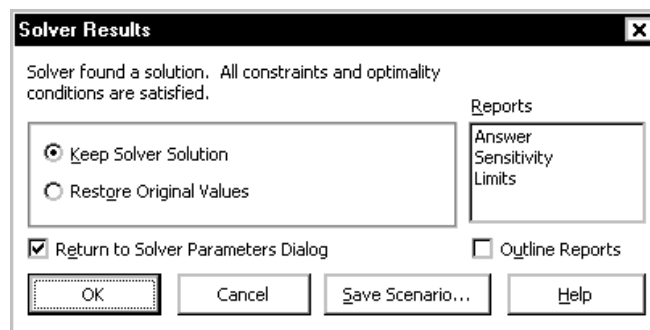
The Population Report is supported by the Evolutionary Solver; it gives you summary statistical information about the entire population of candidate solutions maintained by the Evolutionary Solver at the time the solution process was terminated. It can give you further insight into the quality of solutions found by the Evolutionary Solver.

All of the reports are *Microsoft Excel worksheets*, with grid lines and row and column headings turned off. You can turn the grid lines and headings back on, if you wish, by choosing Tools Options... and selecting the View tab in the resulting dialog. In the Premium Solver products, you can request *outlined* reports, which are worksheets where certain rows are grouped together in an outline structure that you can expand or collapse as you wish. Because the reports are worksheets, you can copy and edit the report information, perform calculations on the numbers in the reports, or create graphs directly from the report data. This makes the Premium Solver Platform's reports considerably more useful than those produced by standalone optimization software packages.

---

## Selecting the Reports

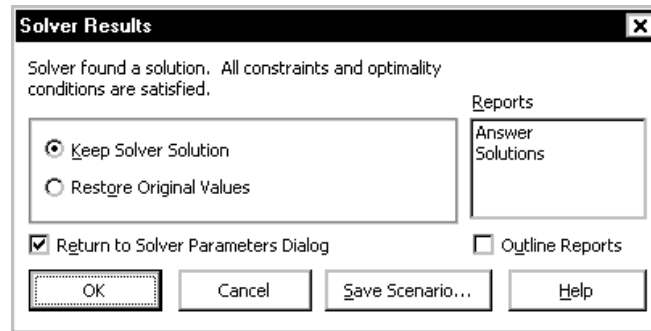
When the Solver finds the solution to an optimization problem, or when the solution process is terminated prematurely due to some error condition (or your own intervention), the Solver Results dialog is displayed, as shown below.



If the solution process was terminated prematurely, the Reports list box in the dialog above will be replaced by the legend “No reports available.” If you checked the Bypass Solver Reports box in the Solver Options dialog, the Reports list box will appear with the choices that would otherwise have been available, but they will be grayed out and you will be unable to select them.

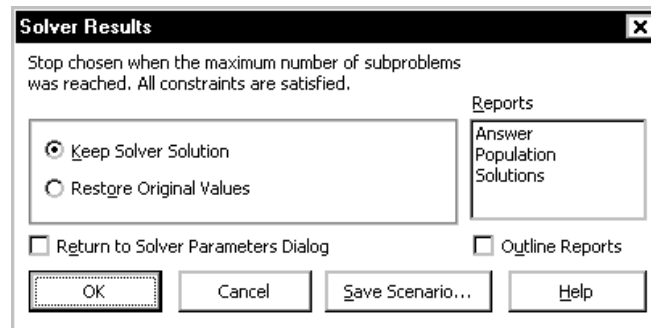
When the LP/Quadratic Solver, SOCP Barrier Solver, or GRG Nonlinear Solver finds the solution to a mixed-integer programming problem, the Reports list box includes only the Answer Report – the Sensitivity and Limits Reports are not

meaningful in this situation. If (and *only* if) the Solver finds more than one integer feasible solution or „incom b e n t“, the list box also includes the Solutions Report:



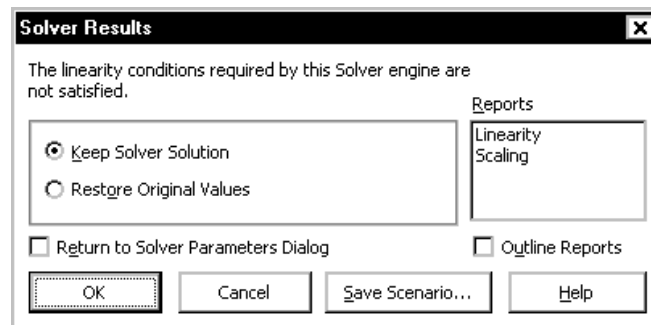
Similarly, when the GRG Nonlinear Solver or the Interval Global Solver finds the solution to a global optimization problem, the Reports list box includes only the Answer Report. If the GRG Solver, run with the „Multistart Search“ box checked, finds more than one locally optimal solution, the Reports list box includes the Solutions Report, as shown above. The Solutions Report also appears when the Interval Global Solver solves a system of nonlinear equations or a system of inequalities, without an objective function. Examples of these reports are shown in the section “The Solutions Report.”

If you are using the Evolutionary Solver, when the solution process is terminated – for any reason – you will see a Solver Results dialog like the one below:



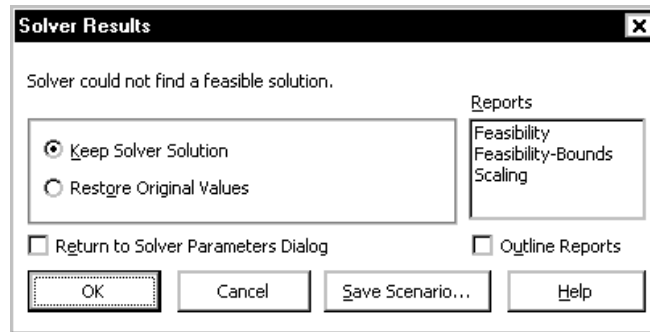
Since the Evolutionary Solver always maintains a population of candidate solutions, including a “best” solution found so far, it offers the Answer, Population and Solutions Reports in all cases – even if it has not found a feasible solution. But since the Evolutionary Solver has no strict test for optimality, linearity or even feasibility, the Linearity, Feasibility, Limits and Sensitivity Reports are not available.

If you’ve selected the Simplex LP or LP/Quadratic Solver engine, but your model contains nonlinear functions of the decision variables, the Solver will report the error via the Solver Results dialog shown below:



The only reports you can select in this situation are the Scaling Report (because a poorly scaled model can give rise to this message – see above) and the Linearity Report, which can help you locate the source of the problem with your model. An example of the Linearity Report is shown later in this chapter.

If the Solver finds that your model is infeasible, it displays a Solver Results dialog like the one shown below.



Again the Scaling Report is available, because a poorly scaled model can give rise to this message. In this case, you can select either version of the Feasibility Report (you are allowed to select both, but the “Feasibility” report contains all of the information in the “Feasibility-Bounds” version, and more). “Feasibility” performs a complete analysis of your model, including bounds on the variables, to find the smallest possible subset of these constraints that is still infeasible. This can sometimes take a great deal of computing time (if necessary, you can interrupt the analysis and production of the report by pressing the ESC key). “Feasibility-Bounds” performs a similar analysis of the constraints, but does not attempt to eliminate bounds on the variables, to save computing time.

### ***Outlining and Comments in Reports***

A useful way to control the information you see in reports is to check the box “Outline Reports” in the Solver Results dialog, to produce the reports you’ve selected in outlined format. Outlining groups the variables and constraints in the reports into “blocks,” just as you entered them in the Solver Parameters dialog; you can expand or collapse the groups to see only the information you want.

Reports in outlined format can display a descriptive comment on each “block” of variables and constraints. Comments for constraint blocks are entered in the Add Constraint and Change Constraint dialogs, displayed when you click the Add and Change buttons in the Solver Parameters dialog. To add comments to blocks of variables, click the Variables button to display the Variables list box, then click the Add or Change buttons to display the Add Variable or Change Variable dialog.

### ***Using the Solver Results Dialog***

When the Reports list box is available, you can select one or more of the reports shown. Simply click on the report names to select the reports you want, or press Alt-R and then down-arrow from the keyboard. To select more than one report, hold down the SHIFT or CTRL key while you click on the report names with the mouse.

Once the reports are selected, you can choose one of the options “Keep Solver Solution” or “Restore Original Values,” and optionally save the decision variable values in a named scenario by clicking on the Save Scenario... button. When you click on OK, the reports will be produced. Clicking on Cancel instead will cancel generation of the reports, and will discard the solution (restoring the original values).

The reports are Microsoft Excel worksheets that are inserted in the current workbook, just before the sheet containing the Solver model.

After the reports (if any) are produced, the Solver will return to worksheet Ready mode unless you have checked the box “Return to Solver Parameters Dialog.” This check box saves you the effort of selecting Tools Premium Solver... over and over, if you are solving several variations of the same problem, or solving with different option settings. When you check the “Return to Solver Parameters Dialog” box, it remains checked (until you change it) for the duration of your Excel session. To return to worksheet Ready mode, you can either click the Close button in the Solver Parameters dialog, or uncheck this box in the Solver Results dialog.

---

## The Scaling Report

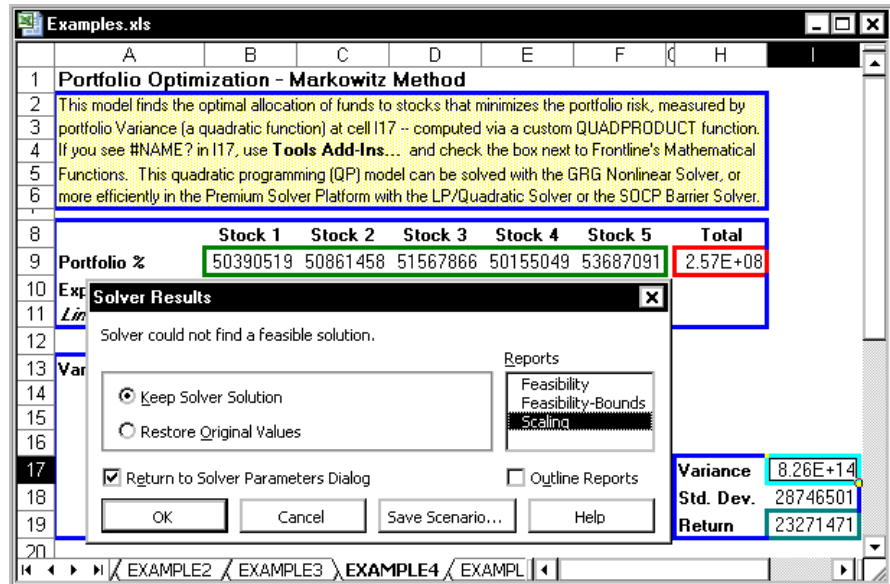
The effects of poor scaling in a large, complex optimization model can be among the most difficult problems to identify and resolve. It can cause Solver engines to return a variety of messages, with results that are suboptimal or otherwise very different from your expectations. Most Solver engines include an Automatic Scaling option to deal with scaling problems, but this can only help with the Solver's internal calculations – not with poor scaling that occurs in the middle of your Excel model.

For example, if one of your formulas adds or subtracts two quantities of very different magnitudes, such as a dollar amount in millions or billions and a return or risk measure in fractions of a percent, the result will be accurate to only a few significant digits. The effect might not be apparent given the initial values of the variables, but when the Solver explores Trial Solutions with very different values for the variables, the effect will be magnified.

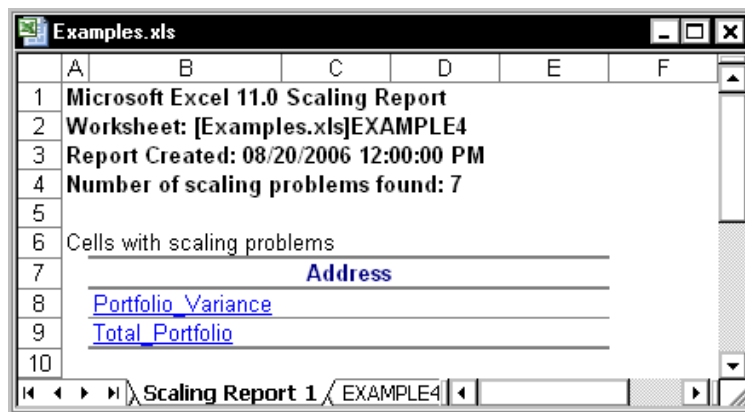
You can see an example of the effects of poor scaling if you open **Examples.xls** (in the Solver Parameters dialog, just click the Help button, then the Examples button), and select worksheet EXAMPLE4. Suppose that in this Portfolio Optimization model, you decide to make a simple change: Instead of using percentages for the stock allocations, you'd rather see the actual dollars to be invested, in your \$1 billion institutional portfolio. So you change the constraint `TotalPortfolio = 1` (or 100%) to be `TotalPortfolio = 1000000000`. You change the cell formatting to display large numbers instead of percentages, and you select the GRG Nonlinear Solver (for the sake of this example, since it is more susceptible to scaling problems than the LP/Quadratic Solver). When you click Solve, you're surprised to find that the Solver reports it cannot find a feasible solution, as shown on the next page.

The Trial Solution on the worksheet doesn't even allocate the entire \$1 billion to stocks. Since you're familiar with the Solver options, you turn on the Use Automatic Scaling box in the GRG Solver Options dialog and click Solve again, but you just get a different infeasible result. What's wrong with the Solver? (Past users of the Premium Solver products have on occasions wrestled with problems just like this.)





Noticing that the **Scaling Report** is available in the Solver Results dialog, you select this report and click OK. The Scaling Report is inserted into your workbook:



The report indicates that there are scaling problems with the formula at the cell with defined name `Portfolio_Variance` (EXAMPLE4!I17). In a very large model, this cell might be very hard to find by manual inspection. You can click on the underlined cell reference to jump to cell I17 on the EXAMPLE4 worksheet. You see that the Variance is a *very* large number – 8.26E+14, or 826 trillion. The Scaling Report has drawn your attention to a scaling issue in the formulas that calculate your model – outside of the Solver's own calculations.

In seeking the optimal solution, the Solver is likely to try extreme values – large and small – for the variables. This doesn't cause problems when the largest value is 100% or 1 and the smallest is 0, but it does cause problems when the largest value is 1 billion. At this point, calculation of the Portfolio Variance involves adding a *very small* value and a *very large* one (the Stock 5 variance times 1 billion *squared*) which leads to a loss of accuracy. This loss of accuracy leads directly to the Solver's problems finding a solution.

## An Example Model

To illustrate the other reports provided by the Premium Solver Platform, we'll start with the model on worksheet EXAMPLE1 in the workbook Examples.xls.

The screenshot shows the Excel Solver interface for 'Example 1: Product mix problem'. The model is set up on the 'EXAMPLE1' worksheet. The objective is to maximize the total profit, which is calculated in cell D18 as \$16,000. The decision variables are the number of TV sets, stereos, and speakers to build, located in cells D9, E9, and F9 respectively, with values of 100, 100, and 100. The constraints are based on the inventory of parts: Chassis (450), Picture Tube (250), Speaker Cone (800), Power Supply (450), and Electronics (600). The Solver Parameters dialog box is open, showing the objective cell and the constraints.

Part Name	Inventory	No. Used	TV set	Stereo	Speaker
Chassis	450	200	1	1	0
Picture Tube	250	100	1	0	0
Speaker Cone	800	500	2	2	1
Power Supply	450	200	1	1	0
Electronics	600	400	2	1	1

Profits:	
By Product	
TV set	\$75
Stereo	\$50
Speaker	\$35
<b>Total</b>	<b>\$16,000</b>

You can easily load this model by clicking the Help button, then the Examples button in the Solver Parameters dialog

First, we'll solve this model in its original form, using the LP/Q uadratic and G R G Solvers, and produce Answer, Sensitivity and Limits Reports. In brief, the **Answer Report** summarizes the original and final values of the decision variables and constraints, with information about which constraints are "binding" at the solution. The **Sensitivity Report** provides information about how the solution would change for small changes in the constraints or the objective function. And the **Limits Report** shows you the largest and smallest value each decision variable can assume and still satisfy the constraints, while all other variables are held fixed at their solution values.

Next, we'll change the available inventory of Chassis at cell B11 to -1. This is shown in Examples.xls on the EXAMPLE2 worksheet. When we attempt to solve, we receive the message "Solver could not find a feasible solution," and we can produce the report shown below in the section "The **Feasibility Report**."

Next, we'll deliberately introduce a *nonlinear function* into the model, by editing the formula at cell C11 to read =SUMPRODUCT(D11:F11,\$D\$9:\$F\$9)^0.9. This is shown in Examples.xls on the EXAMPLE3 worksheet. When we attempt to solve this model with the Simplex LP or LP/Q uadratic Solver, we'll receive the message "The linearity conditions required by this Solver engine are not satisfied," and we can produce the report shown below in "The **Linearity Report**."

Returning to the unmodified version of EXAMPLE1, we'll solve the model using the Evolutionary Solver, waiting until we receive the message "Solver cannot improve the current solution." This allows us to produce the report shown below in "The **Population Report**."

In the Premium Solver Platform V7.0, the **Solutions Report** has been generalized to report multiple solutions for integer programming problems, global optimization

problems, and non-smooth optimization problems, solved by any of the built-in or plug-in Solver engines. We'll illustrate this useful report with additional examples.

The Solutions Report has a special meaning for the Interval Global Solver, because it can find multiple solutions for *systems of equations* and *systems of inequalities*. To illustrate this, we'll return to EXAMPLE1 and make the Set Cell blank, removing the objective function from this model. What remains is a set of  $\leq$  constraints – a *system of inequalities* (and bounds on the variables). When we solve this model with the LP/Quadratic and GRG Solvers, we find only a single feasible solution, which is not very informative. But when we solve it with the Interval Global Solver, we get an “inner solution” – an entire region of feasible solutions. To illustrate the Solutions Report for a *system of equations*, we'll introduce a simple but historically interesting nonlinear equation mentioned in the Introduction.

---

## The Answer Report

The Answer Report, which is available whenever a solution has been found, provides basic information about the decision variables and constraints in the model. It also gives you a quick way to determine which constraints are “binding” or satisfied with equality at the solution, and which constraints have slack. In Version 7.0, the Answer Report includes the message that appeared in the Solver Results dialog, the name of the Solver engine used to solve the problem, and statistics such as the time, iterations and subproblems required to solve the problem. An example Answer Report for the worksheet model EXAMPLE1 (when there are no upper bounds on the decision variables) is shown on the next page.

First shown are the objective function (Set Cell) and decision variables (adjustable cells), with their original value and final values. Next are the constraints, with their final cell values; a formula representing the constraint; a “status” column showing whether the constraint was binding or non-binding at the solution; and the *slack* value – the difference between the final value and the lower or upper bound imposed by that constraint.

A binding constraint, which is satisfied with equality, will always have a slack of zero. (In the standard Microsoft Excel Solver, an exception to this occurs when the right hand side of a constraint is itself a function of the decision variables. In the Premium Solver products, this special case is handled differently, and the slack value for a binding constraint will *always* be zero.)

This example shows the effect of automatic outlining of the Solver reports, which you can select via the “Outline Reports” check box in the Solver Results dialog. The outline groups correspond directly to the blocks of variables and constraints you entered in the Solver Parameters dialog – one group per row in the Constraints or Variables list box. Comments entered in the Add Constraint and Add Variable dialogs for each block appear in the Answer Report; they are visible whether the outline is expanded or collapsed.

Examples.xls

1	2	A	B	C	D	E	F	G
1		Microsoft Excel 11.0 Answer Report						
2		Worksheet: [Examples.xls]EXAMPLE1						
3		Report Created: 08/20/2006 12:00:00 PM						
4		Result: Solver found a solution. All constraints and optimality conditions are satisfied.						
5		Engine: Standard LP/Quadratic						
6		Solution Time: 00 Seconds						
7		Iterations: 3						
8		Subproblems: 0						
9		Incumbent Solutions: 0						
10								
11		Target Cell (Max)						
12		Cell	Name	Original Value	Final Value			
13		\$D\$18	Total Profits:	16000	25000			
14								
15		Adjustable Cells						
16		Cell	Name	Original Value	Final Value			
17		\$D\$9:\$F\$9						
18		\$D\$9	Number to Build-> TV set	100	200			
19		\$E\$9	Number to Build-> Stereo	100	200			
20		\$F\$9	Number to Build-> Speaker	100	0			
21								
22								
23		Constraints						
24		Cell	Name	Cell Value	Formula	Status	Slack	
25		\$C\$11:\$C\$15<=\$B\$11:\$B\$15 Inventory Constraints						
26		\$C\$11	Chassis No. Used	400	\$C\$11<=\$B\$11	Not Binding	50	
27		\$C\$12	Picture Tube No. Used	200	\$C\$12<=\$B\$12	Not Binding	50	
28		\$C\$13	Speaker Cone No. Used	800	\$C\$13<=\$B\$13	Binding	0	
29		\$C\$14	Power Supply No. Used	400	\$C\$14<=\$B\$14	Not Binding	50	
30		\$C\$15	Electronics No. Used	600	\$C\$15<=\$B\$15	Binding	0	
31								
32		\$D\$9:\$F\$9>=0 Non-Negativity Constraints						
33		\$D\$9	Number to Build-> TV set	200	\$D\$9>=0	Not Binding	200	
34		\$E\$9	Number to Build-> Stereo	200	\$E\$9>=0	Not Binding	200	
35		\$F\$9	Number to Build-> Speaker	0	\$F\$9>=0	Binding	0	
36								
37								

Answer Report 1 / EXAMPLE1 / EXAMPLE2 / EXAI

When creating a report, the Solver constructs the entries in the Name column by searching for the first text cell to the left and the first text cell above each variable (changing) cell and each constraint cell. If you lay out your Solver model in tabular form, with text labels in the leftmost column and topmost row, these entries will be most useful – as in the example above. Also note that the *formatting* for the Original Value, Final Value and Cell Value is “inherited” from the formatting of the corresponding cell in the Solver model.

## The Sensitivity Report

The Sensitivity Report provides classical sensitivity analysis information for both linear and nonlinear programming problems, including dual values (in both cases) and range information (for linear problems only). The dual values for (nonbasic) variables are called Reduced Costs in the case of linear programming problems, and Reduced Gradients for nonlinear problems. The dual values for binding constraints are called Shadow Prices for linear programming problems, and Lagrange Multipliers for nonlinear problems.

Constraints which are simple *upper and lower bounds* on the variables, that you enter in the Constraints list box of the Solver Parameters dialog, are handled specially (for efficiency reasons) by both the linear and nonlinear Solver algorithms, and will *not* appear in the Constraints section of the Sensitivity report. When an upper or lower

bound on a variable is *binding* at the solution, a nonzero Reduced Cost or Reduced Gradient for that variable will appear in the “Adjustable Cells” section of the report; this is normally the same as a Lagrange Multiplier or Shadow Price for the upper or lower bound.

*Note:* The *formatting* of cells in the Sensitivity can make a significant difference in how the Reduced Gradient, Lagrange Multiplier, Reduced Cost and Shadow Prices are displayed. Bear this in mind when designing your model and when reading the report. Since the report is a *worksheet*, you can always change the cell formatting with the Format menu.

An example of a Sensitivity Report generated for EXAMPLE1 when the Solver engine is the nonlinear GRG solver (and there are no upper bounds on the variables) is shown below. Note that it includes only the solution values and the dual values: Reduced Gradients for variables and Lagrange Multipliers for constraints. If you solve a *quadratic programming* problem with the LP/Quadratic Solver, the report will also appear in this format.

Examples.xls

Microsoft Excel 11.0 Sensitivity Report				
Worksheet: [Examples.xls]EXAMPLE1				
Report Created: 08/20/2006 12:00:00 PM				
Target Cell (Max)				
Cell	Name	Final Value		
\$D\$18	Total Profits:	25000		
Adjustable Cells				
Cell	Name	Final Value	Reduced Gradient	
\$D\$9:\$F\$9				
\$D\$9	Number to Build-> TV set	200	0	
\$E\$9	Number to Build-> Stereo	200	0	
\$F\$9	Number to Build-> Speaker	7.10543E-15	-2.5	
Constraints				
Cell	Name	Final Value	Lagrange Multiplier	
\$C\$11:\$C\$15<=\$B\$11:\$B\$15				
\$C\$11	Chassis No. Used	400	0	
\$C\$12	Picture Tube No. Used	200	0	
\$C\$13	Speaker Cone No. Used	800	13	
\$C\$14	Power Supply No. Used	400	0	
\$C\$15	Electronics No. Used	600	25	

Sensitivity Report 1 / EXAMPLE1

## Interpreting Dual Values

Dual values are the most basic form of sensitivity analysis information. The dual value for a variable is nonzero only when the variable's value is equal to its upper or lower bound at the optimal solution. This is called a *nonbasic* variable, and its value was driven to the bound during the optimization process. Moving the variable's value away from the bound will *worsen* the objective function's value; conversely, “loosening” the bound will *improve* the objective. The dual value measures the increase in the objective function's value *per unit increase* in the variable's value. In the example Sensitivity Report above, the dual value for producing speakers is -2.5, meaning that if we were to build one speaker (and therefore less of something else), our total profit would decrease by \$2.50.

The dual value for a constraint is nonzero only when the constraint is equal to its bound. This is called a *binding* constraint, and its value was driven to the bound during the optimization process. Moving the constraint left hand side value away from the bound will *worsen* the objective function's value; conversely, "loosening" the bound will *improve* the objective. The dual value measures the increase in the objective function's value *per unit increase* in the constraint's bound. In the example on the previous page, increasing the number of electronics units from 600 to 601 will allow the Solver to increase total profit by \$25.

If you are not accustomed to analyzing sensitivity information for nonlinear problems, you should bear in mind that the dual values are valid only at the single point of the optimal solution – if there is any curvature involved, the dual values begin to change (along with the constraint gradients) as soon as you move away from the optimal solution. In the case of linear problems, the dual values remain constant over the range of Allowable Increases and Decreases in the variables' objective coefficients and the constraints' right hand sides, respectively.

## Interpreting Range Information

In linear programming problems, unlike nonlinear problems, the dual values are *constant* over a range of possible changes in the objective function coefficients and the constraint right hand sides. The Sensitivity Report for linear programming problems includes this range information.

A Sensitivity Report for EXAMPLE1 when the Solver engine is the standard Simplex or LP/Quadratic Solver (and there are no upper bounds on the decision variables) is shown below. In addition to the dual values (Reduced Costs for variables, Shadow Prices for constraints), this report provides information about the range over which these values will remain valid.

Microsoft Excel 11.0 Sensitivity Report							
Worksheet: [Examples.xls]EXAMPLE1							
Report Created: 08/20/2006 12:00:00 PM							
Target Cell (Max)							
Cell	Name	Final Value					
\$D\$18	Total Profits:	25000					
Adjustable Cells							
Cell	Name	Final Value	Reduced Cost	Objective Coefficient	Allowable Increase	Allowable Decrease	
\$D\$9:\$F\$9							
\$D\$9	Number to Build-> TV set	200	0	75	25.00000002	5.00000002	
\$E\$9	Number to Build-> Stereo	200	0	50	25.00000001	12.50000001	
\$F\$9	Number to Build-> Speaker	0	-2.5	35	2.5	1E+30	
Constraints							
Cell	Name	Final Value	Shadow Price	Constraint R.H. Side	Allowable Increase	Allowable Decrease	
\$C\$11:\$C\$15	<= \$B\$11:\$B\$15						
\$C\$11	Chassis No. Used	400	0	450	1E+30	50	
\$C\$12	Picture Tube No. Used	200	0	250	1E+30	50	
\$C\$13	Speaker Cone No. Used	800	13	800	100	100	
\$C\$14	Power Supply No. Used	400	0	450	1E+30	50	
\$C\$15	Electronics No. Used	600	25	600	50	200	

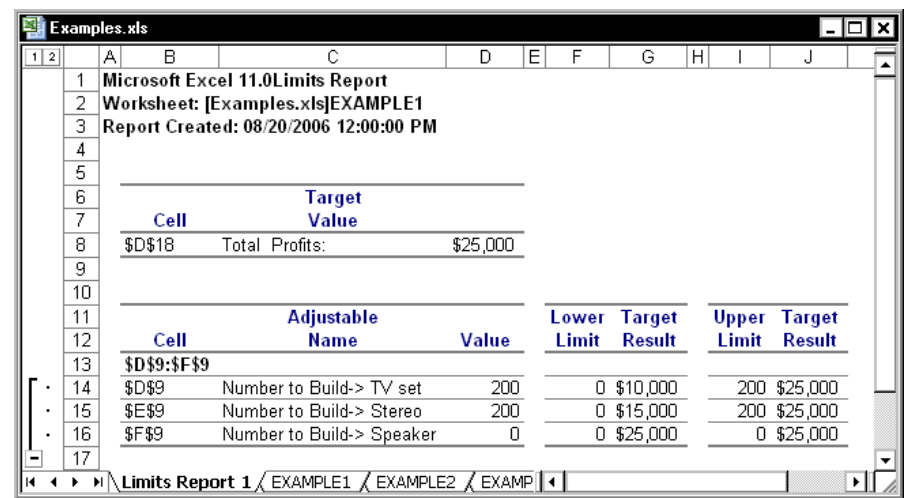
For each decision variable, the report shows its coefficient in the objective function, and the amount by which this coefficient could be increased or decreased without changing the dual value. In the example below, we'd still build 200 TV sets even if

the profitability of TV sets decreased up to \$5 per unit. Beyond that point, or if the unit profit of speakers increased by more than \$2.50, we'd start building speakers.

For each constraint, the report shows the constraint right hand side, and the amount by which the RHS could be increased or decreased without changing the dual value. In this example, we could use up to 50 more electronics units, which we'd use to build more TV sets instead of stereos, increasing our profits by \$25 per unit. Beyond 650 units, we would switch to building speakers at an incremental profit of \$20 per unit (a new dual value). A value of 1E+30 in these reports represents "infinity." In the example below, we wouldn't build any speakers regardless of how much the profit per speaker were *decreased*.

# The Limits Report

The Limits Report was designed by Microsoft to provide a specialized kind of "sensitivity analysis" information. It is created by re-running the Solver model with each decision variable (or Changing Cell) in turn as the objective (both maximizing and minimizing), and all other variables held fixed. Hence, it shows a "lower limit" for each variable, which is the smallest value that a variable can take while satisfying the constraints and holding all of the other variables constant, and an "upper limit," which is the largest value the variable can take under these circumstances. An example of the Limits Report for EXAMPLE1 is shown below.



# The Feasibility Report

The purpose of the Feasibility Report is to help you isolate the source of infeasibilities in your model. Most often, an infeasible result simply means that you've made a mistake in formulating your model, such as specifying a  $\leq$  relation when you meant to use  $\geq$ . However, if your model contains hundreds or thousands of constraints, it can be quite challenging to locate an error of this type. By isolating the infeasibility to a small subset of the constraints, the Feasibility Report can show you where to look, and hence save you a good deal of time.

To produce the Feasibility Report, the Solver may test many different variations of your model, each one with different combinations of your original constraints. This process ultimately leads to a so-called "Irreducibly Infeasible System" (IIS) of

constraints and variable bounds which, taken together, make the problem infeasible, but with the property that if any one of the constraints or bounds is removed from the IIS, the problem becomes feasible.

In a model with many constraints that “interact” with each other in complex ways, there may be many possible subsets of the constraints and bounds that constitute an IIS. Often, some of these subsets have many fewer constraints than others. The Solver attempts to find an IIS containing as few constraints as possible, trying first to eliminate “formula” constraints and then to eliminate simple variable bounds – since it is usually easier to understand the effects of variable bounds on the infeasibility of the resulting IIS.

If we attempt to solve EXAMPLE2 in the **Examples.xls** workbook – which is identical to EXAMPLE1 except that cell B11 (the right hand side of the constraint  $C11 \leq B11$ ) is set to -1 – we receive the message “Solver could not find a feasible solution.” At this point, we know only that the problem is somewhere in the set of five constraints ( $C11:C15 \leq B11:B15$ ) and three bounds on the variables. To pinpoint the problem, we select Feasibility from the Reports list box, producing a report like the one shown below. The Feasibility Report narrows the full set of constraints to the single constraint  $C11 \leq B11$  and bounds on variables D9 and E9.

Microsoft Excel 11.0 Feasibility Report						
Worksheet: [Examples.xls]EXAMPLE2						
Report Created: 08/20/2006 12:00:00 PM						
Constraints Which Make the Problem Infeasible						
Cell	Name	Cell Value	Formula	Status	Slack	
\$C\$11	Chassis No. Used	0	$\$C\$11 \leq \$B\$11$	Violated	-1	
\$D\$9	Number to Build-> TV set	0	$\$D\$9 \geq 0$	Binding	0	
\$E\$9	Number to Build-> Stereo	0	$\$E\$9 \geq 0$	Binding	0	

If your model is very large, computing the IIS may take a good deal of time. The Solver displays an estimated “% Done” on the Excel status bar as it solves variations of your model, and you can always interrupt the process by pressing ESC (in which case no report appears). Instead of the full Feasibility Report, 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 “Feasibility-Bounds” version of the report, which analyzes only the constraints while keeping the variable bounds in force. This report may be sufficient to isolate the source of the infeasibility, but you must take into account the bounds on all of the variables when reading it.

In some cases, of course, there may be no error in your model – it may correctly describe the real-world situation, and the fact that it is infeasible will probably tell you something important about the situation you are modeling. Even in such cases, the Feasibility Report can help you focus on the aspects of the real-world situation that contribute to the infeasibility, and what you can do about them.

## The Linearity Report

The purpose of the Linearity Report is to help you pinpoint nonlinear formulas in your model. The format of the Linearity Report is similar to that of the Answer



Report: It lists each decision variable and constraint on a separate row, with its cell reference, a “name” as described in “The Answer Report,” the cell’s original and final values, and a column containing “Yes” (the objective or constraint is a linear function, or the variable occurs linearly throughout the model) or “No” (the function is nonlinear, or the variable occurs nonlinearly). Since you are normally interested in the nonlinearities, any “No” entries appear in boldface.

If we attempt to solve EXAMPLE3 in the **Examples.xls** workbook – which is identical to EXAMPLE1 except that the formula at cell C11 is edited to read =SUMPRODUCT(D11:F11,\$D\$9:\$F\$9)^0.9, a nonlinear expression – we receive the message “The linearity conditions required by this Solver engine are not satisfied.” To pinpoint the problem, we select Linearity from the Reports list box, producing a report like the one shown below.

Microsoft Excel 11.0 Linearity Report					
Worksheet: [Examples.xls]EXAMPLE3					
Report Created: 08/20/2006 12:00:00 PM					
Target Cell (Max)					
Cell	Name	Original Value	Final Value	Linear Function	
\$D\$18	Total Profits:	\$16,000	\$16,000	Yes	
Adjustable Cells					
Cell	Name	Original Value	Final Value	Occurs Linearly	
\$D\$9	Number to Build-> TV set	100	0	<b>No</b>	
\$E\$9	Number to Build-> Stereo	100	0	<b>No</b>	
\$F\$9	Number to Build-> Speaker	100	0	Yes	
Constraints					
Cell	Name	Cell Value	Formula	Linear Function	
\$C\$11	Chassis No. Used	118	\$C\$11<=\$B\$11	<b>No</b>	
\$C\$12	Picture Tube No. Used	100	\$C\$12<=\$B\$12	Yes	
\$C\$13	Speaker Cone No. Used	500	\$C\$13<=\$B\$13	Yes	
\$C\$14	Power Supply No. Used	200	\$C\$14<=\$B\$14	Yes	
\$C\$15	Electronics No. Used	400	\$C\$15<=\$B\$15	Yes	

Cells D9 and E9 are shown as occurring nonlinearly in the model, and the constraint at cell C11 (the formula that was edited) is a nonlinear function of the variables. Although the formula at C11 refers to all three variable cells D9:F9, the coefficient of F9 in this formula (at cell F11) is 0 – hence cell F9 does not participate in this function, and only cells D9 and E9 are shown as occurring nonlinearly in the model.

With this information, it is easy to pinpoint the formula at C11 as the source of the nonlinearity. In real-world models, where a constraint such as C11 may depend on many other cell formulas, your next step will be to locate the specific formulas that are nonlinear, determine whether they are correct for your problem, and decide whether they can be rewritten as linear functions, or whether there is an alternative, linear formulation of your problem (see the chapter “Building Large-Scale Models” for ideas). In the Premium Solver Platform, you can use the Solver Model dialog to produce a Structure Report that will pinpoint the *exact cell formulas* throughout your model that are nonlinear.

# The Population Report

The Population Report gives you summary information about the entire population of candidate solutions maintained by the Evolutionary Solver at the end of the solution process. The Population Report can give you insight into the performance of the Evolutionary Solver as well as the characteristics of your model, and help you decide whether additional runs of the Evolutionary Solver are likely to yield even better solutions.

For each variable and constraint, the Population Report shows the best value found by the Evolutionary Solver, and the mean (average) value, standard deviation, maximum value, and minimum value of that variable or constraint across the entire population of candidate solutions at the end of the solution process. These values will give you an idea of the diversity of solutions represented by the population.

If we run the Evolutionary Solver on EXAMPLE1 with a Max Subproblems limit of 5000, and upper bounds of 200 on the variables, we find a solution of D9 = E9 = 200 and F9 = 0 (the same as the linear programming optimal solution). We can then stop the solution process and produce a Population Report like the one below.

Microsoft Excel 11.0 Population Report						
Worksheet: [Examples.xls]EXAMPLE1						
Report Created: 08/20/2006 12:00:00 PM						
Adjustable Cells						
Cell	Name	Best Value	Mean Value	Standard Deviation	Maximum Value	Minimum Value
\$D\$9	Number to Build-> TV set	200	163	69.48651079	199.9999554	23.38742391
\$E\$9	Number to Build-> Stereo	200	183	49.26542407	200	60.64074294
\$F\$9	Number to Build-> Speaker	0	1	2.455891684	7.383656412	0
Constraints						
Cell	Name	Best Value	Mean Value	Standard Deviation	Maximum Value	Minimum Value
\$C\$11	Chassis No. Used	400	346	113.2484242	399.9999554	84.02816685
\$C\$12	Picture Tube No. Used	200	163	69.48651079	199.9999554	23.38742391
\$C\$13	Speaker Cone No. Used	800	693	224.1970044	800.0001796	175.4399901
\$C\$14	Power Supply No. Used	400	346	113.2484242	399.9999554	84.02816685
\$C\$15	Electronics No. Used	600	510	179.1167681	600.0001796	114.7992472

You can see that the Best Values of the variables are far from the Mean Values across the whole population, but they are equal to the Maximum Values for cells D9 and E9, and to the Minimum Value of cell F9. Since the solution is feasible, and since the optimization process tends to drive variable values to extremes, this may indicate that we have found a globally optimal solution (which is true in this case). The Standard Deviations are relatively large, but this is not too surprising since points in the population have not yet converged to the point where we would receive "Solver has converged to the current solution."

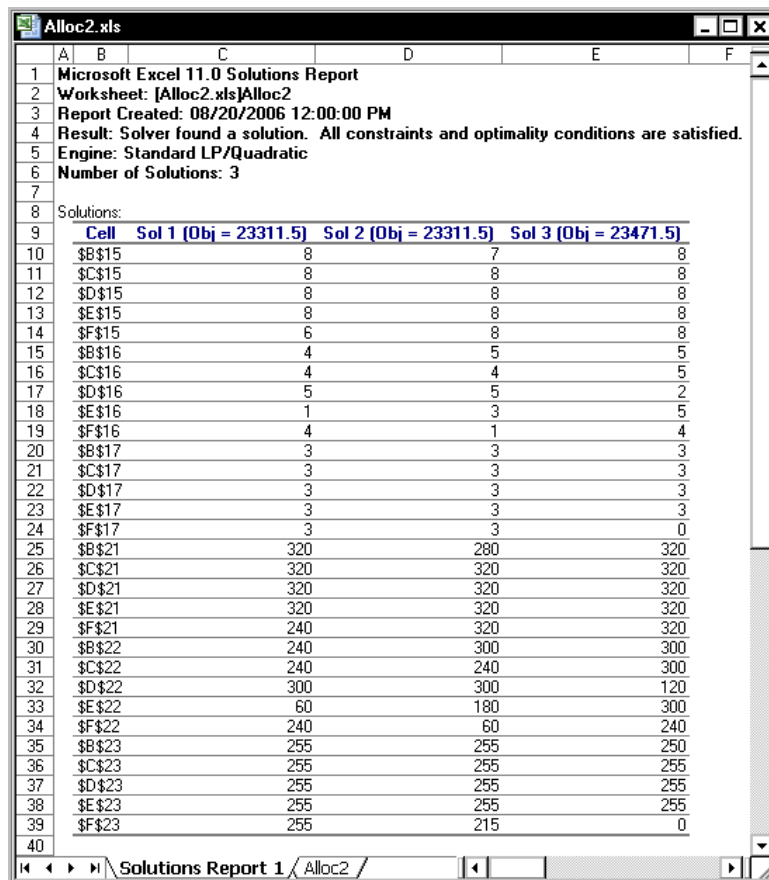
How you interpret the Population Report depends in part on your knowledge of the problem, and past experience solving it with the Evolutionary Solver or with other Solver engines. For example, if the Best Values are similar from run to run, and if the Standard Deviations are small, this may be reason for confidence that your solution is close to the global optimum. However, if the Best Values vary from run to run, small Standard Deviations might indicate a lack of diversity in the population, suggesting that you should increase the Mutation Rate and run the Solver again.

# The Solutions Report

Where the Answer Report gives you detailed information about the single “best solution” that appears on the worksheet when the Solver Results dialog is displayed, the Solutions Report gives you objective function and decision variable values for a number of alternative solutions, found during the optimization process.

## Integer Programming Problems

For mixed-integer problems, the report shows each „incumbent” or feasible integer solution found by the Branch & Bound method during the solution process. Below is an example of the Solutions Report for a problem called Alloc2.xls:



Cell	Sol 1 (Obj = 23311.5)	Sol 2 (Obj = 23311.5)	Sol 3 (Obj = 23471.5)
\$B\$15	8	7	8
\$C\$15	8	8	8
\$D\$15	8	8	8
\$E\$15	8	8	8
\$F\$15	6	8	8
\$B\$16	4	5	5
\$C\$16	4	4	5
\$D\$16	5	5	2
\$E\$16	1	3	5
\$F\$16	4	1	4
\$B\$17	3	3	3
\$C\$17	3	3	3
\$D\$17	3	3	3
\$E\$17	3	3	3
\$F\$17	3	3	0
\$B\$21	320	280	320
\$C\$21	320	320	320
\$D\$21	320	320	320
\$E\$21	320	320	320
\$F\$21	240	320	320
\$B\$22	240	300	300
\$C\$22	240	240	300
\$D\$22	300	300	120
\$E\$22	60	180	300
\$F\$22	240	60	240
\$B\$23	255	255	250
\$C\$23	255	255	255
\$D\$23	255	255	255
\$E\$23	255	255	255
\$F\$23	255	215	0

This problem was solved by the LP/Quadratic Solver with the Integer Tolerance set to 0.0 and all Cuts & Heuristics disabled. (On this problem, with Cuts & Heuristics enabled, the Solver quickly finds the true integer optimal solution as the first incumbent; the Solutions Report is available only when multiple incumbents are found.) As shown above, three incumbents were found.

## Global Optimization Problems

For global optimization problems, the report shows each locally optimal solution found by the Multistart method. On the next page is an example of the Solutions Report for a simple two-variable global optimization problem Branin.xls, solved by the GRG Nonlinear Solver with the Multistart Search option selected:

	A	B	C	D	E
1	<b>Microsoft Excel 11.0 Solutions Report</b>				
2	<b>Worksheet: [Branin.xls]Branin Model</b>				
3	<b>Report Created: 08/20/2006 12:00:00 PM</b>				
4	<b>Result: Solver converged in probability to a global solution.</b>				
5	<b>Engine: Standard GRG Nonlinear</b>				
6	<b>Number of Solutions: 2</b>				
7					
8	Solutions:				
9	<b>Cell Sol 1 (Obj = 0.397887) Sol 2 (Obj = 2.79118)</b>				
10	X		3.141592647	-2.61950252	
11	Y		2.275000022	10	
12					

In this problem, the “Branin function” must be minimized for variables  $x$  and  $y$ , subject to bounds  $-5 \leq x, y \leq 10$ . There are three distinct locally optimal solutions with objective values 2.7911 (worst), 0.5989 (better) and 0.3979 (best and globally optimal). The Solver was started at the point  $x = -2.5, y = 10$ , which is close to the worst of the three locally optimal solutions. The Multistart Search process runs the Solver from representative starting points in clusters of randomly selected points; on this run, it first found a solution close to the worst locally optimal point, then found a solution at the best and globally optimal point.

## Non-Smooth Optimization Problems

For arbitrary non-smooth optimization problems, the report shows members of the Solver’s final population of solutions. Below is an example of the Solutions Report for the global optimization problem Branin.xls, solved by the Evolutionary Solver.

	A	B	C	D
1	<b>Microsoft Excel 11.0 Solutions Report</b>			
2	<b>Worksheet: [Branin.xls]Branin Model</b>			
3	<b>Report Created: 08/20/2006 12:00:00 PM</b>			
4	<b>Result: Stop chosen when the maximum number of fe</b>			
5	<b>Engine: Standard Evolutionary</b>			
6	<b>Number of Solutions: 10</b>			
7				
8	Solutions:			
9	<b>Cell Sol 1 (Obj = 0.397888) Sol 2 (Obj = 0.397889)</b>			
10	X		3.141934696	3.14186093
11	Y		2.274814077	2.275831156
12				
13	<b>Cell Sol 9 (Obj = 0.991541) Sol 10 (Obj = 3.64982)</b>			
14	X		3.075493644	2.319032803
15	Y		1.57035295	3.431191466
16				

Again the Solver was started at the point  $x = -2.5, y = 10$ , close to the worst of the three locally optimal solutions, and it was given a limit of only 200 subproblems. Unlike the Solutions Report for gradient-based nonlinear optimizers like the GRG Nonlinear Solver, the final population of solutions is not likely to include many distinct locally optimal points. The best solutions in the Evolutionary Solver’s final population are all in the neighborhood of the globally optimal solution, which is  $x = 3.14159, y = 2.2750$ . But since the Evolutionary Solver doesn’t require gradient

information or tests for local optimality, it is unlikely to find the globally optimal solution with very high accuracy for a smooth nonlinear problem like Branin.xls.

## Solutions for Systems of Inequalities

The Solutions Report has a special meaning for the Interval Global Solver. It is available for problems with no objective function to be maximized or minimized, and with all equality constraints (a *system of equations*) or all inequality constraints (a *system of inequalities*). For a system of equations, the report is available only if the number of variables and the number of constraints are equal.

If we make the Set Cell blank in EXAMPLE1, removing the objective function from the model, what remains is a set of  $\leq$  constraints – a *system of inequalities* (and bounds on the variables). As explained in “Elements of Solver Models” in the chapter “Solver Models and Optimization,” when there is no objective, the Solver will simply find a solution that satisfies the constraints.

If we solve EXAMPLE1 with a blank Set Cell using the LP/Quadratic Solver, we get a solution where all three variables D9, E9 and F9 are 0. This is certainly a feasible solution, but it is not very informative. If we solve the model with the GRG Solver, we get a solution where all three variables are 100 (equal to the starting values of the variables). This too is a feasible solution, but it is also not very informative. Can we learn something more about the range of feasible solutions?

If we solve the model with the Interval Global Solver, we get a solution where all three variables are 75. If we select the Solutions Report from the Solver Results dialog and click OK, a report like the one below will appear.

Cell	From Value	To Value
\$D\$9	0.000000	149.999999
\$E\$9	0.000000	149.999999
\$F\$9	0.000000	149.999999

The report tells us that the problem has an infinite number of solutions, and it gives us an “inner solution” – a set of ranges or *intervals* for each decision variable, such that *all* points within these ranges satisfy the system of inequalities. An inner solution is always a “box” with a dimension for each decision variable (in general, the ranges for each variable may be different), and this box lies entirely within the feasible region. It does *not* usually enclose *all* of the feasible points – which can form an arbitrary multidimensional “shape” – and it is also *not unique* (there can be many possible inner solutions). But it will give you a much better idea of the range of feasible solutions than you can get from the Answer Report.

## Solutions for Systems of Equations

The Solutions Report for a *system of equations* can be considerably more valuable than the Answer Report or the Solutions Report for a system of inequalities. It relies on the unique ability of the Interval Global Solver to find *all real solutions* of a system of nonlinear equations.

We can illustrate the Solutions Report for a system of equations with the simplest case of a single equation – drawn from the 1983 textbook *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, by Jack E. Dennis and Robert B. Schnabel. This classic (and still popular) textbook – a key learning resource for the designers of the Microsoft Excel Solver at Frontline Systems in 1990 – describes the capabilities and limitations of methods for nonlinear optimization and solution of nonlinear equations, using an example in Section 2.1, titled “**What Is Not Possible:**”

“Consider the problem of finding the real roots of each of the following nonlinear equations in one unknown:

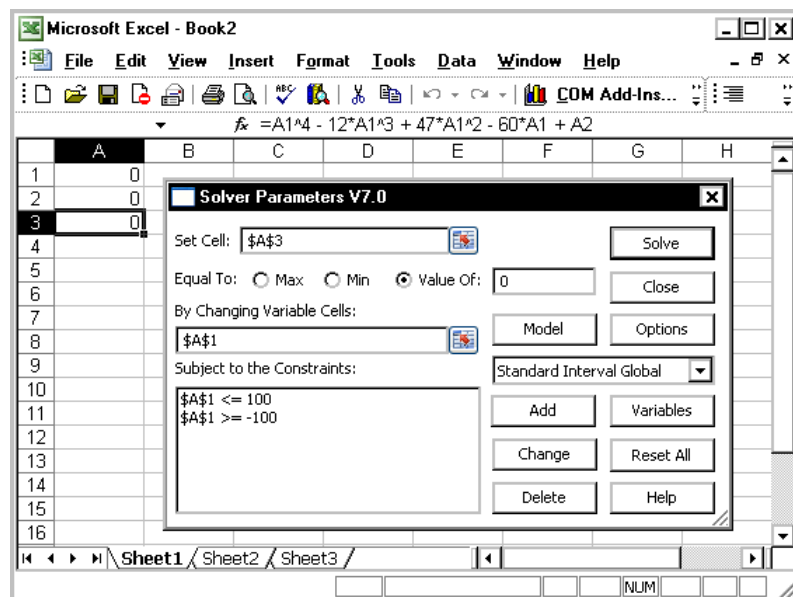
$$f_1(x) = x^4 - 12x^3 + 47x^2 - 60x,$$

$$f_2(x) = x^4 - 12x^3 + 47x^2 - 60x + 24,$$

$$f_3(x) = x^4 - 12x^3 + 47x^2 - 60x + 24.1.$$

It would be wonderful if we had a general-purpose computer routine that would tell us: “The roots of  $f_1(x)$  are  $x = 0, 3, 4$ , and  $5$ ; the real roots of  $f_2(x)$  are  $x = 1$  and  $x = 0.888$ ;  $f_3(x)$  has no real roots.” It is unlikely that there will ever be such a routine. In general, the questions of existence and uniqueness ... are beyond the capabilities one can expect of algorithms that solve nonlinear problems.”

Note that  $f_1(x)$ ,  $f_2(x)$  and  $f_3(x)$  differ only in the constant term – 0, 24 and 24.1. Let's try to solve these equations (for zero roots) in Microsoft Excel. Starting with a blank worksheet, enter 0 in cell A1 for  $x$ , 0 in cell A2 for the constant term, and in cell A3 enter the equation as `=A1^4 - 12*A1^3 + 47*A1^2 - 60*A1 + A2`. In the Solver Parameters dialog, enter A1 as the Changing Cell, and enter A3 as the Set Cell with Value Of 0, or else leave the Set Cell blank and enter `A3 = 0` in the Constraints list box. Since the Interval Global Solver requires bounds on the variables, also add constraints `A1 <= 100` and `A1 >= -100`. Using `A2 = 0` initially, we are solving  $f_1(x)$ .



If you select the Interval Global Solver, click Solve, and select the Solutions Report in the Solver Results dialog, a report like the one below will appear.

Cell	Sol 1 (Obj = 0)	Sol 2 (Obj = 0)	Sol 3 (Obj = 0)	Sol 4 (Obj = 0)
\$A\$1	5.000000001	4.000000092	3	0

These are exactly the solutions  $x = 0, 3, 4$ , and  $5$  listed for  $f_1(x)$  in the textbook. If we now set cell A2 = 24, click Solve, and select the Solutions Report in the Solver Results dialog, a report like the one below appears, with the solutions for  $f_2(x) = 0$ :

Cell	Sol 1 (Obj = 0)	Sol 2 (Obj = 0)
\$A\$1	0.888305779	1

Again, these are exactly the solutions  $x = 1$  and  $x = 0.888$  listed in the textbook. If we set A2 = 24.1 and click Solve, the Solver Results dialog appears with “Solver could not find a feasible solution.”

As this example illustrates, the *Interval Global Solver* is “a general-purpose computer routine” that will tell us: “The roots of  $f_1(x)$  are  $x = 0, 3, 4$ , and  $5$ ; the real roots of  $f_2(x)$  are  $x = 1$  and  $x = 0.888$ ;  $f_3(x)$  has no real roots.” And this capability is not limited to polynomial functions – it is effective for all continuously differentiable functions.

Dennis and Schnabel’s pessimistic prediction that “It is unlikely that there will ever be such a routine” was probably correct for classical nonlinear optimization methods that evaluate functions only over real numbers. But the ability of the Premium Solver Platform to evaluate Excel formulas over *intervals*, combined with interval methods for global optimization, has made such a routine not only possible, but easy to use.





# Using the Object-Oriented API

---

## Controlling the Solver's Operation

This chapter explains how to control the Solver using the new object-oriented API (Application Programming Interface) supported by the Premium Solver and Premium Solver Platform. This API is compatible with the object-oriented APIs offered by Frontline's Risk Solver Engine for Monte Carlo simulation, and Frontline's Solver Platform SDK, used to build custom applications of optimization and simulation using C++, C#, VB.NET, Java, MATLAB and other languages.

You can also control the Solver using “traditional” VBA functions, which are upward compatible from the VBA functions supported by the standard Excel Solver. These functions are described in the next chapter, “Using Traditional VBA Functions.”

### Why Use the Object-Oriented API?

The new object-oriented API is more powerful and much more convenient for programming the Solver than the “traditional” VBA functions.

#### With the “traditional” VBA functions:

You work with procedural functions that correspond to operations – such as **SolverOK** and **SolverSolve** – you can perform interactively in the Solver dialogs. To access the model and its variables and constraints, you must call the **SolverGet** function and process the arrays of text and numbers that it returns.

To obtain solution values, you must use the Excel object model (usually the **Range** object) to access the decision variable cells on the worksheet. You must take care to access the correct cells for specific decision variables.

To obtain dual values and ranges, you must call the **SolverFinish** function to insert a report worksheet into the workbook, then use the Excel object model to access cells in the report. You must take *extra* care to access the correct report cells containing dual values and ranges for specific variables and constraints.

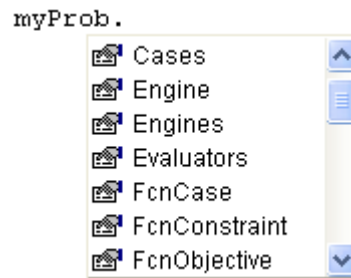
#### With the new Object-Oriented API:

You work with objects that correspond to the **Problem**, **Model**, **Solver**, **Engine**, **Variables**, and **Functions**. You can access sets of variables and constraints in the current model directly with expressions such as `myProb.VarDecision` and `myProb.FcnConstraint`.

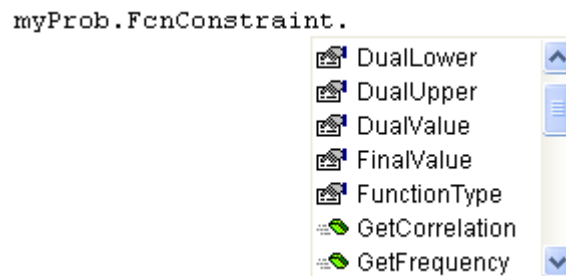
You can obtain solution values directly, with expressions such as `myProb.VarDecision.FinalValue(i)`. If you need the cell address for a set of decision variable cells, you can write an expression `myProb.VarDecision.Name`.

You can access dual values and ranges for variables and constraints directly, with expressions such as `myProb.VarDecision.DualValue(i)` or `myProb.FcnConstraint.DualValue(i)`.

The result is VBA code that's easier to read, and easier to write in the first place. Since the VBA Editor recognizes the object model exposed by the Premium Solver Platform – just as it recognizes the object model exposed by Excel – you'll receive **IntelliSense** prompts as you write code. For example, if you type a line **Dim myProb as New Problem**, then start a new line with **myProb.**, you'll be prompted with the properties and methods available for Problems:



If you select `FcnConstraint` and then type a period, you'll be prompted with the properties and methods available for Functions:



This makes it much easier to write correct code, without consulting the manual. What's more, you can use this object-oriented API when programming Excel and the Premium Solver Platform from new languages such as **VB.NET** and **C#**, working in Visual Studio, and receive IntelliSense prompts in the syntax of these languages!

Of course, you can continue to use the “traditional” VBA functions to maintain existing applications, and to write code that will run with earlier versions of the Premium Solver Platform or with the standard Excel Solver (with appropriate restrictions on functionality and problem size).

If you're using new functionality in the Premium Solver Platform, in Version 7.0 and beyond, the object-oriented API is your best bet. If you're using **Risk Solver Engine**, you'll find that it has an object-oriented API for simulation that closely resembles, and is compatible with, the Premium Solver Platform's object-oriented API for optimization. And if you're planning to move your application outside of Excel in the future – so it will run as a standalone program – you'll find that Frontline's **Solver Platform SDK** offers an object-oriented API that closely resembles the new APIs in the Premium Solver Platform and Risk Solver Engine.

## Adding a Reference in the VBA Editor

To use the new object-oriented API in VBA, you must first add a reference to the type library for the Premium Solver Platform V7.0 COM server. To do this:

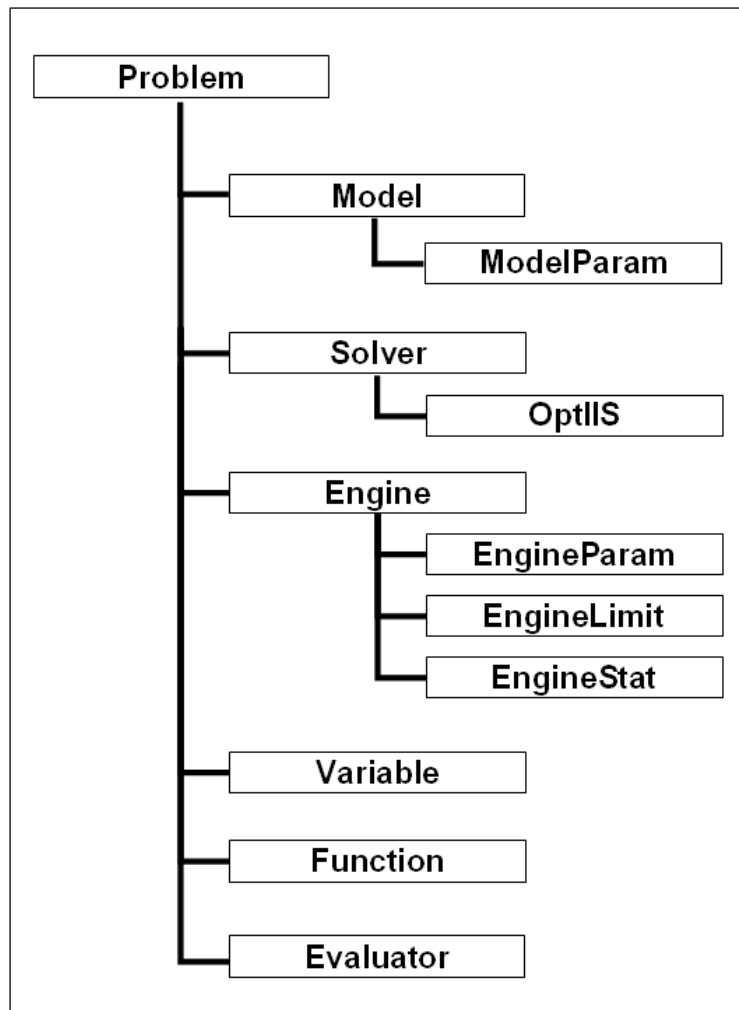
1. Press Alt-F11 to open the VBA Editor.
2. Select menu choice **Tools References**.
3. Scroll down until you find **Premium Solver Platform V7.0**.
4. Check the box next to this entry, and click OK to close the dialog.
5. Use File Save to save your workbook.

Note that this is a **different** reference from **Solver**, which is the reference you add in order to use the “traditional” VBA functions.

---

## Premium Solver Platform Object Model

The Premium Solver Platform makes available a hierarchy of objects for describing optimization problems, as pictured below.

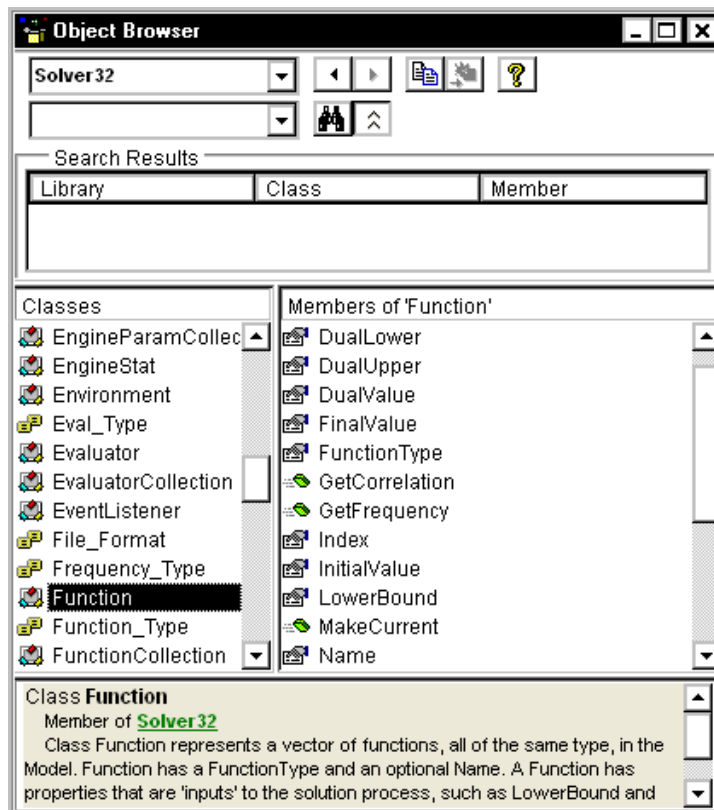


The **Problem** object represents the whole problem, and the **Model** object represents the internal structure of the model, which in the Premium Solver Platform is defined by your formulas on the spreadsheet. The **Solver** object represents the optimization process – you call its `Optimize` method to find an optimal solution. The **Engine** object represents either a built-in or plug-in Solver engine. A **Variable** object represents a range of one or more contiguous cells that contains decision variables, while a **Function** object represents a range of cells that contains either constraint left hand sides or the objective. Each Problem has a collection of Variable objects, and a collection of Function objects. An **Evaluator** represents a function you write that the Solver will call on each iteration (Trial Solution), or on each subproblem in a larger problem.

The Model has a collection of **ModelParam** objects, each representing a single option or parameter of the PSI Interpreter (appearing in the Solver Model dialog). An Engine has a collection of **EngineParam** objects, each representing a single option or parameter of a Solver engine (appearing in its Solver Options dialog). It also has an **EngineLimit** object, holding problem size limits for this Solver engine, and an **EngineStat** object, holding performance statistics for the last optimization problem solved by this engine. An **OptIIS** object holds results of an infeasibility analysis of the problem.

## Using the VBA Object Browser

You can examine the Premium Solver Platform objects, properties and methods in the VBA Object Browser. To do this, press **Alt-F11** to open the VBA Editor, and select menu choice **View Object Browser**. This displays a child window like the one pictured below.



The dropdown list at the top left corner of the Object Browser initially displays <All Libraries> – change this to select Solver32. In the object browser pictured, we've highlighted the properties of the Function object

---

## Programming the Object Model

You use the Premium Solver Platform object-oriented API by first creating an instance of a **Problem**, and initializing it with the Solver model defined on a worksheet in an open workbook. When you do this, a collection of **Variable** objects and a collection of **Function** objects are created automatically. Each Variable object corresponds to a cell range of decision variables that appears in the By Changing Cells edit box or Variables list, and each Function object corresponds to a cell range of constraints that appears in the Constraints list.

Once you have an initialized Problem object, you can do several things:

- Set Solver and Engine parameters such as the maximum time or number of iterations, the method used to compute gradients, and other options and tolerances.

- Perform an optimization, and check the final status of the solution process.

- Get results of the optimization, by accessing properties of the Variable and Function objects, and performance statistics, by accessing properties of the EngineStat object.

### Example VBA Code Using the Object Model

Below is an example of VBA code that could be linked to a command button on the worksheet:

```
Private Sub CommandButton1_Click()  
    Dim prob As New Problem  
  
    prob.Engine = prob.Engines("Standard LP/Quadratic")  
    prob.Engine.Params("MaxTime") = 600  
  
    prob.Solver.Optimize  
    MsgBox "Status = " & prob.Solver.OptimizeStatus  
    MsgBox "Obj = " & prob.FcnObjective.FinalValue(0)  
  
    For i = 0 To prob.Variables.Count - 1  
        For j = 0 To prob.Variables(i).Size - 1  
            MsgBox prob.Variables(i).FinalValue(j)  
        Next j  
    Next i  
  
    Set prob = Nothing  
End Sub
```

The first line creates an instance of a Problem, which by default is associated with the Solver model defined on the active worksheet. You could associate the Problem object with a different worksheet by calling the `prob.Init` method.

The second line selects the Standard LP/Quadratic Solver engine, and the third line sets the maximum solution time to 600 seconds. The **string names** of parameters such as "MaxTime" are documented in the chapter "Solver Options," and are usually the same as the names of the corresponding parameters passed via the "traditional" VBA functions, such as SolverOptions, SolverModel and SolverGRGOptions.

The next set of three lines performs the optimization, displays the Solver Result status code (for example 0), and displays the final value of the objective.

The double for-loop in the next five lines steps through the Variable objects – each one representing a range of contiguous cells – and displays the final value for each variable cell in each range.

## Evaluators Called During the Solution Process

You can write a VBA function that the Solver will call at certain points during the solution process. In this “callback function,” you can access information about the problem and solution so far, to monitor or report progress and decide whether to stop or continue the solution process.

The object-oriented API defines an Evaluator object that is associated with your “callback function” and specifies when the Premium Solver Platform should call it. You can define Evaluators to be called on each iteration or Trial Solution, or on each subproblem or each new solution (“incumbent”) when the solution process involves multiple subproblems (global optimization problems, and problems with integer variables).

The VBA function you write to serve as an Evaluator must be contained in a **class module** – not a regular VBA module – and it must be declared to have the **WithEvents** property.

Here is an example of code for an Evaluator, in a class module named Class1:

```
Private WithEvents EvalIterator As Solver32.Evaluator

Private Function EvalIterator_Evaluate _
    (ByVal Evaluator As Solver32.IEvaluator) As _
    Solver32.Engine_Action

    MsgBox "Iteration = " _
        & Evaluator.Problem.Engine.Stat.Iterations _
        & Chr(13) & Chr(10) & "Objective = " _
        & Evaluator.Problem.FcnObjective.Value(0) _
        & Chr(13) & Chr(10)

    EvalIterator_Evaluate = Engine_Action_Continue
End Function

Public Sub MySolve()
    Set EvalIterator = New Evaluator
    Dim prob As New Problem
    prob.Evaluators(Eval_Type_Iteration) = EvalIterator
    prob.Solver.Optimize
    Set EvalIterator = Nothing
End Sub
```

Having created the class module Class1, in a regular VBA module you can create an instance of Class1, and then call the MySolve method in Class1:

```
Private Sub CommandButton1_Click()
    Dim c As New Class1
    c.MySolve
End Sub
```

## Refinery.xls: Multiple Blocks of Variables and Functions

A further example of programming the object-oriented API is shown in the model **Refinery.xls**, which is installed in the Examples folder, normally at the path C:\Program Files\Frontline Systems\Premium Solver Platform\Examples.

The Refinery.xls model, which is based on Problem 12.6 in the 3rd edition of *Model Building in Mathematical Programming* by H.P. Williams (see the Recommended Books on [www.solver.com](http://www.solver.com) for details), has ten blocks of decision variables and eleven blocks of constraints, plus bounds on certain variables.

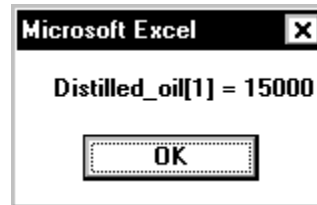
The VBA code for this model illustrates some of the many ways you can use the object-oriented API. For example, the following line displays the time that was required to solve the problem:

```
MsgBox prob.Engine.Stat.Milliseconds & " msec"
```

The following code illustrates one way to display final solution values for each of the ten blocks of decision variables in this problem:

```
For i = 0 To prob.Variables.Count - 1
    prob.Variables(i).MakeCurrent
    For j = 0 To prob.VarDecision.Size - 1
        MsgBox prob.VarDecision.Name _
            & "[" & j+1 & "] = " & _
            prob.VarDecision.FinalValue(j)
    Next j
Next i
```

When executed, this code displays MessageBoxes such as:



The line `prob.Variables(i).MakeCurrent` associates the Problem property `VarDecision` (a single block of decision variables) with each of the ten blocks of variables in turn, allowing you to refer to solution values and dual values of this block in more compact notation. A similar line of code can be used to make the Problem property `FcnConstraint` represent one of the eleven blocks of constraints. The `.Name` property of the block, which usually returns a string such as "\$A \$1:\$A \$10", returns "Distilled\_oil" in this case, since the Excel model has a defined name for this block of cells.

## CuttingStock.xls: Multiple Problems and Dynamically Generated Variables

A more ambitious example of programming the object-oriented API is shown in the model **CuttingStock.xls**, which is installed in the Examples folder, normally at the path C:\Program Files\Frontline Systems\Premium Solver Platform\Examples.

This application uses the object-oriented API to define and repeatedly solve two optimization problems, passing information back and forth between the two problems. One problem is instantiated from a worksheet with the `prob.Init` method as

mentioned earlier; the other problem is created “from scratch,” with new dynamically generated decision variables added each time the problem is solved.

## Cutting Stock Problem

CuttingStock.xls solves a classical “cutting stock” problem, which arises for example in lumber and paper mills. Imagine that you have a number of sheets of wood or rolls of paper of a fixed width, waiting to be cut; you have customer orders for sheets or rolls of various different widths. Your task is to cut the larger, fixed-width sheets or rolls into different sizes in a way that minimizes the total stock used while meeting customer demand.

You might for example cut a 100 inch sheet into two sheets of 45 inches (leaving 10 inches wasted), three sheets of 31 inches (leaving 7 inches wasted), one 45-inch sheet and one 31-inch sheet (leaving 24 inches wasted), etc. Each of these is called a *pattern*, and the main problem will have a decision variable representing the number of copies of that pattern to cut. In a „real-life” application, the number of possible patterns is exponentially large, yielding a model that is too large to solve.

## Column Generation Method

We can instead use the technique of *column generation* (columns here refers to variables in the main problem). We choose a small initial set of patterns to include in the model, and solve the main problem (an LP). Since it is unlikely that we chose the perfect set of patterns initially, we use the dual variable information from the main problem to generate a new pattern. We generate this new pattern by solving a second optimization problem, called a „knapsack” problem. A decision variable for the new pattern is dynamically created and added to the main problem, which is solved again. These two problems, the main problem and the knapsack problem, are solved in turn until no more patterns can be generated that will reduce the total stock used.

## Worksheets and VBA Code

In CuttingStock.xls, sheet Input contains the „knapsack” problem, which is solved to generate new patterns, and sheet Patterns contains the main problem. Our VBA code executes a loop, alternately solving the main problem and the knapsack problem. When the solution to the main problem can no longer be improved, we solve a final problem where we add integer constraints on the variables, so the final solution yields an exact count of the patterns that should be cut.

Open CuttingStock.xls and press Alt-F11 to view its VBA code in the VBA Editor. The first block of code clears the Patterns sheet and sets up the initial patterns. This code simply sets values and formulas into cells, using the Excel object model.

```
Dim nPat As Integer, i As Integer, _  
    nNumDemands As Integer  
nPat = Range("Demands").Count  
nNumDemands = nPat  
Worksheets("Patterns").Activate  
Range("$A$1:$Z$100").Clear  
  
' create initial patterns  
For i = 1 To nPat  
    Cells(2 + i, i) = Int(Range("RollSize").Value2 _  
        / Range("Widths").Cells(i).Value2)  
    Cells(2 + i, nPat + 1).Formula = "=sumproduct(" &  
        & Range(Cells(1, 1), Cells(1, nPat)).Address  
        & ", " & Range(Cells(2 + i, 1), _  
            Cells(2 + i, nPat)).Address & ")"
```



```

Cells(3 + nPat, i) = Range("RollSize").Value2 - _
Cells(2 + i, i) * Range("Widths").Cells(i).Value2
Next i

```

The code then enters the main loop in which we add a new pattern to the main problem, and setup and solve that problem using the Premium Solver Platform's object model:

```

'generate patterns
Cells(2, 1).Formula = "=sum(" _
    & Range(Cells(1, 1), Cells(1, nPat)).Address & ")"
For i = 1 To nNumDemands
    Cells(2 + i, nPat + 1).Formula = "=sumproduct(" _
        & Range(Cells(1, 1), Cells(1, nPat)).Address _
        & "," & Range(Cells(2 + i, 1), Cells(2 + i, _
            nPat)).Address & ")"
Next i

Dim prob As New Problem

' variables
prob.Variables.Clear
Dim vars As New Variable
vars.Init Range(Cells(1, 1), Cells(1, nPat))
vars.NonNegative
prob.Variables.Add vars
Set vars = Nothing

' objective
prob.Functions.Clear
Dim objective As New Solver32.Function
objective.Init Range(Cells(2, 1), Cells(2, 1))
objective.FunctionType = Function_Type_Objective
prob.Functions.Add objective
Set objective = Nothing

' constraints
ReDim constraints(1 To nNumDemands) As _
    New Solver32.Function
For i = 1 To nNumDemands
    constraints(i).Init Range(Cells(2 + i, nPat + 1), _
        Cells(2 + i, nPat + 1))
    constraints(i).LowerBound(0) = _
        Range("Demands").Cells(i).Value2
    prob.Functions.Add constraints(i)
Next i

prob.Solver.SolverType = Solver_Type_Minimize
prob.Engine = prob.Engines("Standard LP/Quadratic")
prob.Solver.Optimize

```

Next, the code obtains the dual values from the solution to the main problem, via the Premium Solver Platform object model, and stores these values as parameters of the knapsack problem on the Input worksheet, via the Excel object model:

```

' capture shadow prices
Worksheets("Input").Activate
Dim j As Integer
j = 1
For i = 1 To prob.Functions.Count
    If prob.Functions(i - 1).FunctionType = _
        Function_Type_Constraint Then
        Cells(2 + j, 5) = _
            prob.Functions(i - 1).DualValue(0)
    End If
    j = j + 1
Next i

```

```

        j = j + 1
    End If
Next i

```

The code then sets up and solves the knapsack problem, again using the Premium Solver Platform object model:

```

Dim prob1 As New Problem
prob1.Init Worksheets("Input")
prob1.Engine = prob1.Engines("Standard LP/Quadratic")
prob1.Engine.Params("IntTolerance").Value = 0
prob1.Solver.Optimize

```

If the objective value of the knapsack problem is less than 1 (allowing for rounding error) – meaning that there are no more patterns that will improve the solution – we can exit the loop.

```

If 1 - prob1.FcnObjective.FinalValue(0) _
    >= -0.00001 Then
    Exit Do
End If

```

Otherwise, the code writes the new pattern to the Patterns (main problem) worksheet, using the Excel object model:

```

Worksheets("Patterns").Activate
Cells(nNumDemands + 3, nPat + 1) = _
    Range("RollSize").Value2
' write out new pattern, and associated waste
For i = 1 To nNumDemands
    Cells(2 + i, nPat + 1) = _
        prob1.VarDecision.FinalValue(i - 1)
    Cells(nNumDemands + 3, nPat + 1) = _
        Cells(nNumDemands + 3, nPat + 1).Value2 - _
        prob1.VarDecision.FinalValue(i - 1) * _
        Range("widths").Cells(i).Value
Next i

nPat = nPat + 1
Set prob1 = Nothing

```

When the Do ... Loop is exited, all patterns have been generated. Finally, the code solves one more problem with integer constraints on the variables, to ensure that we produce the exact count needed to meet customer demand:

```

Worksheets("Patterns").Activate
prob.Init Worksheets("Patterns")
prob.Functions.Clear
prob.Variables.Clear

' variables
Dim finalvars As New Variable
finalvars.Init Range(Cells(1, 1), Cells(1, nPat))
For i = 1 To nPat
    finalvars.IntegerType(i - 1) = Integer_Type_Integer
Next i
finalvars.NonNegative
prob.Variables.Add finalvars

' objective
Cells(3 + nNumDemands, nPat + 1).Formula = _
    "=sumproduct(" & Range(Cells(1, 1), _
        Cells(1, nPat)).Address & "," _
    & Range(Cells(3 + nNumDemands, 1), _
        Cells(3 + nNumDemands, nPat)).Address & ")"

```

```

Dim TotalWaste As New Solver32.Function
TotalWaste.Init Range(Cells(3 + nNumDemands, _
    nPat + 1), Cells(3 + nNumDemands, nPat + 1))
TotalWaste.FunctionType = Function_Type_Objective
prob.Functions.Add TotalWaste

' constraints
ReDim constraints(1 To nNumDemands) As _
    New Solver32.Function
For i = 1 To nNumDemands
    Cells(2 + i, nPat + 1).Formula = "=sumproduct(" _
        & Range(Cells(1, 1), Cells(1, nPat)).Address _
        & "," & Range(Cells(2 + i, 1), Cells(2 + i, _
            nPat)).Address & ")"
    constraints(i).Init Range(Cells(2 + i, nPat + 1), _
        Cells(2 + i, nPat + 1))
    constraints(i).LowerBound(0) = _
        Range("Demands").Cells(i).Value2
    prob.Functions.Add constraints(i)
Next i

prob.Engine = prob.Engines("Standard LP/Quadratic")
prob.Engine.Params("IntTolerance").Value = 0
prob.Solver.Optimize
Set prob = Nothing

```

This example illustrates some of the power of the object-oriented API. Although you could use the “traditional” VBA functions, described in the next chapter, to obtain similar results, it would require quite a bit more code to do so, especially at the step of obtaining the dual values from the solution of the main problem and using them to solve the next knapsack problem.

If you wanted to move this application from Excel to a standalone program, you'd find that nearly all the code in CuttingStock.xls that references the Premium Solver Platform object model could be re-used, with few or no changes, in building an application for the Solver Platform SDK. You'd have to rewrite the code that references cells via the Excel object model to use arrays in a programming language instead, but this would not be difficult.

---

## Object-Oriented API Structure

This section summarizes the objects available in the Premium Solver Platform's object-oriented API. The API is designed for compatibility with the objects, properties and methods of the Solver Platform SDK, Frontline's “flagship” product for software developers building applications in a programming language.

### Primary Objects

The primary objects in the API represent elements of your optimization problem: the entire **Problem**, the **Model** (implemented by Excel formulas on the worksheet), a **Solver** for optimization and several **Engines** that can perform optimization; a set of **Variable** objects, each one representing a contiguous range of decision variable cells on the worksheet, and a set of **Function** objects, each one representing either the objective (a single cell) or a contiguous range of constraint cells on the worksheet.

Class Name	Description
Problem	Represents an entire Problem.
Solver	Represents either simulation or optimization.
Engine	Represents a Solver Engine that can perform either optimization or (in the Solver Platform SDK or Risk Solver Engine) simulation.
Evaluator	Represents user-written code to be called by an Engine when various events occur during the solution of a Problem.
Model	Defines how the user's model can be evaluated.
Variable	Represents a vector of variables, all of the same type.
Function	Represents a vector of functions, all of the same type.

A Problem object has members that represent *collections* of Solvers, Engines, Evaluators, Functions, and Variables. You can subscript the name of a collection to access a specific object – for example one Engine or one Variable object – and you can write for-loops that step through all of the objects in a collection. For example:

```
For i = 0 To myProb.Variables.Count - 1
    MsgBox myProb.Variables(i).Name
Next i
```

In VBA, you can also iterate through a collection using a “foreach” loop:

```
Dim myVar As Variable
For Each myVar In myProb.Variables
    MsgBox myVar.Name
Next
```

Since a Variable object represents a contiguous range of decision variable cells on the worksheet, its properties (for example FinalValue) represent arrays of numbers. Again you can subscript these properties in your VBA code. For example:

```
For i = 0 To prob.Variables.Count - 1
    For j = 0 To prob.Variables(i).Size - 1
        MsgBox prob.Variables(i).FinalValue(j)
    Next j
Next i
```

## Secondary Objects

The secondary objects in the API allow you to work with sets of numbers that are associated with the Model, an Engine, a Variable or Function, or optimization results. The **ModelParam** and **EngineParam** objects each represent one parameter for the modeling system (PSI Interpreter) or a Solver Engine, respectively. The **OptIIS** and **Statistics** objects group related properties for convenient access.

The **DoubleMatrix** and **DependMatrix** objects are quite powerful – they can be created in your code with a Dim statement to hold a large, sparse matrix of numbers or dependency information, respectively. A matrix object could be dimensioned as (say) 1 million rows by 1 million columns, but would reserve memory only for its

nonzero elements; you can use this object much like a two-dimensional array in assignments statements in your code.

Class Name	Description
ModelParam	Represents a single Model (PSI Interpreter) parameter.
EngineParam	Represents a single Engine parameter.
EngineLimit	Represents the problem size limits for an Engine.
EngineStat	Represents statistics from the last run of an Engine.
OptIIS	Represents an Irreducibly Infeasible Subset of the constraints and variable bounds in an optimization problem.
Statistics	Represents statistics for variable and function values across a population of final solutions.
DoubleMatrix	Represents a matrix of double values, of dimension $m$ (rows) by $n$ (columns).
DependMatrix	Represents a matrix of integers of values drawn from the <code>Depend_Type</code> enum.

## Primary Objects

This section describes the primary objects available in the Premium Solver Platform's object-oriented API, and their properties and methods. As noted above, these objects represent the main elements of your optimization problem.

### Problem Object

The Problem object is created to represent an optimization problem, with a VBA statement such as:

```
Dim prob As New Problem
```

### Problem Methods

You can use the **Init** method to instantiate the Problem from a named model or worksheet – this will create all of the Variable and Function objects for the problem defined in that model or worksheet. (If you don't use **Init**, the Problem is instantiated from the active worksheet.) You can use the **Load** method to load a set of problem specifications, or the **Save** method to save problem specifications on the worksheet.

```
Problem.Init Worksheet
```

```
Problem.Load RangeOrModel, Format
```

```
Problem.Save Range, Format
```

*Worksheet* is an Excel Worksheet object. *Range* is an Excel range object, and *RangeOrModel* may be either an Excel Range object or a text string model name. *Format* is one of the symbolic constants XLStd, XLPSI, or XLWB.

## Accessing Multiple Solutions

For global optimization and mixed-integer programming problems, most Solver engines find multiple solutions. The best of these solutions is returned via the `FinalValue` property of the `Variable` and `Function` objects associated with the problem. But you can access any of the solutions from your VBA code. To do this, set the `Problem` object `SolutionIndex` property to an integer from 0 to the value of the `NumSolutions` property – 1; then access the `Value` property of the `Variable` and `Function` objects associated with the problem.

## Problem Properties

Property Name	Property Type	Description
Name	string	Name of worksheet defining problem.
ProblemType	Problem_Type	Problem type (linear, nonlinear, etc.)
NumSolutions	integer	Number of solutions available.
SolutionIndex	integer	Index of the currently selected solution (0 to NumSolutions – 1).
ObjectiveIndex	integer	Index of the current objective function; currently always 0.
Solver	Solver	Current Solver – always for optimization.
Engine	Engine	Current Solver engine.
Model	Model	Current Model.
VarDecision	Variable	Current decision variable block.
FcnConstraint	Function	Current constraint function block.
FcnObjective	Function	Current objective function block.
Collection Name	Object Type	Description
Solvers	Solver	One item – Solver for optimization.
Engines	Engine	Solver engines – initially five items.
Evaluators	Evaluator	User-defined Evaluators – initially none.
Variables	Variable	Ranges of contiguous variable cells.
Functions	Function	Ranges of contiguous constraint/obj cells.

## Problem\_Type Constants

The **Problem\_Type** enum has a set of symbolic constant values that reflect the type of optimization model:

```
Problem_Type_NA
Problem_Type_OptLP
Problem_Type_OptQP
Problem_Type_OptQCP
Problem_Type_OptSOCP
Problem_Type_OptNLP
Problem_Type_OptNSP
```

The value of the `ProblemType` property will be `Problem_Type_NA` unless the `Model` object `DependCheck` method is called. After this method is called, the `ProblemType` property will reflect the diagnosis of the model: LP (linear program), QP (quadratic program), QCP (quadratically constrained QP), SOCP (second order code program), NLP (smooth nonlinear program), or NSP (non-smooth program).

## Solver Object

The Solver object represents the optimization process – you call its method `Optimize` to solve the problem, and access its property `OptimizeStatus` to check the final status (e.g. optimal, infeasible, unbounded) of the optimization.

There is only one instance of the Solver object in a problem; this is created automatically when you create a `Problem`, and is accessible via a reference such as `myProb.Solver`. In contrast, there are several `Engine` objects, one for each built-in or plug-in Solver engine installed for the Premium Solver Platform.

## Solver Methods

The **Optimize** method runs the currently selected Solver engine to solve the problem for the current `Model`, `Variables` and `Functions`. The **IISFind** method may be called if the `OptimizeStatus` property indicates that no feasible solution was found – it finds an Irreducibly Infeasible Subset (IIS) of the constraints (see “The Feasibility Report” in the chapter “Solver Reports” for more information about this analysis). The `Report` method produces a Solver report, in the form of an Excel worksheet inserted into the active workbook.

```
Solver.Optimize
Solver.IISFind
Solver.Report ReportName
```

*ReportName* is one of the following text strings: "Scaling", "Answer", "Sensitivity", "Limits", "Feasibility", "Linearity", "Population", "Solutions".

## Solver Properties

Property Name	Property Type	Description
SolverType	Solver_Type	Type of solution to find: Solver_Type_Maximize, Solver_Type_Minimize, or Solver_Type_FindFeas.
Problem	Problem	Problem associated with this Solver.
Index	integer	Index in the Solver Collection; always 0.
NumSolutions	integer	Number of different solutions available.
SolutionIndex	integer	Index of the current solution; used to access multiple solutions.
OptimizeStatus	Optimize_Status	Status after optimization. See status codes/messages in the chapter “Diagnosing Solver Results.”
OptIIS	OptIIS	Retrieves the Irreducibly Infeasible Subset found by IISFind.

## Engine Object

An Engine object represents a single Solver engine, such as the LP/Quadratic Solver or the GRG Nonlinear Solver.

A collection of Engine objects, one for each installed (built-in or plug-in) Solver engine, is created automatically when you create a Problem, and is accessible via a reference such as myProb.Engines. You can subscript the **Engines** collection, either by an integer index or by a text string name, to access a specific Engine.

The Problem's **Engine** property refers to the currently selected Solver engine, which will be run when the Solver.Optimize method is called. To select a different Solver engine, you can either assign a new reference to the Engine property, for example myProb.Engine = myProb.Engines("Standard LP/Quadratic"), or you can call the chosen Solver Engine object's **MakeCurrent** method.

### Engine Methods

The **MakeCurrent** method causes this Solver engine to become the currently selected Engine; after calling this method, myProb.Engine will refer to this Solver engine, and myProb.Solver.Optimize will run this Solver engine. The ParamReset method resets all of the Solver engine parameters (EngineParam objects) in this Engine's Params collection to their default values.



Engine.MakeCurrent

Engine.ParamReset

### ***Engine Properties***

<b>Property Name</b>	<b>Property Type</b>	<b>Description</b>
Problem	Problem	Problem associated with this Engine.
ProblemType	Problem_Type	Type of problem solved by this Engine.
Name	string	Name of this Solver engine.
FileSpec	string	Filename/path of this Engine's dynamic link library. For built-in Engines, returns an empty string.
Params	EngineParam Collection	The collection of all parameters for this Solver engine.
Limit	EngineLimit array	Engine size limits; indexed by Problem Type.
Index	integer	Index in the EngineCollection of this Engine object.
Stat	EngineStat	Engine statistics from the last Optimize method call using this Solver engine.
<b>Constant Name</b>	<b>Description</b>	
LPQPName	Name of the built-in LP/QP Solver engine.	
LPName	Name of the built-in LP Simplex Solver engine.	
SOCPPName	Name of the built-in SOCP Barrier Solver engine.	
GRGName	Name of the built-in GRG Nonlinear Solver engine.	
EVOName	Name of the built-in Evolutionary Solver engine.	
INTName	Name of the built-in Interval Global Solver engine.	

## **Evaluator Object**

An Evaluator object represents a “callback function” that you have written in VBA, that the Solver will call at certain times during the optimization process. No Evaluators are required, but you may find it useful to create one or more Evaluators to monitor or report progress, especially if the optimization takes a long time.

See the earlier section “Evaluators Called During the Solution Process” for an example of VBA code that defines and uses an Evaluator.

## Evaluator Methods

The **Evaluate** method is your “callback function” – the method that the Solver will call during the optimization process, at the times specified by the EvalType property.

```
Evaluator.Evaluate
```

## Evaluator Properties

Property Name	Property Type	Description
EvalType	Eval_Type	Specifies when to call this Evaluator: Eval_Type_Iteration, Eval_Type_Subproblem, or Eval_Type_NewSolution.
Problem	Problem	Problem associated with this Evaluator.
RefUser	Variant	Any user data that you wish to make available to the Evaluator when it is called.

## Model Object

The Model object represents your model, as defined by formulas on an Excel worksheet. There is one instance of the Model object in a problem; it is created automatically when you create a Problem, and is accessible via a reference such as myProb.Model.

## Model Methods

The **DependCheck** method performs a dependency analysis of your model – much like pressing the Check Model button in the Solver Model dialog. After this method is called, access the Problem object **ProblemType** property and the Model object **AllGradDepend** property for dependency information about the current model.

```
Model.DependCheck Transformed, CheckFor
```

*Transformed* is either False (check dependencies in the Original model), or True (check dependencies in the Transformed model); see “Transforming a Non-Smooth Model” in the chapter “Analyzing and Solving Models” for more information.

*CheckFor* is 1 to check for the ability to compute **Gradients**, 2 to check the model **Structure**, or 3 to check the models (Structure and) **Convexity**.

## Model Properties

Property Name	Property Type	Description
Problem	Problem	Problem associated with this Model.
Params	ModelParam Collection	The collection of all parameters for the PSI Interpreter used when it analyzes the model.

NumVariables	integer array	Index by Variable_Type_Decision to get the total number of decision variables.
NumFunctions	integer array	Index by Function_Type_Constraint to get the total number of constraint functions.
AllGradDepend	DependMatrix	The dependency matrix – rows represent constraints, columns represent variables, values are from the Depend_Type enum.

## Depend\_Type Constants

The **Depend\_Type** enum has a set of symbolic constant values that reflect the nature of the dependence between a specific function and decision variable in the **AllGradDepend** matrix:

```
Depend_Type_None
Depend_Type_Linear
Depend_Type_Quadratic
Depend_Type_Smooth
Depend_Type_NonSmooth
```

If a function does not depend at all on a specific variable, the matrix entry for that function (row) and variable (column) will be **Depend\_Type\_None**.

## Variable Object

In the Premium Solver Platform, a Variable object represents a block of decision variables. In Risk Solver Engine, a Variable object represents a block of *uncertain* variables, and in the Solver Platform SDK, both decision variables and uncertain variables can be used. Hence, in the object-oriented API each Variable object has a **VariableType** property, with symbolic values drawn from the Variable\_Type enum (always Variable\_Type\_Decision in the Premium Solver Platform).

When you create a Problem, a collection of Variable objects – one for each block of contiguous decision variable cells on the Excel worksheet – is created automatically, and is accessible via a reference such as `myProb.Variables`. You can subscript the **Variables** collection, either by an integer index or by a text string such as “\$A\$1:\$A\$10”, to access a specific Variable object.

The Variable objects that are created automatically for a new Problem, based on the Excel worksheet, have properties that are „read only”: You can get, but you cannot set the Name, Value, LowerBound, UpperBound, IntegerType, etc.

You can create a new Variable object (with a line of code such as **Dim myVar as New Variable**), call **myVar.Init** to associate it with a cell range on the worksheet, set the other properties of this Variable, then add this new Variable to the Variables collection of the Problem (with code such as **myProb.Variables.Add myVar**). This is illustrated in the section “Cutting Stock.xls: Multiple Problems and Dynamically Generated Variables.” After you do this, the cell range for the new Variable will be available in the Solver Parameters dialog Variable list box, and it will be saved as part of the model when the workbook is saved. Immediately after you add the object to the Variables collection, you will find that its properties are now „read only”.

The Problem’s **VarDecision** property initially refers to the first Variable object in the collection (i.e. the first block of variables in the Variables list box in the Solver Parameters dialog). To select a different Variable, you can either assign a new

reference to the VarDecision property, for example `myProb.VarDecision = myProb.Variables("$A$1:$A$10")`, or you can call the chosen Variable objects **MakeCurrent** method.

### Variable Methods

The **Init** method causes this Variable object to be associated with a cell range on the worksheet. The **MakeCurrent** method makes this Variable object the „currently selected block of variables; after calling this method, `myProb.VarDecision` will refer to this Variable object. The **NonNegative** method provides a quick way to specify that all variables in the block are nonnegative (LowerBound of 0).

`Variable.Init Range`

`Variable.MakeCurrent`

`Variable.NonNegative`

*Range* is an Excel range object.

### Variable Properties

Property Name	Property Type	Description
Problem	Problem	Problem associated with this Variable.
VariableType	Variable_Type	Type of the Variable – in the PSP, always Variable_Type_Decision.
Name	string	Name of the Variable – in the PSP, either a cell range such as “\$A\$1:\$A\$10” or a defined name appearing on the worksheet.
Size	integer	Number of elements in this Variable (vector of decision variables).
Index	integer	Index of this Variable object in the Variables Collection.
Position	integer	Starting position of this vector of variables in the Model flat address space.
Value	double	Current values of the vector of variables.
Statistics	Statistics	Statistics for this vector of variables; set only when the Evolutionary Solver or the OptQuest Solver is selected and the Optimize method is called.
LowerBound	double	Lower bounds on the vector of variables. Currently, lower bounds must be the <i>same</i> for all variables in one Variable object.
UpperBound	double	Upper bounds on the vector of variables. Currently, upper bounds must be the <i>same</i> for all variables in one Variable object.

IntegerType	Integer_Type	Indicates whether this vector of variables is continuous, integer, or binary integer.
GroupIndex	integer	Index of the alldifferent group to which the variables belong; 0 if these variables are not part of any alldifferent group.
ConeType	Cone_Type	Indicates whether this vector of variables belongs to a cone, and the type of cone.
ConeIndex	integer	Index of the cone to which the variables belong; 0 if these variables do not belong to any cone.
InitialValue	double	Initial values of the vector of variables.
FinalValue	double	Final values of the vector of variables (after an optimization).
DualValue	double	Dual values of the vector of variables (after an optimization).
DualLower	double	Lower bounds of the ranges of the objective coefficients for which the dual values are valid (after an optimization).
DualUpper	double	Upper bounds of the ranges of the objective coefficients for which the dual values are valid (after an optimization).

## Function Object

In the Premium Solver Platform, a Function object represents either a block of constraints, or the objective function. In Risk Solver Engine, a Function object represents a block of *uncertain* functions, and in the Solver Platform SDK, each of these function types can be used. Hence, in the object-oriented API each Function object has a **FunctionType** property, with symbolic values drawn from the **Function\_Type** enum (either **Function\_Type\_Constraint** or **Function\_Type\_Objective** in the Premium Solver Platform).

When you create a Problem, a collection of Function objects – one for the objective (Set Cell), and one for each block of contiguous constraint cells on the Excel worksheet – is created automatically, and is accessible via a reference such as `myProb.Functions`. You can subscript the **Functions** collection, either by an integer index or by a text string such as “\$A \$1:\$A \$10”, to access a specific Function object.

The Function objects that are created automatically for a new Problem, based on the Excel worksheet, have properties that are „read only”: You can get, but you cannot set the Name, Value, LowerBound, UpperBound, etc.

You can create a new Function object (with a line of code such as **Dim myFunc as New Solver32.Function**), call **myFunc.Init** to associate it with a cell range on the worksheet, set the other properties of this Function, then add this new Function to the Functions collection of the Problem (with code such as **myProb.Functions.Add myFunc**). After you do this, the cell range for the new Function will be available in

the Solver Parameters dialog Constraint list box, and it will be saved as part of the model when the workbook is saved. Immediately after you add the object to the Functions collection, you will find that its properties are now „read only“.

The Problem's **FcnObjective** property refers to the objective (Set Cell) in the Solver Parameters dialog, and its **FcnConstraint** property initially refers to the first constraint Function object in the collection (i.e. the first block of constraints in the Constraints list box in the Solver Parameters dialog). To select a different Function, you can either assign a new reference to the FcnConstraint property, for example `myProb.FcnConstraint = myProb.Functions("$A$1:$A$10")`, or you can call the chosen Function object's **MakeCurrent** method.

## Function Methods

The **Init** method causes this Function object to be associated with a cell range on the worksheet. The **MakeCurrent** method makes this Function object the „currently selected“ block of functions; after calling this method, `myProb.FcnConstraint` will refer to this Function object. The **NonNegative** method provides a quick way to specify that all constraints in the block have a LowerBound of 0. The **Relation** method lets you specify a relation –  $\leq$ ,  $=$ , or  $\geq$  – and a right hand side value to be applied to all elements of a block of constraints.

```
Function.Init Range
Function.MakeCurrent
Function.NonNegative
Function.Relation Rel, RHS
```

*Range* is an Excel range object. *Rel* is one of the symbolic constants `Cons_Rel_EQ` ( $=$ ), `Cons_Rel_GE` ( $\geq$ ) or `Cons_Rel_LE` ( $\leq$ ). Currently *RHS* must be a single numeric value, or an array in which all the numeric values are the *same*.

## Function Properties

Property Name	Property Type	Description
Problem	Problem	Problem associated with this Function.
FunctionType	Function_Type	Type of the Function – in the PSP, either <code>Function_Type_Objective</code> or <code>Function_Type_Constraint</code> .
Name	string	Name of the Function – in the PSP, either a cell range such as “\$A\$1:\$A\$10” or a defined name appearing on the worksheet.
Size	integer	Number of elements in this Function (vector of constraints); always 1 for the objective function.
Index	integer	Index of this Function object in the Functions Collection.

Position	integer	Starting position of this vector of functions in the Model flat address space.
Value	double	Current values of the vector of constraints, or of the objective.
Statistics	Statistics	Statistics for this vector of functions; set only when the Evolutionary Solver or the OptQuest Solver is selected and the Optimize method is called.
LowerBound	double	Lower bounds on the vector of functions. Currently, you can set either the LowerBound or the UpperBound, but <i>not both</i> , and lower bounds must be the <i>same</i> for all of the functions in one Function object.
UpperBound	double	Upper bounds on the vector of functions. Currently, you can set either the LowerBound or the UpperBound, but <i>not both</i> , and upper bounds must be the <i>same</i> for all of the functions in one Function object.
InitialValue	double	Initial values of the vector of functions.
FinalValue	double	Final values of the vector of functions (after an optimization).
DualValue	double	Dual values of the vector of functions (after an optimization).
DualLower	double	Lower bounds of the ranges of the right-hand sides for which the dual values are valid (after an optimization).
DualUpper	double	Upper bounds of the ranges of the right-hand sides for which the dual values are valid (after an optimization).

---

## Secondary Objects

This section describes the secondary objects available in the Premium Solver Platform's object-oriented API, and their properties and methods. As noted above, these objects allow you to work with sets of numbers that are associated with the Model, an Engine, a Variable or Function, or optimization results.

## ModelParam Object

A ModelParam object represents a single parameter or option controlling the PSI Interpreter when it analyzes your model, when you call the Model.DependCheck method or the Solver.Optimize method. The Model object for a Problem has a property Params which is a collection of ModelParam objects. As with other collections, you can subscript the Params collection with an integer index or a string name, for example myProb.Model.Params("ReqSmooth") = 1.

### ModelParam Properties

Property Name	Property Type	Description
Name	string	Name of the parameter.
Value	double	Current value of the parameter.
Default	double	Default value of the parameter.
MinValue	double	Minimum value of the parameter.
MaxValue	double	Maximum value of the parameter.

### ModelParam Names

The ModelParam names for current parameters of the PSI Interpreter match the symbolic names of parameters of the “traditional” VBA function SolverModel, and are summarized below:

"SolveWith"	1 = Use PSI Interpreter, 2 = Use Excel Interpreter
"SolveTransformed"	1 = Solve Transformed, 0 = Solve Original model
"CheckFor"	1 = Gradients, 2 = Structure, 3 = Convexity, 4 = Automatic
"ShowTransformations"	1 = Create Transformation Report, 0 = Don't create
"ShowExceptions"	1 = Create Structure Report, 0 = Don't create
"DesiredModel"	1 = Linear, 2 = Quadratic, 3 = Conic, 4 = Smooth Nonlinear, 5 = Non-smooth
"Engines"	1 = All, 2 = Valid, 3 = Good, 4 = Best
"ReqSmooth"	1 = Treat ABS, IF, MAX, MIN, SIGN as non-smooth, 0 = Treat as smooth
"FastSetup"	1 = Use old-style Fast Problem Setup, 0 = Don't
"Sparse"	1 = PSI Interpreter runs in Sparse mode, 0 = Dense
"ActiveOnly"	1 = PSI Interpreter analyzes active worksheet only, 0 = analyzes referenced cells throughout workbook

The “SolveWith” and “SolveTransformed” parameters are effective when your code calls the Solver.Optimize method. The “ShowTransformations”, “ShowExceptions”, “DesiredModel”, and “Engines” parameters are effective when your code calls the Model.DependCheck method. Other parameters affect any use of the PSI Interpreter.



## EngineParam Object

An EngineParam object represents a single parameter or option controlling the behavior of a Solver engine. Each Engine object has a property Params which is a collection of EngineParam objects. As with other collections, you can subscript the Params collection with an integer index or a string name. For example, you can write **myProb.Model.Params("MaxTime") = 600** to set the maximum solution time to 10 minutes (600 seconds).

### EngineParam Properties

Property Name	Property Type	Description
Name	string	Name of the parameter.
Value	double	Current value of the parameter.
Default	double	Default value of the parameter.
MinValue	double	Minimum value of the parameter.
MaxValue	double	Maximum value of the parameter.

To find EngineParam names for the parameters of different Solver engines, consult the “Solver Options” chapters in this Guide (for built-in Engines) and the Solver Engines Guide (for plug-in Engines). The parameter names normally match the symbolic names of parameters of the “traditional” Solver Option VBA functions.

## EngineLimit Object

An EngineLimit object holds limits on the maximum size or complexity of problems handled by a Solver engine. These limits are „read only” – you can access them in your VBA code, which can be useful if your code is written to work with a variety of Solver engines. Each Engine object contains an EngineLimit object; the currently selected Solver engine’s limits may be referenced as **myProb.Engine.EngineLimit**.

### EngineLimit Properties

Property Name	Property Type	Description
IterationLimit	integer	Maximum number of iterations.
VarDecisionLimit	integer	Maximum number of variables.
FcnConstraintLimit	integer	Maximum number of constraints, apart from variable bounds.
VarBoundLimit	integer	Maximum number of explicit bounds on variables.
VarIntegerLimit	integer	Maximum number of integer (including alldifferent) variables.

## EngineStat Object

An EngineStat object holds performance statistics from the last time an Engine was used to solve a problem, by calling the Solver.Optimize method. These statistics are „read only“, but you can read, analyze and report them in your VBA code.

### *EngineStat Properties*

Property Name	Property Type	Description
Milliseconds	integer	Time taken to solve the problem, in thousands of a second.
Iterations	integer	Number of iterations performed.
Subproblems	integer	Number of subproblems explored, in a global optimization problem or an integer programming problem.
LocalSolutions	integer	Number of locally optimal solutions found in a global optimization problem, or number of improved incumbents found in an integer programming problem.
FunctionEvals	integer	Number of function evaluations performed.
JacobianEvals	integer	Number of Jacobian (1st derivative) evaluations performed, if used by the Solver engine.
HessianEvals	integer	Number of Hessian (2nd derivative) evaluations performed, if used by the Solver engine.

## OptIIS Object

An OptIIS object holds information about an Irreducibly Infeasible Subset (IIS) of the constraints, which is found when you call the Solver.FindIIS method. You can use myProb.Solver.OptIIS to access the OptIIS information for a problem.

The FindIIS method may be called for a problem when the result of calling the Optimize method is an „infeasible OptimizeStatus“. An IIS is a subset of the constraints such that a problem consisting of just these constraints is still infeasible, but if any one of the constraints in the IIS is dropped, the problem becomes feasible. Examining the IIS may help you determine why a problem is infeasible.

## OptIIS Properties

Property Name	Property Type	Description
NumBounds	integer	Number of variable bounds in the IIS.
NumConstraints	integer	Number of constraints in the IIS.
BoundIndex	integer array	Array of indices of variables (in the Models flat address space) whose bounds are included in the IIS.
BoundStatus	IIS_Status array	Array of status values for variable bounds, corresponding to indices in the BoundIndex array.
ConstraintIndex	integer array	Array of indices of constraints (in the Models flat address space) that are included in the IIS.
ConstraintStatus	IIS_Status array	Array of status values for constraints, corresponding to indices in the ConstraintIndex array.

The **IIS\_Status** values can be any of the symbolic names **IIS\_Status\_LowerBound**, **IIS\_Status\_UpperBound**, or **IIS\_Status\_Fixed**.

## Statistics Object

A Statistics object holds statistics for variable and function values across a population of final solutions. Each Variable object and Function object contains one Statistics object, holding values for that vector of variables or functions. The Statistics properties are set after an optimization, but only when the Evolutionary Solver or the OptQuest Solver is used.

## Statistics Properties

Property Name	Property Type	Description
NumValues	integer	Number of values in Statistics arrays.
NumErrors	integer	0 in the Premium Solver Platform.
Minimum	double array	Minimum value across the population.
Maximum	double array	Maximum value across the population.
Mean	double array	Mean value across the population.
StdDev	double array	Standard deviation across the population.

The Statistics object has additional properties, such as Mode, Variance, Skewness and Kurtosis, that are used only for uncertain variables and functions, when solving simulation problems with Risk Solver Engine or the Solver Platform SDK.

## DoubleMatrix Object

A DoubleMatrix object is a flexible and powerful tool for working with large, sparse matrices of numbers. It automatically allocates and manages memory only for the nonzero elements of the matrix. 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

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.

Methods **NZBgn** and **NZEnd** should be called, to save time, if you wish to assign a large number of non-zeros to the matrix at one time, without accessing or using these values until all of them have been assigned. First call **NZBgn**, then perform the assignments, then call **NZEnd**. 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.Create Rows, Columns
DoubleMatrix.InitDense Rows, Columns, Elements
DoubleMatrix.InitSparse Rows, Columns, Elements
DoubleMatrix.Clear
DoubleMatrix.Destroy
DoubleMatrix.NZBgn
DoubleMatrix.NZEnd
DoubleMatrix.NZCancel
```

*Rows* is the number of rows, and *Columns* is the number of columns in the matrix. *Elements* is a one-dimensional VBA array whose elements are assigned to the matrix.

### DoubleMatrix Properties

Property Name	Property Type	Description
IsEmpty	Boolean	Flag indicating the matrix is empty.
IsSparse	Boolean	Flag indicating sparse or dense storage of the matrix.

ArrayOrder	Array_Order	Specifies whether rows or columns are the major (first) dimension. The default array order is <code>Array_Order_ByCol</code> ; you may instead specify <code>Array_Order_ByRow</code> . Changing the <code>ArrayOrder</code> property transposes the matrix.
MajorDim	integer	Retrieves the major (first) dimension (rows or columns depending on the <code>ArrayOrder</code> property setting).
MinorDim	integer	Retrieves the minor (second) dimension (rows or columns depending on the <code>ArrayOrder</code> property setting).
Rows	integer	Retrieves the number of rows in the matrix. Does not depend on the <code>Array_Order</code> .
Columns	integer	Retrieves the number of columns in the matrix. Does not depend on the <code>Array_Order</code> .
NumElements	integer	In a sparse matrix, the actual number of non-zero elements stored. In a dense matrix, this equals (Rows*Columns).
Value	double	The value of an element of the matrix – may appear on either side of an assignment statement.

A `DoubleMatrix` object has several other properties, not documented here, that are primarily useful in cases where the data that will be used to initialize the matrix is already organized in sparse column-major or row-major form. For more information, consult the VBA Object Browser or the Solver Platform SDK documentation, or contact Frontline Systems at [info@solver.com](mailto:info@solver.com).

## DependMatrix Object

A `DependMatrix` object is used to hold information about dependencies of problem functions (objective and constraints) on decision variables. In the matrix, rows represent constraints or the objective, and columns represent variables.

The value of the `Model` object **AllGradDepend** property is a `DependMatrix`. This matrix is initialized (only) when you call the `Model` object **DependCheck** method. An element of the matrix at position (i, j) has a value from the `Depend_Type` enum, that tells you whether constraint i depends in a linear, quadratic, smooth nonlinear, or nonsmooth way on variable j, or does not depend on variable j at all.

### Depend\_Type Constants

The `Depend_Type` enum has a set of symbolic constant values that reflect the nature of the dependence between a specific function and decision variable:

```
Depend_Type_None
Depend_Type_Linear
```

Depend\_Type\_Quadratic  
Depend\_Type\_Smooth  
Depend\_Type\_NonSmooth

If a function does not depend at all on a specific variable, the matrix entry for that function (row) and variable (column) will be Depend\_Type\_None.

### ***DependMatrix Methods***

A DependMatrix is created for you as the value of the Model object AllGradDepend property; you can simply access its elements by subscripting, for example by writing myProb.Model.AllGradDepend( i, j). Should you need to create such a matrix, however, you can use the same methods documented above for a DoubleMatrix: **Create**, **InitDense**, **InitSparse**, **Clear**, **Destroy**, **NZBgn**, **NZEnd** and **NZCancel**.

### ***DependMatrix Properties***

A DependMatrix has the same properties as a DoubleMatrix, as documented above: **IsEmpty**, **IsSparse**, **ArrayOrder**, **MajorDim**, **MinorDim**, **Rows**, **Columns**, **NumElements** and **Value**. Only the **Value** property is different: Where this property is of type double for a DoubleMatrix, it is of type Depend\_Type for a DependMatrix.

A DependMatrix object has several other properties, not documented here, that are primarily useful in cases where the data that will be used to initialize the matrix is already organized in sparse column-major or row-major form. For more information, consult the VBA Object Browser or the Solver Platform SDK documentation, or contact Frontline Systems at [info@solver.com](mailto:info@solver.com).

# Using Traditional VBA Functions

---

## Controlling the Solver's Operation

This chapter explains how to control the Solver using “traditional” VBA functions, which are upward compatible from the VBA functions supported by the standard Excel Solver. The Premium Solver and Premium Solver Platform also support a new object-oriented API that is high-level, easy to use, and compatible with the object-oriented API offered by Frontline’s Risk Solver Engine for Monte Carlo simulation, and the Solver Platform SDK, used to build custom applications of optimization and simulation using C++, C#, VB.NET, Java, MATLAB and other languages.

Using traditional VBA functions, you can display or completely hide the Solver dialog boxes, create or modify the choices of objective (Set Cell), variables (Changing Cells) and constraints, check whether an optimal solution was found, and produce reports. You do this by calling a set of Solver-specific functions from a macro program you write in Visual Basic Applications Edition (VBA). If you need to work with solution values or report information in your VBA code, create and solve multiple optimization problems, or port your code to run as a standalone application, you may find that the object-oriented API is a better choice.

### Running Predefined Solver Models

Controlling the Solver can be as simple as *adding one line* to your macro program code! Each worksheet in a workbook may have a Solver problem defined, which is saved automatically with the workbook. You can create this Solver model interactively if you wish. If you distribute such a workbook, with a worksheet containing a Solver model and a VBA module, you can simply add a reference to the Solver add-in, activate the worksheet, and add one line to call the function **SolverSolve** in VBA.

### Using the Macro Recorder

If you want to setup a Solver model “from scratch” programmatically, one easy way to see how to use the traditional VBA functions is to turn on the Macro Recorder (select **Tools** | **Macro** | **Record New Macro...**) and then set up a Solver model interactively. Microsoft Excel will record a macro in VBA that calls the Solver functions to mimic the actions you perform. You can then edit and customize this macro, and incorporate it into your application.

## Using Microsoft Excel Help

You can learn about the standard Solver functions in Excel's online Help. In Excel 2007, Excel 2003 and Excel XP, open the Visual Basic Editor (Alt-F11), select Help - Microsoft Visual Basic Help, click on the Index tab, and type „Solver“ to display an index list of function names. In Excel 2000, open Help and choose the Contents tab. Open the section “Programming Information,” and within this section open “Functions.” This will display the Solver function names.

## Referencing Functions in Visual Basic

To use the VBA functions, your Visual Basic module must include a reference to the **Solver** add-in (Solver.xla). In Microsoft Excel, first select Tools Premium Solver... to ensure that the Premium Solver or Premium Solver Platform add-in is open. Then (after closing the Solver dialog) press Alt-F11 to open the Visual Basic Editor, choose Tools References... and make sure that the box next to **Solver** is checked. Note that this is different from a reference to the **Premium Solver Platform V7.0** type library, which you add when you use the new object-oriented API in VBA.

## Checking Function Return Values

The Solver functions generally return integer values, *which you should check in your VBA code*. The normal return value is 0, indicating that the function succeeded. Other possible return values are given in the descriptions of the individual functions. If the arguments you supply are invalid, an error condition can be raised, which you would have to handle via an On Error VBA statement.

Of particular interest is the return value of the **SolverSolve** function, which describes the result of the actual optimization step. The return value can range from -1 to 21 in the Premium Solver products, with additional values starting at 1000 for the Premium Solver Platform's Interval Global Solver and field-installable Solver engines. These integer values are summarized in the description of the **SolverSolve** function below, but for a comprehensive discussion, see the chapter “Diagnosing Solver Results,” starting with the subsection “Standard Solver Result Messages.”

One group of functions can return a variety of numeric, logical, string or array values, depending on the arguments you supply. These functions (SolverGet, SolverOkGet, etc.) may be used to “read” the settings of the current Solver model, on the active sheet or any other worksheet whose name you supply.

## Standard, Model and Premium Macro Functions

The following sections describe each of the VBA function calls supported by the Premium Solver products. These functions are a compatible superset of the function calls available in the standard Excel Solver.

The functions are listed alphabetically in three groups. The first group consists of functions available in both the standard Excel Solver and the Premium Solver products. The second group (Premium VBA Functions) consists of functions that are available only in the Premium Solver products. The third group (Solver Model VBA Functions) consists of functions that are available only in the Premium Solver Platform. If you want to write VBA code that will work with both the standard Solver and the Premium Solver products, you should limit yourself to functions in the first group, and consult the notes on each function call to determine which arguments are supported by the standard Solver.



---

# Standard VBA Functions

The VBA functions in this section are available in both the standard Excel Solver and the Premium Solver products. Some of these functions have extra arguments that are supported only in the Premium Solver products, as noted in each function description.

---

## SolverAdd (Form 1)

Equivalent to choosing Solver... from the Tools menu and pressing the Add button in the Solver Parameters dialog box. Adds a constraint to the current problem.

*VBA Syntax*

**SolverAdd (CellRef:=, Relation:=, FormulaText:=, Comment:=, Report:=)**

**CellRef** is a reference to a cell or a range of cells on the active worksheet and forms the left hand side of the constraint.

**Relation** specifies the arithmetic relationship between the left and right hand sides, or whether **CellRef** must have an integer value at the solution.

Relation	Relationship
<b>1</b>	<=
<b>2</b>	=
<b>3</b>	>=
<b>4</b>	<b>int</b> (CellRef is an integer variable)
<b>5</b>	<b>bin</b> (CellRef is a binary integer variable)
<b>6</b>	<b>dif</b> (CellRef is an alldifferent group)
<b>7</b>	<b>soc</b> (CellRef belongs to a second order cone)
<b>8</b>	<b>src</b> (CellRef belongs to a rotated second order cone)

**FormulaText** is the right hand side of the constraint and will often be a single number, but it may be a formula (as text) or a reference to a range of cells.

**Comment** is a string corresponding to the Comment field in the Add Constraint dialog, only in the Premium Solver products.

**Report** is no longer used, but is included for compatibility with previous versions of the Premium Solver products.

The standard Excel Solver supports only **Relation** values 1 to 5. If **Relation** is 4 to 8, **FormulaText** is ignored, and **CellRef** must be a subset of the Changing Cells.

If **FormulaText** is a reference to a range of cells, the number of cells in the range must match the number of cells in **CellRef**, although the shape of the areas need not be the same. For example, **CellRef** could be a row and **FormulaText** could refer to a column, as long as the number of cells is the same.

### Remarks

The SolverAdd, SolverChange and SolverDelete functions correspond to the Add, Change, and Delete buttons in the Solver Parameters dialog box. You use these functions to define constraints. For many macro applications, however, you may find it more convenient to load the problem in a single step using the SolverLoad call.

Each constraint is uniquely identified by the combination of the cell reference on the left and the relationship (<=, =, >=, int, bin, dif, soc or src) between its left and right sides. This takes the place of selecting the constraint in the Solver Parameters dialog box. You can manipulate constraints with SolverChange and SolverDelete.

---

## SolverAdd (Form 2)

Equivalent to choosing Solver... from the Tools menu, pressing the Variables button, and then pressing the Add button in the Solver Parameters dialog box. Adds a set of decision variable cells to the current problem. This form is supported only by the Premium Solver products.

*VBA Syntax*

**SolverAdd (CellRef:=, Comment:=, Report:=)**

**CellRef** is a reference to a cell or a range of cells on the active worksheet and forms a set of decision variables.

**Comment** is a string corresponding to the Comment field in the Add Variable Cells dialog, only in the Premium Solver products.

**Report** is no longer used, but is included for compatibility with previous versions of the Premium Solver products.

### Remarks

The SolverAdd, SolverChange and SolverDelete functions correspond to the Add, Change, and Delete buttons in the Solver Parameters dialog box. In this form, you can use these functions to add or change sets of decision variables. For many macro applications, however, you may find it more convenient to load the problem in a single step using the SolverLoad function.

Note that SolverOk defines the *first* entry in the By Changing Variable Cells list box. Use SolverAdd to define additional entries in the Variables Cells list box. Do *not* call SolverOk with a different ByChange:= argument *after* you have defined more than one set of variable cells.

---

## SolverChange (Form 1)

Equivalent to choosing Solver... from the Tools menu and pressing the Change button in the Solver Parameters dialog box. Changes the right hand side of an existing constraint.

*VBA Syntax*

**SolverChange (CellRef:=, Relation:=, FormulaText:=, Comment:=, Report:=)**

For an explanation of the arguments and selection of constraints, see **SolverAdd**.

### Remarks

If the combination of **CellRef** and **Relation** does not match any existing constraint, the function returns the value 4 and no action is taken.

To change the **CellRef** or **Relation** of an existing constraint, use SolverDelete to delete the old constraint, then use SolverAdd to add the constraint in the form you want.

---

## SolverChange (Form 2)

Equivalent to choosing Solver... from the Tools menu, pressing the Variables button, and then pressing the Change button in the Solver Parameters dialog box. Changes a set of decision variable cells. This form is supported only by the Premium Solver products.

*VBA Syntax*

### **SolverChange (CellRef:=, Relation:=, Comment:=, Report:=)**

**CellRef** is a reference to a cell or a range of cells on the active worksheet, currently defined in the Variable Cells list box as a set of decision variable cells.

**Relation** is a reference to a different cell or range of cells on the active worksheet, which will replace **CellRef** as a new set of variable cells.

**Comment** is a string corresponding to the Comment field in the Change Variable Cells dialog, only in the Premium Solver products.

**Report** is no longer used, but is included for compatibility with previous versions of the Premium Solver products.

#### **Remarks**

If **CellRef** does not match any existing set of variable cells, the function returns the value 1 and no action is taken.

---

## **SolverDelete (Form 1)**

Equivalent to choosing Solver... from the Tools menu and pressing the Delete button in the Solver Parameters dialog box. Deletes an existing constraint.

*VBA Syntax*

### **SolverDelete (CellRef:=, Relation:=, FormulaText:=)**

For an explanation of the arguments and selection of constraints, see **SolverAdd**. The **FormulaText** argument is optional, but if present, it is used to confirm that the correct constraint block is being deleted.

#### **Remarks**

If the combination of **CellRef** and **Relation** does not match any existing constraint, the function returns the value 4 and no action is taken. If the constraint is found, it is deleted, and the function returns the value 0.

---

## **SolverDelete (Form 2)**

Equivalent to choosing Solver... from the Tools menu, pressing the Variables button, and then pressing the Delete button in the Solver Parameters dialog box. Deletes an existing set of variable cells. This form is supported only by the Premium Solver products.

*VBA Syntax*

### **SolverDelete (CellRef:=)**

**CellRef** is a reference to a cell or a range of cells on the active worksheet, currently defined in the Variable Cells list box as decision variable cells.

#### **Remarks**

If **CellRef** does not match any existing set of variable cells, the function returns the value 1 and no action is taken. If the variable cells are found, they are deleted, and the function returns the value 0.

---

## SolverFinish

Equivalent to selecting options and clicking OK in the Solver Results dialog that appears when the solution process is finished. The dialog box will *not* be displayed.

*VBA Syntax*

**SolverFinish (KeepFinal:=, ReportArray:=, ReportDesc:=, OutlineReports:=)**

The **ReportDesc** and **OutlineReports** arguments are available only in the Premium Solver products.

**KeepFinal** is the number 1, 2 or 3 and specifies whether to keep or discard the final solution. If **KeepFinal** is 1 or omitted, the final solution values are kept in the variable cells. If **KeepFinal** is 2, the final solution values are discarded and the former values of the variable cells are restored.

If **KeepFinal** is 3 – which can occur only if you are solving a problem with integer constraints which has no feasible integer solution – Solver will immediately re-solve the “relaxation” of the problem, temporarily ignoring the integer constraints. In this case, **SolverFinish** will return when the solution process is complete, and its return value will be one of the integer values ordinarily returned by **SolverSolve**.

**ReportArray** is an array argument specifying what reports should be produced. If the Solver found a solution, it may have any of the following values:

<b>If ReportArray is</b>	<b>The Solver creates</b>
Array(1)	An Answer Report
Array(2)	A Sensitivity Report
Array(3)	A Limits Report
Array(4)	A Solutions Report

Array(4) is used only for integer programming and global optimization problems. A combination of these values produces multiple reports. For example, if **ReportArray** = Array(1,2), the Solver will create an Answer Report and a Sensitivity Report.

If you are using the Interval Global Solver engine, you can produce an Answer Report when **SolverSolve** returns 0, and a Solutions Report if you successfully solve a system of inequalities or a square system of equations:

<b>If ReportArray is</b>	<b>The Solver creates</b>
Array(1)	An Answer Report
Array(2)	A Solutions Report

If you are using the Evolutionary Solver engine, you can produce an Answer Report, a Population Report or a Solutions Report unless **SolverSolve** returns 18, 19 or 20 (which means that the Solver returned an error before a population was formed):

<b>If ReportArray is</b>	<b>The Solver creates</b>
Array(1)	An Answer Report
Array(2)	A Population Report
Array(3)	A Solutions Report

If the Solver found that the linearity conditions for the selected Solver engine were not satisfied (**SolverSolve** returns 7), you can produce a Linearity Report or a Scaling Report:

<b>If ReportArray is</b>	<b>The Solver creates</b>
Array(1)	A Linearity Report
Array(2)	A Scaling Report

If the Solver could not find a feasible solution (**SolverSolve** returns 5), you can produce either version of the Feasibility Report, or a Scaling Report:

<b>If ReportArray is</b>	<b>The Solver creates</b>
Array(1)	A Feasibility Report
Array(2)	A Feasibility-Bounds Report
Array(3)	A Scaling Report

If you are using the Premium Solver Platform and a field-installable Solver engine, it may produce some or all of the reports mentioned above and/or its own custom reports. To determine what you should use for the **ReportArray** argument, solve a problem interactively with this Solver engine, and examine the Reports list box in the Solver Results dialog. Then use the ordinal position of the report you want:

<b>If ReportArray is</b>	<b>The Solver creates</b>
Array(1)	The first report listed
Array(2)	The second report listed (and so on)

**ReportDesc** is an array of character strings that allows you to select reports by their names, rather than their ordinal positions in the Reports list. For example, you can select an Answer Report with Array ("Answer"), or both the Answer Report and the Sensitivity Report with Array ("Answer", "Sensitivity"). The possible strings are:

"Answer"	Answer Report
"Sensitivity"	Sensitivity Report
"Limits"	Limits Report
"Solutions"	Solutions Report
"Population"	Population Report
"Linearity"	Linearity Report
"Feasibility"	Feasibility Report (full version)
"Feasibility-Bounds"	Feasibility Report (w/o bounds)
"Scaling"	Scaling Report

The report names you can include in the array depend on the currently selected Solver engine and the integer value returned by **SolverSolve**, as described above.

**OutlineReports** is a logical value corresponding to the Outline Reports check box. If TRUE, any reports you select will be produced in outlined format, and comments (if any) associated with each block of variables and constraints will be included in the report; if it is FALSE, the reports will be produced in "regular" format.

---

## SolverFinishDialog

Equivalent to selecting options in the Solver Results dialog that appears when the solution process is finished. The dialog box *will* be displayed, and the user will be able to change the options that you initially specify.

*VBA Syntax*

**SolverFinishDialog (KeepFinal:=, ReportArray:=, ReportDesc:=, OutlineReports:=)**

For an explanation of the arguments of this function, see **SolverFinish**.

---

## SolverGet

Returns information about the current Solver problem. The settings are specified in the Solver Parameters and Solver Options dialog boxes, or with the other Solver

functions described in this chapter. Values of the `TypeNum:=` argument from 1 to 18 are supported by the standard Excel Solver.

**SolverGet** is provided for compatibility with the standard Excel Solver and earlier versions of the Premium Solver products. For programmatic control of new features and options included in Version 5.0 or later of the Premium Solver products, see the dialog-specific “Get” functions in the sections “Solver Model VBA Functions” and “Premium VBA Functions.”

#### *VBA Syntax*

#### **SolverGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want. The following settings are specified in the Solver Parameters dialog box.

TypeNum	Returns
1	The reference in the Set Cell box, or the #N/A error value if Solver has not been used on the active document
2	A number corresponding to the Equal To option 1 = Max 2 = Min 3 = Value Of
3	The value in the Value Of box
4	The reference in the Changing Cells box (in the Premium Solvers, only the first entry in the Variables list box)
5	The number of entries in the Constraints list box
6	An array of the left hand sides of the constraints as text
7	An array of numbers corresponding to the relations between the left and right hand sides of the constraints: 1 = <= 2 = = 3 = >= 4 = int 5 = bin 6 = dif 7 = soc 8 = src
8	An array of the right hand sides of the constraints as text

The following settings are specified in the Solver Options dialog box:

TypeNum	Returns
9	The Max Time value (as a number in seconds)
10	The Iterations value (max number of iterations)
11	The Precision value (as a decimal number)
12	The integer Tolerance value (as a decimal number)
13	In the standard Solver: TRUE if the Assume Linear Model check box is selected; FALSE otherwise. In the Premium Solvers: TRUE if the linear Simplex or LP/Quadratic Solver is selected; FALSE if any other Solver is selected
14	TRUE if the Show Iteration Result check box is selected; FALSE otherwise

- 15 TRUE if the Use Automatic Scaling check box is selected;  
FALSE otherwise
- 16 A number corresponding to the type of Estimates:  
1 = Tangent  
2 = Quadratic
- 17 A number corresponding to the type of Derivatives:  
1 = Forward  
2 = Central
- 18 A number corresponding to the type of Search:  
1 = Newton  
2 = Conjugate

The following settings are supported by the Premium Solver products:

TypeNum	Returns
19	The Convergence value (as a decimal number) in the nonlinear GRG Solver
20	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
21	The Integer Cutoff value (as a decimal number)
22	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise
23	An array of the entries in the Variables list box as text
24	A number corresponding to the Solver engine dropdown list for the currently selected Solver engine: 1 = Nonlinear GRG Solver 2 = Simplex or LP/Quadratic Solver 3 = Evolutionary Solver 4 = Interval Global Solver 5 = SOCP Barrier Solver In the Premium Solver Platform, other values may be returned for field-installable Solver engines
25	The Pivot Tolerance (as a decimal number) in the Simplex, LP/Quadratic, and Large-Scale LP Solvers
26	The Reduced Cost Tolerance (as a decimal number) in the Simplex, LP/Quadratic, and Large-Scale LP Solvers
27	The Coefficient Tolerance (as a decimal number) in the Large-Scale LP Solver
28	The Solution Tolerance (as a decimal number) in the Large-Scale LP Solver
29	TRUE if the Estimates option in the GRG Solver is set to Tangent; FALSE if the Estimates option is set to Quadratic
30	A number corresponding to the type of Scaling in the Large-Scale LP Solver: 1 = None 2 = Row Only 3 = Row & Col

**SheetName** is the name of a worksheet that contains the Solver problem for which you want information. If **SheetName** is omitted, it is assumed to be the active sheet.

---

## SolverLoad

Equivalent to choosing Solver... from the Tools menu, choosing the Options button from the Solver Parameters dialog box, and choosing the Load Model... button in the Solver Options dialog box. Loads Solver model specifications that you have previously saved on the worksheet. The **Format** and **ModelName** arguments are supported only in Version 7.0 or later of the Premium Solver products.

*VBA Syntax*

**SolverLoad (LoadArea:=, Merge:=, Format:=, ModelName:=)**

**LoadArea** is a reference to a range of cells from which you want to load a complete model specification. In Solver versions prior to V7.0, **LoadArea** must be a reference on the active worksheet; in Version 7.0 or later it can be on any worksheet.

**Merge** is a logical value corresponding to either the Merge button or the Replace button in the dialog that appears after you select the **LoadArea** reference and click OK. If it is TRUE, the variable cell selections and constraints from the **LoadArea** are merged with the currently defined variables and constraints. If FALSE, the current model specifications and options are erased (equivalent to a call to the **SolverReset** function) before the new specifications are loaded.

**Format** corresponds to the Format dropdown list in the Load Model dialog: 1 for “Classic” format and 2 for “PSI Function” format.

In “Classic” format, the first cell in **LoadArea** contains a formula for the Set Cell edit box; the second cell contains a formula for the changing cells; subsequent cells contain additional variable selections and constraints in the form of logical formulas. The final cells optionally contain an array of Solver option values.

In “PSI Function” format, the cells contain calls to functions such as PsiVar(), PsiCon(), PsiObj() and PsiOption(). For details on these functions, see “Defining Your Model with PSI Functions” in the chapter “Building Solver Models.”

**ModelName** is used only when **Format** = 2, it overrides the **LoadArea** argument and specifies either the name of a worksheet containing the model to be loaded, or the name of a model previously saved in PSI function format.

---

## SolverOk

Equivalent to choosing Solver... from the Tools menu and specifying options in the Solver Parameters dialog. Specifies basic Solver options. The dialog box will *not* be displayed.

*VBA Syntax*

**SolverOk (SetCell:=, MaxMinVal:=, Valueof:=, ByChange:=, Engine:=, EngineDesc:=)**

**SetCell** corresponds to the Set Cell box in the Solver Parameters dialog box (the objective function in the optimization problem). **SetCell** must be a reference to a cell on the active worksheet. If you enter a cell, you must enter a value for **MaxMinVal**.

**MaxMinVal** corresponds to the options Max, Min and Value Of in the Solver Parameters dialog box. Use this option only if you entered a reference for **SetCell**.

MaxMinVal	Option specified
1	Maximize
2	Minimize
3	Value Of



**ValueOf** is the number that becomes the target for the cell in the Set Cell box if **MaxMinVal** is 3. **ValueOf** is ignored if the cell is being maximized or minimized.

**ByChange** indicates the changing cells (decision variables), as entered in the By Changing Variable Cells edit box. **ByChange** must be a cell reference (usually a cell range or multiple reference) on the active worksheet. In the Premium Solver products, you can add more changing cell references using Form 2 of the **SolverAdd** function.

**Engine** corresponds to the engine dropdown list in the Solver Parameters dialog. See the **EngineDesc** argument for an alternative way of selecting the Solver “engine.”

Engine	Solver engine specified
1	Nonlinear GRG Solver
2	Simplex or LP/Quadratic Solver
3	Evolutionary Solver
4	Interval Global Solver
5	SOCP Barrier Solver

In the Premium Solver Platform, other values for **Engine** may be specified to select field-installable Solver engines. However, these values depend on the ordinal position of the Solver engine in the dropdown list, which may change when additional Solver engines are installed.

**EngineDesc**, which is supported only by the Premium Solver products, provides an alternative way to select the Solver engine from the dropdown list in the Solver Parameters dialog. **EngineDesc** allows you to select a Solver engine by name rather than by ordinal position in the list:

EngineDesc	Solver engine specified
“Standard GRG Nonlinear”	Nonlinear GRG Solver
“Standard Simplex LP”	Simplex LP Solver
“Standard LP/Quadratic”	LP/Quadratic Solver
“Standard Evolutionary”	Evolutionary Solver
“Standard Interval Global”	Interval Global Solver
“Standard SOCP Barrier”	SOCP Barrier Solver
“KNITRO Solver”	KNITRO Solver
“Large-Scale GRG Solver”	Large-Scale GRG Solver
“Large-Scale LP Solver”	Large-Scale LP Solver
“Large-Scale SQP Solver”	Large-Scale SQP Solver
“MOSEK Solver Engine”	MOSEK Solver Engine
“OptQuest Solver”	OptQuest Solver
“XPRESS Solver Engine”	XPRESS Solver Engine

---

## SolverOkDialog

Equivalent to choosing Solver... from the Tools menu and specifying options in the Solver Parameters dialog. The Solver Parameters dialog box *will* be displayed, and the user will be able to change the options you initially specify.

*VBA Syntax*

**SolverOkDialog (SetCell:=, MaxMinVal:=, Valueof:=, ByChange:=, Engine:=, EngineDesc:=)**

For an explanation of the arguments of this function, see SolverOk.

---

## SolverOptions

Equivalent to choosing Solver... from the Tools menu, then choosing the Options button in the Solver Parameters dialog box. Specifies Solver algorithmic options. Arguments supported by the standard Excel Solver include **MaxTime**, **Iterations**, **Precision**, **AssumeLinear**, **StepThru**, **Estimates**, **Derivatives**, **SearchOption**, **IntTolerance**, **Scaling**, **Convergence** and **AssumeNonNeg**.

**SolverOptions** is provided for compatibility with the standard Excel Solver and early versions of the Premium Solver products. For programmatic control of new features and options included in the Premium Solver products, see the functions in the sections “Solver Model VBA Functions” and “Premium VBA Functions.”

*VBA Syntax*

**SolverOptions** (**MaxTime:=**, **Iterations:=**, **Precision:=**, **AssumeLinear:=**, **StepThru:=**, **Estimates:=**, **Derivatives:=**, **SearchOption:=**, **IntTolerance:=**, **Scaling:=**, **Convergence:=**, **AssumeNonNeg:=**, **IntCutoff:=**, **BypassReports:=**, **PivotTol:=**, **ReducedTol:=**, **CoeffTol:=**, **SolutionTol:=**, **Crash:=**, **ScalingOption:=**)

The arguments correspond to the options in the Solver Options dialog box. If an argument is omitted, the Solver maintains the current setting for that option. If any of the arguments are of the wrong type, the function returns the #N/A error value. If all arguments are of the correct type, but an argument has an invalid value, the function returns a positive integer corresponding to its position. A zero return value indicates that all options were accepted.

**MaxTime** must be an integer greater than zero. It corresponds to the Max Time edit box.

**Iterations** must be an integer greater than zero. It corresponds to the Iterations edit box.

**Precision** must be a number between zero and one, but not equal to zero or one. It corresponds to the Precision edit box.

**AssumeLinear** is a logical value corresponding to the Assume Linear Model check box. This argument is included for compatibility with the standard Microsoft Excel Solver. It is ignored by the Premium Solver products, which use the **Engine** or **EngineDesc** argument of **SolverOk** or **SolverOkDialog** instead.

**StepThru** is a logical value corresponding to the Show Iteration Results check box. If TRUE, Solver pauses at each trial solution; if FALSE it does not. If you have supplied **SolverSolve** with a valid VBA function argument, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

**Estimates** is the number 1 or 2 and corresponds to the Estimates option: 1 for Tangent and 2 for Quadratic.

**Derivatives** is the number 1 or 2 and corresponds to the Derivatives option: 1 for Forward and 2 for Central.

**SearchOption** is the number 1 or 2 and corresponds to the Search option: 1 for Newton and 2 for Conjugate.

**IntTolerance** is a number between zero and one, corresponding to the Tolerance edit box. This argument applies only if integer constraints have been defined.

**Scaling** is a logical value corresponding to the Use Automatic Scaling check box. If TRUE, then Solver rescales the objective and constraints internally to similar orders of magnitude. If FALSE, Solver uses values directly from the worksheet. In early

Excel versions, this option affects the nonlinear GRG Solver only; in Excel 97, 2000, XP, 2003 and 2007 and the Premium Solver products, this option affects all Solver engines.

**Convergence** is a number between zero and one, but not equal to zero or one. It corresponds to the Convergence box.

**AssumeNonNeg** is a logical value corresponding to the Assume Non-Negative check box. If TRUE, Solver supplies a lower bound of zero for all variables without explicit lower bounds in the Constraint list box. If FALSE, no action is taken.

**IntCutoff** is a number corresponding to the Integer Cutoff edit box. This argument applies only if integer constraints have been defined.

**BypassReports** is a logical value corresponding to the Bypass Solver Reports check box. If TRUE, the Solver will skip preparing the information needed to create Solver Reports. If FALSE, the Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution process considerably.

**PivotTol** is a number between zero and one, corresponding to the Pivot Tolerance edit box for the Simplex LP Solver in the Premium Solver.

**ReducedTol** is a number between zero and one, corresponding to the Reduced Cost Tolerance edit box for the Simplex LP Solver in the Premium Solver.

**CoeffTol** is no longer used, but is included for compatibility with previous versions of the Premium Solver products.

**SolutionTol** is no longer used, but is included for compatibility with previous versions of the Premium Solver products.

**Crash** is no longer used, but is included for compatibility with previous versions of the Premium Solver products.

**ScalingOption** is no longer used, but is included for compatibility with previous versions of the Premium Solver products.

---

## SolverReset

Equivalent to choosing Solver... from the Tools menu and choosing the Reset All button in the Solver Parameters dialog box. Erases all cell selections and constraints from the Solver Parameters dialog box, and restores all the settings on the Solver Options, Limit Options and Integer Options dialog tabs to their defaults. The SolverReset function may be automatically performed when you call SolverLoad.

*VBA Syntax*

**SolverReset**

---

## SolverSave

Equivalent to choosing Solver... from the Tools menu, choosing the Options button from the Solver Parameters dialog box, and choosing the Save Model... button in the Solver Options dialog box. Saves the model specifications on the worksheet. The **Format** and **ModelName** arguments are supported only in Version 7.0 or later of the Premium Solver products.

*VBA Syntax*

**SolverSave (SaveArea:=, Format:=, ModelName:=)**

**SaveArea** is a reference to a range of cells or to the topmost cell in a column of cells where you want to save the current model's specifications. In Solver versions prior to V7.0, **SaveArea** must be a reference on the active worksheet; in Version 7.0 or later it can be on any worksheet.

**Format** corresponds to the Format dropdown list in the Load Model dialog: 1 for "Classic" format and 2 for "PSI Function" format.

In "Classic" format, the first cell in **SaveArea** contains a formula for the Set Cell edit box; the second cell contains a formula for the changing cells; subsequent cells contain additional variable selections and constraints in the form of logical formulas. The final cells optionally contain an array of Solver option values.

In "PSI Function" format, the cells contain calls to functions such as `PsiVar()`, `PsiCon()`, `PsiObj()` and `PsiOption()`. For details on these functions, see "Defining Your Model with PSI Functions" in the chapter "Building Solver Models."

**ModelName** is used only when **Format** = 2, and specifies a text name for the model. This name is saved as the last argument of each PSI Function call in the saved model specifications.

#### Remarks

If you specify only one cell for **SaveArea**, the area is extended downwards for as many cells as are required to hold the model specifications.

If you specify more than one cell and the area is too small for the problem, the model specifications will not be saved, and the function will return the value 2.

---

## SolverSolve

Equivalent to choosing Solver... from the Tools menu and choosing the Solve button in the Solver Parameters dialog box. If successful, returns an integer value indicating the condition that caused the Solver to stop, as described below.

*VBA Syntax*

**SolverSolve (UserFinish:=, ShowRef:=)**

**UserFinish** is a logical value specifying whether to show the standard Solver Results dialog box.

If **UserFinish** is TRUE, SolverSolve returns its integer value without displaying anything. Your VBA code should decide what action to take (for example, by examining the return value or presenting its own dialog box); it *must* call **SolverFinish** in any case to return the worksheet to its proper state.

If **UserFinish** is FALSE or omitted, Solver displays the standard Solver Results dialog box, allowing the user to keep or discard the final solution values, and optionally produce reports.

**ShowRef** is a VBA function to be called in place of displaying the Show Trial Solution dialog box. It is used when you want to gain control whenever Solver finds a new "Trial Solution" value, the user presses the ESC key, or a limit on the solution process is exceeded. Here is an example of defining and using the argument **ShowRef**:

```
Sub Test
    answer = SolverSolve(True, ShowTrial )
End Sub

Function ShowTrial(Reason As Integer)
```

```
Msgbox Reason
ShowTrial = 1
End Function
```

The argument **Reason**, which *must* be present, is an integer value from 1 to 5:

1. Function called (on every iteration) because the Show Iteration Results box in the Solver Options dialog was checked, *or* function called because the user pressed ESC to interrupt the Solver.
2. Function called because the Max Time limit in the Solver Options dialog was exceeded.
3. Function called because the Max Iterations limit in the Solver Options dialog was exceeded.
4. Function called because the Max Subproblems limit on the Integer Options or Limit Options dialog tab was exceeded (Premium Solver products only).
5. Function called because the Max Integer Sols limit on the Integer Options dialog tab or the Max Feasible Sols limit on the Limit Options dialog tab was exceeded (Premium Solver products only).

The function must return 0 if the Solver should stop (same as the Stop button in the Show Trial Solution dialog), 1 if it should continue running (same as the Continue button), or 2 if it should restart the solution process (same as the Restart button).

**Note:** In the standard Excel Solver and the Premium Solver prior to V5.0, the function should return FALSE instead of 0 to stop, or TRUE instead of 1 to continue running; the restart alternative is not available.

Your VBA function can inspect the current solution values on the worksheet, or take other actions such as saving or charting the intermediate values. However, it should *not* alter the values in the variable cells, or alter the formulas in the objective and constraint cells, as this could adversely affect the solution process.

In the Premium Solver Platform, if the PSI Interpreter is used, the worksheet is not updated with new variable values until the end of the solution process; you cannot use the traditional VBA functions to inspect variable values on each Trial Solution, but you *can* use the object-oriented API to do this. See “Evaluators Called During the Solution Process” in the chapter “Using the Object-Oriented API” for details.

### Remarks

If a Solver problem has not been completely defined, **SolverSolve** returns the #N/A error value. Otherwise the Solver engine is started, and the problem specifications are passed to it. When the solution process is complete, **SolverSolve** returns an integer value indicating the stopping condition. The standard Excel Solver returns values from 0 to 13; the Premium Solver products return values from -1 to 21. When the Interval Global Solver or field-installable Solver engines are used, the Premium Solver Platform may return engine-specific values for custom stopping conditions, starting at 1000.

#### Value Stopping Condition

-1	A licensing problem was detected, or your trial license has expired.
0	Solver found a solution. All constraints and optimality conditions are satisfied.

1	Solver has converged to the current solution. All constraints are satisfied.
2	Solver cannot improve the current solution. All constraints are satisfied.
3	Stop chosen when the maximum iteration limit was reached.
4	The Set Cell values do not converge.
5	Solver could not find a feasible solution.
6	Solver stopped at user's request.
7	The linearity conditions required by this Solver engine are not satisfied.
8	The problem is too large for Solver to handle.
9	Solver encountered an error value in a target or constraint cell.
10	Stop chosen when the maximum time limit was reached.
11	There is not enough memory available to solve the problem.
12	Error <i>condition</i> at cell address (Premium Solver Platform only).
13	Error in model. Please verify that all cells and constraints are valid.
14	Solver found an integer solution within tolerance. All constraints are satisfied.
15	Stop chosen when the maximum number of feasible [integer] solutions was reached.
16	Stop chosen when the maximum number of feasible [integer] subproblems was reached.
17	Solver converged in probability to a global solution.
18	All variables must have both upper and lower bounds.
19	Variable bounds conflict in binary or alldifferent constraint.
20	Lower and upper bounds on variables allow no feasible solution.
21	Solver encountered an error computing derivatives (Premium Solver Platform only).
1000	Interval Global Solver requires Solve With Automatic and strictly smooth functions (Premium Solver Platform only).
1001	Function cannot be evaluated for given real or interval arguments (Premium Solver Platform only).
1002	Solution found, but not proven globally optimal (Premium Solver Platform only).

---

## Solver Model VBA Functions

The VBA functions in this section are available only in the Premium Solver and Premium Solver Platform. You can use these functions to programmatically use the Polymorphic Spreadsheet Interpreter to check your model for Gradients, Structure and Convexity, obtain model statistics, produce the Structure Report and

Transformation Report, determine whether and how the Interpreter will be used when you call SolverSolve, and control the Interpreter's advanced options.

---

## SolverModel

Equivalent to choosing Solver... from the Tools menu, choosing the Model button in the Solver Parameters dialog, setting options in the Solver Model dialog, and clicking Close. Specifies options for the Polymorphic Spreadsheet Interpreter.

*VBA Syntax*

**SolverModel (Interpreter:=, CheckFor:=, SolveTransformed:=, ShowTransformations:=, ShowExceptions:=, DesiredModel:=, Interactive:=, UsePsiFunctions:=, Engines:=, ReqSmooth:=, FastSetup:=, Sparse:=, ActiveOnly:=)**

The arguments correspond to the options in the Solver Model dialog box. The Check For option appears on all four tabs of this dialog; the Interpreter option appears on the Original tab; the Solve Transformed Problem and Show Transformations options appear on the Transformed tab; the Show Exceptions and Desired Model options appear on the Diagnosis tab; and the remaining options appear on the Options tab. If an argument is omitted, the Solver maintains the current setting for that option. If any of the arguments are of the wrong type, the function returns the #N/A error value. If all arguments are of the correct type, but an argument has an invalid value, the function returns a positive integer corresponding to its position. A zero return value indicates that all options were accepted.

**Interpreter** is a number corresponding to the option selected in the Solve With option group on the Original tab:

Interpreter	Action on SolverSolve
1	Use PSI Interpreter
2	Use Excel Interpreter

**CheckFor** is a number corresponding to the option selected in the Check For option group:

CheckFor	Option Selected
1	Gradients
2	Structure
3	Convexity
4	Automatic

**SolveTransformed** is a logical value corresponding to the Solve Transformed Problem check box on the Transformed tab. If TRUE, the Solver solves the Transformed problem when the SolverSolve function is called. If FALSE, the Solver solves the Original problem when the SolverSolve function is called.

**ShowTransformations** is a logical value corresponding to the Show Transformations check box on the Transformed tab. If TRUE, the Solver produces a Transformation Report when the SolverModelCheck function is called. If FALSE, the report is not produced.

**ShowExceptions** is a logical value corresponding to the Show Exceptions to Desired Model check box on the Options tab. If TRUE, the Solver produces a Structure Report when the SolverModelCheck function is called. If FALSE, the report is not produced.

**DesiredModel** is a number corresponding to the option selected in the Desired Model option group:

<b>DesiredModel</b>	<b>Desired Model Type</b>
1	Linear
2	Quadratic
3	Conic
4	Nonlinear
5	Nonsmooth

**Interactive** is a logical value corresponding to the Use Interactive Optimization check box on the Options tab. If TRUE, Interactive Optimization is enabled when Excel is in worksheet Ready mode. If FALSE, Interactive Optimization is disabled.

**UsePsiFunctions** is a logical value corresponding to the Use PSI Functions check box on the Options tab. If TRUE, the Solver recognizes PSI functions such as PsiVar(), PsiCon(), PsiObj(), etc. that define the model. If FALSE, the Solver ignores any PSI functions it finds on the worksheet.

**Engines** is a number corresponding to the option selected in the Select Solver Engines Based on Model Type option group:

<b>Engines</b>	<b>Engines Shown in List</b>
1	All
2	Valid
3	Good
4	Best

**ReqSmooth** is a logical value corresponding to the Req Smooth check box in the Advanced options group on the Options tab. If TRUE, the Solver treats the special functions ABS, IF, MAX, MIN, and SIGN as non-smooth. If FALSE, the Solver treats these functions as smooth nonlinear.

**FastSetup** is a logical value corresponding to the Fast Setup check box in the Advanced options group on the Options tab. If TRUE, the Solver attempts to use old-style Fast Problem Setup before using the Polymorphic Spreadsheet Interpreter. If FALSE, the Solver uses the Interpreter directly. If the Interpreter option is set to Excel Interpreter, this option is ignored and the Solver will always attempt to use old-style Fast Problem Setup.

**Sparse** is a logical value corresponding to the Sparse check box in the Advanced options group on the Options tab. If TRUE, the Polymorphic Spreadsheet Interpreter will operate internally in its own Sparse mode. If FALSE, the Interpreter operates in Dense mode.

**ActiveOnly** is a logical value corresponding to the Active Only check box in the Advanced options group on the Options tab. If TRUE, the Polymorphic Spreadsheet Interpreter will analyze objective and constraint function formulas only on the active sheet. If FALSE, the Interpreter analyze all objective and constraint function formulas in the workbook.

**SolveWith** is included for compatibility with the Premium Solver Platform V6.0 – V6.5. In V7.0, it corresponds to the Interpreter and Check For options, as follows:

<b>SolveWith</b>	<b>Equivalent To</b>
1	Interpreter = Excel
2	Interpreter = PSI, CheckFor = Gradients
3	Interpreter = PSI, CheckFor = Structure
4	Interpreter = PSI, CheckFor = Convexity
5	Interpreter = PSI, CheckFor = Automatic



---

## SolverModelCheck

Equivalent to choosing Solver... from the Tools menu, choosing the Model button in the Solver Parameters dialog box, and clicking the Check Model button in the Solver Model dialog. The type of analysis performed is determined by the current setting of the SolverModel CheckFor argument.

*VBA Syntax*

### SolverModelCheck (Transformed:=)

**Transformed** is a logical value that corresponds to the tab (Original or Transformed) active when the Check Model button is pressed. If TRUE, the Polymorphic Spreadsheet Interpreter checks the Transformed model. If FALSE, the Interpreter checks the Original model.

---

## SolverModelGet

Returns Solver Model option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Model dialog. The available settings include the “read-only” edit boxes on the Original and Transformed tabs of the Solver Model dialog; these values are valid only after you call SolverModelCheck (FALSE) and SolverModelCheck (TRUE) respectively. SolverModelGet(28) will return the type of model (Original or Transformed) as determined by the most recent call to SolverModelCheck.

*VBA Syntax*

### SolverModelGet (TypeNum:=, SheetName:=)

**TypeNum** is a number specifying the type of information you want. The following settings are specified in the Solver Model dialog box.

TypeNum	Returns
1	The All Variables value on the Original tab
2	The Smooth Variables value on the Original tab
3	The Quadratic Variables value on the Original tab
4	The Linear Variables value on the Original tab
5	The Bounds value on the Original tab
6	The Integers value on the Original tab
7	The All Functions value on the Original tab
8	The Smooth Functions value on the Original tab
9	The Quadratic Functions value on the Original tab
10	The Linear Functions value on the Original tab
11	The All NonZeroes value on the Original tab
12	The Smooth NonZeroes value on the Original tab
13	The Quadratic NonZeroes value on the Original tab
14	The Linear NonZeroes value on the Original tab
15	The Sparsity % value on the Original tab

16       The Total Cells value on the Original tab

17       A number corresponding to the Check For option: 1 for Gradients, 2 for Structure, or 3 for Convexity

18       TRUE if the Solve Transformed Problem check box is selected; FALSE otherwise

19       TRUE if the Show Transformations check box is selected; FALSE otherwise

20       A number corresponding to the Desired Model option: 1 for Linear, 2 for Quadratic, 3 for Conic, 4 for Nonlinear, or 4 for Nonsmooth

21       A number corresponding to the V6 Solve With option: 1 for No Action (Excel Interpreter), 2 for Gradients, 3 for Structure, 4 for Convexity, or 5 for Automatic

22       A number corresponding to the Engines option: 1 for All, 2 for Valid, 3 for Good, or 4 for Best

23       TRUE if the Req Smooth check box is selected; FALSE otherwise

24       TRUE if the Fast Setup check box is selected; FALSE otherwise

25       TRUE if the Sparse check box is selected; FALSE otherwise

26       TRUE if the Active Only check box is selected; FALSE otherwise

27       TRUE if the Show Exceptions to Desired Model check box is selected; FALSE otherwise

28       A string corresponding to the type of model: "LP", "QP", "QCP", "NLP", "NSP", or "Unknown". "LP", "QP", "QCP", or "NLP" may be followed by a space and "Convex" or "NonCvx".

29       TRUE if the Interactive Optimization check box is selected; FALSE otherwise

30       TRUE if the Use PSI Functions check box is selected; FALSE otherwise

31       The All Variables value on the Transformed tab

32       The Smooth Variables value on the Transformed tab

33       The Quadratic Variables value on the Transformed tab

34       The Linear Variables value on the Transformed tab

35       The Bounds value on the Transformed tab

36       The Integers value on the Transformed tab

37       The All Functions value on the Transformed tab

38       The Smooth Functions value on the Transformed tab

39       The Quadratic Functions value on the Transformed tab

40       The Linear Functions value on the Transformed tab

41       The All NonZeroes value on the Transformed tab

42       The Smooth NonZeroes value on the Transformed tab

43       The Quadratic NonZeroes value on the Transformed tab

44       The Linear NonZeroes value on the Transformed tab

45           The Sparsity % value on the Transformed tab  
46           The Total Cells value on the Transformed tab

---

## SolverDependents

This function is included for backward compatibility only; use the SolverModel-Check function in new applications. Equivalent to choosing Solver... from the Tools menu, choosing the Model button in the Solver Parameters dialog box, selecting the Structure option in the Check For option group, and clicking the Check Model button in the Solver Model dialog.

*VBA Syntax*

**SolverDependents**

---

## SolverFormulas

This function is included for backward compatibility only; use the SolverModel-Check function in new applications. Equivalent to choosing Solver... from the Tools menu, choosing the Model button in the Solver Parameters dialog box, selecting the Gradients option in the Check For option group, and clicking the Check Model button in the Solver Model dialog.

*VBA Syntax*

**SolverFormulas**

---

# Premium VBA Functions

The VBA functions in this section were first introduced in Version 3.0 of the Premium Solver products, and expanded in later versions. To control most of the new features and options in the Premium Solver products, you'll need to use these functions – notably, the **SolverEVOptions**, **SolverIGOptions**, **SolverLimOptions** and **SolverIntOptions** functions. (Or, in Version 7.0 and later, you can use the new object-oriented API to control these features and options.) If you want to write VBA code that can be used with both the standard Solver and the Premium Solver products, you should use only functions in the section “Standard VBA Functions.”

---

## SolverEVGet

Returns Evolutionary Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the Evolutionary Solver is selected in the Solver Engine dropdown list.

*VBA Syntax*

**SolverEVGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want. The following settings are specified in the Evolutionary Solver Options dialog box.

TypeNum	Returns
1	The Max Time value (as a number in seconds)
2	The Iterations value (max number of iterations)

3	The Precision value (as a decimal number)
4	The Convergence value (as a decimal number)
5	The Population Size value (as a decimal number)
6	The Mutation Rate value (as a decimal number)
7	TRUE if the Require Bounds on Variables check box is selected; FALSE otherwise
8	TRUE if the Show Iteration Result check box is selected; FALSE otherwise
9	TRUE if the Use Automatic Scaling check box is selected; FALSE otherwise
10	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
11	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise
12	The Random Seed value (as a decimal number)
13	A number corresponding to the Local Search option: 1 for Randomized Local Search, 2 for Deterministic Pattern Search, 3 for Gradient Local Search, or 4 for Automatic Choice
14	TRUE if the Fix Nonsmooth Variables check box is selected; FALSE otherwise

**SheetName** is the name of a worksheet that contains the Solver problem for which you want information. If **SheetName** is omitted, it is assumed to be the active sheet.

---

## SolverEVOptions

Equivalent to choosing Solver... from the Tools menu and then choosing the Options button in the Solver Parameters dialog box when the Evolutionary Solver is selected in the Solver Engine dropdown list. Specifies options for the Evolutionary Solver.

*VBA Syntax*

**SolverEVOptions (MaxTime:=, Iterations:=, Precision:=, Convergence:=, PopulationSize:=, MutationRate:=, RandomSeed:=, RequireBounds:=, StepThru:=, Scaling:=, AssumeNonNeg:=, BypassReports:=, LocalSearch:=, FixNonSmooth:=)**

The arguments correspond to the options in the Solver Options dialog box. If an argument is omitted, the Solver maintains the current setting for that option. If any of the arguments are of the wrong type, the function returns the #N/A error value. If all arguments are of the correct type, but an argument has an invalid value, the function returns a positive integer corresponding to its position. A zero return value indicates that all options were accepted.

**MaxTime** must be an integer greater than zero. It corresponds to the Max Time edit box.

**Iterations** must be an integer greater than zero. It corresponds to the Iterations edit box.

**Precision** must be a number between zero and one, but not equal to zero or one. It corresponds to the Precision edit box.

**Convergence** is a number between zero and one, but not equal to zero or one. It corresponds to the Convergence box.

**PopulationSize** must be an integer greater than or equal to zero. It corresponds to the Population Size edit box.

**MutationRate** must be a number between zero and one, but not equal to zero or one. It corresponds to the Mutation Rate edit box.

**RandomSeed** must be an integer greater than zero. It corresponds to the Random Seed edit box.

**RequireBounds** is a logical value corresponding to the Require Bounds on Variables check box. If TRUE, the Evolutionary Solver will return immediately from a call to the **SolverSolve** function with a value of 18 if any of the variables do not have both lower and upper bounds defined. If FALSE, the Evolutionary Solver will attempt to solve the problem without bounds on all of the variables.

**StepThru** is a logical value corresponding to the Show Iteration Results check box. If TRUE, Solver pauses at each trial solution; if FALSE it does not. If you have supplied **SolverSolve** with a valid VBA function, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

**Scaling** is a logical value corresponding to the Use Automatic Scaling check box. If TRUE, then Solver rescales the objective and constraints internally to similar orders of magnitude. If FALSE, Solver uses values directly from the worksheet.

**AssumeNonNeg** is a logical value corresponding to the Assume Non-Negative check box. If TRUE, Solver supplies a lower bound of zero for all variables without explicit lower bounds in the Constraint list box. If FALSE, no action is taken.

**BypassReports** is a logical value corresponding to the Bypass Solver Reports check box. If TRUE, the Solver will skip preparing the information needed to create Solver Reports. If FALSE, the Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

**LocalSearch** is a number corresponding to the option button selected in the Local Search option group:

<b>LocalSearch</b>	<b>Local Search Strategy</b>
1	Randomized Local Search
2	Deterministic Pattern Search
3	Gradient Local Search
4	Automatic Choice

In the Premium Solver Platform V5.5 and later, a value of 4 selects the Automatic Choice option; this allows the Solver to choose a local search method automatically – Randomized Local Search, Gradient Local Search, or Linear Local Gradient Search, depending on the characteristics of the problem.

**FixNonSmooth** is a logical value corresponding to the Fix Nonsmooth Variables check box. If TRUE, the Solver will fix the non-smooth variables to their current values during each local search, and allow only smooth and linear variables to be varied. If FALSE, the Solver will allow all of the variables to be varied.

---

## SolverGRGGet

Returns GRG Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the GRG Solver is selected in the Solver Engine dropdown list.

*VBA Syntax*

**SolverGRGGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want. The following settings are specified in the GRG Solver Options dialog box.

TypeNum	Returns
1	The Max Time value (as a number in seconds)
2	The Iterations value (max number of iterations)
3	The Precision value (as a decimal number)
4	The Convergence value (as a decimal number)
5	TRUE if the Show Iteration Result check box is selected; FALSE otherwise
6	TRUE if the Use Automatic Scaling check box is selected; FALSE otherwise
7	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
8	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise
9	TRUE if the Recognize Linear Variables check box is selected; FALSE otherwise
10	A number corresponding to the type of Estimates: 1 = Tangent 2 = Quadratic
11	A number corresponding to the type of Derivatives: 1 = Forward 2 = Central
12	A number corresponding to the type of Search: 1 = Newton 2 = Conjugate
13	The Population Size value (as a decimal number)
14	The Random Seed value (as a decimal number)
15	TRUE if the Multistart Search check box is selected; FALSE otherwise
16	TRUE if the Topographic Search check box is selected; FALSE otherwise
17	TRUE if Require Bounds on Variables check box is selected; FALSE otherwise

**SheetName** is the name of a worksheet that contains the Solver problem for which you want information. If **SheetName** is omitted, it is assumed to be the active sheet.

---

## SolverGRGOptions

Equivalent to choosing Solver... from the Tools menu and then choosing the Options button in the Solver Parameters dialog box when the GRG Nonlinear Solver is selected in the Solver Engines dropdown list. Specifies options for the GRG Solver.

*VBA Syntax*

**SolverGRGOptions** (**MaxTime**:=, **Iterations**:=, **Precision**:=, **Convergence**:=, **PopulationSize**:=, **RandomSeed**:=, **StepThru**:=, **Scaling**:=, **AssumeNonNeg**:=, **BypassReports**:=, **RecognizeLinear**:=, **MultiStart**:=, **TopoSearch**:=, **RequireBounds**:=, **Estimates**:=, **Derivatives**:=, **SearchOption**:=)

The arguments correspond to the options in the Solver Options dialog box. If an argument is omitted, the Solver maintains the current setting for that option. If any of the arguments are of the wrong type, the function returns the #N/A error value. If all arguments are of the correct type, but an argument has an invalid value, the function returns a positive integer corresponding to its position. A zero return value indicates that all options were accepted.

**MaxTime** must be an integer greater than zero. It corresponds to the Max Time edit box.

**Iterations** must be an integer greater than zero. It corresponds to the Iterations edit box.

**Precision** must be a number between zero and one, but not equal to zero or one. It corresponds to the Precision edit box.

**Convergence** is a number between zero and one, but not equal to zero or one. It corresponds to the Convergence box.

**PopulationSize** must be an integer greater than or equal to zero. It corresponds to the Population Size edit box.

**RandomSeed** must be an integer greater than zero. It corresponds to the Random Seed edit box.

**StepThru** is a logical value corresponding to the Show Iteration Results check box. If TRUE, Solver pauses at each trial solution; if FALSE it does not. If you have supplied **SolverSolve** with a valid VBA function, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

**Scaling** is a logical value corresponding to the Use Automatic Scaling check box. If TRUE, then Solver rescales the objective and constraints internally to similar orders of magnitude. If FALSE, Solver uses values directly from the worksheet.

**AssumeNonNeg** is a logical value corresponding to the Assume Non-Negative check box. If TRUE, Solver supplies a lower bound of zero for all variables without explicit lower bounds in the Constraint list box. If FALSE, no action is taken.

**BypassReports** is a logical value corresponding to the Bypass Solver Reports check box. If TRUE, the Solver will skip preparing the information needed to create Solver Reports. If FALSE, the Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

**RecognizeLinear** is a logical value corresponding to the Recognize Linear Variables check box. If TRUE, the Solver will recognize variables whose partial derivatives are not changing during the solution process, and assume that they occur linearly in the problem. If FALSE, the Solver will not make any assumptions about such variables. See the chapter “Solver Options” for a further discussion of this option.

**MultiStart** is a logical value corresponding to the Multistart Search check box. If TRUE, the Solver will use Multistart Search, in conjunction with the GRG Solver, to seek a globally optimal solution. If FALSE, the GRG Solver alone will be used to search for a locally optimal solution.

**TopoSearch** is a logical value corresponding to the Topographic Search check box. If TRUE, and if Multistart Search is selected, the Solver will construct a topography from the randomly sampled initial points, and use it to guide the search process.

**RequireBounds** is a logical value corresponding to the Require Bounds on Variables check box. If TRUE, the Solver will return immediately from a call to the **SolverSolve** function with a value of 18 if any of the variables do not have both lower and upper bounds defined. If FALSE, then Multistart Search (if selected) will attempt to find a globally optimal solution without bounds on all of the variables.

**Estimates** is the number 1 or 2 and corresponds to the Estimates option: 1 for Tangent and 2 for Quadratic.

**Derivatives** is the number 1 or 2 and corresponds to the Derivatives option: 1 for Forward and 2 for Central.

**SearchOption** is the number 1 or 2 and corresponds to the Search option: 1 for Newton and 2 for Conjugate.

---

## SolverIGGet

Returns Interval Global Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the Interval Global Solver is selected in the Solver Engine dropdown list.

*VBA Syntax*

**SolverIGGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want. The following settings are specified in the Interval Global Solver Options dialog box.

TypeNum	Returns
1	The Max Time value (as a number in seconds)
2	The Iterations value (max number of iterations)
3	The Accuracy value (as a decimal number)
4	The Resolution value (as a decimal number)
5	The Max Time w/o Improvement value (as a decimal number)
6	TRUE if the Show Iteration Result check box is selected; FALSE otherwise
7	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
8	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise
9	TRUE if the Abs vs. Relative Stop check box is selected; FALSE otherwise
10	TRUE if the Assume Stationary check box is selected; FALSE otherwise



11	A number corresponding to the type of Method: 1 = Classic Interval 2 = Linear Enclosure
12	TRUE if the Second Order check box is selected; FALSE otherwise
13	TRUE if the LP Test check box is selected; FALSE otherwise
14	TRUE if the LP Phase II check box is selected; FALSE otherwise

---

## SolverIGOptions

Equivalent to choosing Solver... from the Tools menu and then choosing the Options button in the Solver Parameters dialog box when the Interval Global Solver is selected in the Solver Engines dropdown list. Specifies options for the Interval Global Solver.

*VBA Syntax*

**SolverIGOptions (MaxTime:=, Iterations:=, Accuracy:=, Resolution:=, MaxTimeNoImp:=, StepThru:=, AssumeNonNeg:=, BypassReports:=, AbsRelStop:=, AssumeStationary:=, Method:=, SecondOrder:=, LPTest:=, LPPhaseII:=)**

The arguments correspond to the options in the Solver Options dialog box. If an argument is omitted, the Solver maintains the current setting for that option. If any of the arguments are of the wrong type, the function returns the #N/A error value. If all arguments are of the correct type, but an argument has an invalid value, the function returns a positive integer corresponding to its position. A zero return value indicates that all options were accepted.

**MaxTime** must be an integer greater than zero. It corresponds to the Max Time edit box.

**Iterations** must be an integer greater than zero. It corresponds to the Iterations edit box.

**Accuracy** must be a number between zero and one, but not equal to zero or one. It corresponds to the Accuracy edit box.

**Resolution** is a number greater than zero. It corresponds to the Resolution box.

**MaxTimeNoImp** is a number corresponding to the Max Time w/o Improvement edit box. This argument determines when the Interval Global Solver will stop with the message "Solver cannot improve the current solution."

**StepThru** is a logical value corresponding to the Show Iteration Results check box. If TRUE, Solver pauses at each trial solution; if FALSE it does not. If you have supplied **SolverSolve** with a valid VBA function, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

**AssumeNonNeg** is a logical value corresponding to the Assume Non-Negative check box. If TRUE, Solver supplies a lower bound of zero for all variables without explicit lower bounds in the Constraint list box. If FALSE, no action is taken.

**BypassReports** is a logical value corresponding to the Bypass Solver Reports check box. If TRUE, the Solver will skip preparing the information needed to create Solver Reports. If FALSE, the Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

**AbsRelStop** is a logical value corresponding to the Abs vs. Relative Stop check box. If TRUE, the Solver will use the absolute difference when comparing the current solution's objective to the best bound. If FALSE, the Solver will use the relative difference when making the comparison.

**AssumeStationary** is a logical value corresponding to the Assume Stationary check box. If TRUE, the Solver will assume that the optimal solution is a stationary point and is not at a decision variable bound. If FALSE, the Solver will search for optimal solutions at all points, including those where variables are at their bounds.

**Method** is the number 1 or 2 and corresponds to the Method option: 1 for Classic Interval and 2 for Linear Enclosure.

**SecondOrder** is a logical value corresponding to the Second Order check box. This option is used only if the Method option is 1. If TRUE, the Solver will use second order (Interval Newton) methods. If FALSE, the Solver will use only first order methods.

**LPTest** is a logical value corresponding to the LP Test check box. This option is used only if the Method option is 2. If TRUE, the Solver will use a Simplex method Phase I test to eliminate boxes that contain no feasible solutions. If FALSE, the Solver will not use this test.

**LPPhaseII** is a logical value corresponding to the LP Phase II check box. If TRUE, the Solver will use a Simplex method Phase II procedure to seek an improved bound on the objective in a box. If FALSE, the Solver will not use this procedure.

---

## SolverIntGet

Returns integer (Branch & Bound) option settings for the current Solver problem on the specified sheet. These settings are entered on the Integer Options dialog tab for any of the Solver engines.

*VBA Syntax*

**SolverIntGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want. The following settings are specified on the Integer Options dialog tab box.

TypeNum	Returns
1	The Max Subproblems value (as a decimal number)
2	The Max Integer Sols value (as a decimal number)
3	The Integer Tolerance value (as a decimal number)
4	The Integer Cutoff value (as a decimal number)
5	TRUE if the Solve Without Integer Constraints check box is selected; FALSE otherwise
6	TRUE if the Probing / Feasibility check box is selected; FALSE otherwise
7	TRUE if the Bounds Improvement check box is selected; FALSE otherwise
8	TRUE if the Optimality Fixing check box is selected; FALSE otherwise.
9	TRUE if the Primal Heuristic check box is selected; FALSE otherwise.

10	TRUE if the Use Dual Simplex for Subproblems check box is selected; FALSE otherwise.
11	The Gomory Cuts value (as a decimal number)
12	The Gomory Passes value (as a decimal number)
13	The Knapsack Cuts value (as a decimal number)
14	The Knapsack Passes value (as a decimal number)
15	The Max Cut Passes at Root value (as a decimal number)
16	The Max Cut Passes in Tree value (as a decimal number)
17	TRUE if the Use Strong Branching check box is selected; FALSE otherwise.
18	TRUE if the Lift and Cover (Cuts) check box is selected; FALSE otherwise.
19	TRUE if the Rounding (Cuts) check box is selected; FALSE otherwise.
20	TRUE if the Knapsack (Cuts) check box is selected; FALSE otherwise.
21	TRUE if the Gomory (Cuts) check box is selected; FALSE otherwise.
22	TRUE if the Probing (Cuts) check box is selected; FALSE otherwise.
23	TRUE if the Odd Hole (Cuts) check box is selected; FALSE otherwise.
24	TRUE if the Clique (Cuts) check box is selected; FALSE otherwise.
25	TRUE if the Rounding Heuristic check box is selected; FALSE otherwise.
26	TRUE if the Local Search Heuristic check box is selected; FALSE otherwise.
27	TRUE if the Flow Cover (Cuts) check box is selected; FALSE otherwise.
28	TRUE if the Mixed Integer Rounding (Cuts) check box is selected; FALSE otherwise.
29	TRUE if the Two Mixed Integer Rounding (Cuts) check box is selected; FALSE otherwise.
30	TRUE if the Reduce and Split check box is selected; FALSE otherwise
31	TRUE if the Special Ordered Sets check box is selected; FALSE otherwise

The return value for **TypeNum** = 6 through 14 is supported only for the Simplex LP Solver. The return value for **TypeNum** = 15 through 31 is supported only for the LP/Quadratic Solver. **SheetName** is the name of a worksheet that contains the Solver problem for which you want information. If **SheetName** is omitted, it is assumed to be the active sheet.

---

## SolverIntOptions

Equivalent to choosing Solver... from the Tools menu, choosing the Options button in the Solver Parameters dialog box, then choosing the Integer Options button in the Solver Options dialog. Specifies options for the integer (Branch & Bound) Solver.

*VBA Syntax*

**SolverIntOptions** (**MaxSubproblems**:=, **MaxIntegerSols**:=, **IntTolerance**:=, **IntCutoff**:=, **SolveWithout**:=, **UseDual**:=, **ProbingFeasibility**:=, **BoundsImprovement**:=, **OptimalityFixing**:=, **VariableReordering**:=, **UsePrimalHeuristic**:=, **MaxGomoryCuts**:=, **GomoryPasses**:=, **MaxKnapsackCuts**:=, **KnapsackPasses**:=; **MaxRootCutPasses**:=, **MaxTreeCutPasses**:=, **StrongBranching**:=, **LiftAndCoverCuts**:=, **RoundingCuts**:=, **KnapsackCuts**:=, **GomoryCuts**:=, **ProbingCuts**:=, **OddHoleCuts**:=, **MirCuts**:=, **TwoMirCuts**:=, **CliqueCuts**:=, **FlowCoverCuts**:=, **RedSplitCuts**:=, **SOSCuts**:=, **RoundingHeur**:=, **LocalHeur**:=)

The arguments correspond to the options on the Integer Options dialog tab. The first five options are common to both the Simplex LP and LP/Quadratic Solvers; the next ten options are specific to the Simplex LP Solver; and the remaining options are specific to the LP/Quadratic Solver. If an argument is omitted, the Solver maintains the current setting for that option. If any of the arguments are of the wrong type, the function returns the #N/A error value. If all arguments are of the correct type, but an argument has an invalid value, the function returns a positive integer corresponding to its position. A zero return value indicates that all options were accepted.

**MaxSubproblems** must be an integer greater than zero. It corresponds to the Max Subproblems edit box.

**MaxIntegerSols** must be an integer greater than zero. It corresponds to the Max Integer Sols (Solutions) edit box.

**IntTolerance** is a number between zero and one, corresponding to the Tolerance edit box.

**IntCutoff** is a number (any value is possible) corresponding to the Integer Cutoff edit box.

**SolveWithout** is a logical value corresponding to the Solve Without Integer Constraints check box. If TRUE, the Solver ignores any integer constraints and solves the “relaxation” of the mixed-integer programming problem. If FALSE, the Solver uses the integer constraints in solving the problem.

**UseDual** is a logical value corresponding to the Use Dual Simplex for Subproblems check box. If TRUE, the Solver uses the Dual Simplex method, starting from an advanced basis, to solve the subproblems generated by the Branch & Bound method. If FALSE, the Solver uses the Primal Simplex method to solve the subproblems.

**ProbingFeasibility** is a logical value corresponding to the Probing / Feasibility check box. If TRUE, the Solver attempts to derive settings for binary integer variables, and implications for feasibility of the subproblem, from the subproblem's bounds on binary integer variables. If FALSE, the Solver does not employ these strategies.

**BoundsImprovement** is a logical value corresponding to the Bounds Improvement check box. If TRUE, the Solver attempts to tighten the bounds of non-binary integer variables, based on the initial or derived settings of binary integer variables in the subproblem. If FALSE, the Solver does not employ this strategy.

**OptimalityFixing** is a logical value corresponding to the Optimality Fixing check box. If TRUE, the Solver attempts to fix the values of binary integer variables based on their coefficients in the objective function and constraints, and on the initial or derived settings of other binary integer variables. If FALSE, the Solver does not employ this strategy.

**VariableReordering** is a logical value corresponding to the Variable Reordering check box. In Version 5 of the Premium Solver products, this option is no longer used and its value is ignored.

**UsePrimalHeuristic** is a logical value corresponding to the Primal Heuristic check box. If TRUE, the Solver uses heuristic methods to attempt to discover an integer feasible solution at the beginning of the Branch & Bound process. If FALSE, the Solver does not employ this strategy.

**MaxGomoryCuts** must be an integer greater than or equal to zero. It corresponds to the Gomory Cuts edit box.

**GomoryPasses** must be an integer greater than or equal to zero. It corresponds to the Gomory Passes edit box.

**MaxKnapsackCuts** must be an integer greater than or equal to zero. It corresponds to the Knapsack Cuts edit box.

**KnapsackPasses** must be an integer greater than or equal to zero. It corresponds to the Knapsack Passes edit box.

**MaxRootCutPasses** must be an integer greater than or equal to zero. It corresponds to the Max Root Cut Passes edit box.

**MaxTreeCutPasses** must be an integer greater than or equal to zero. It corresponds to the Max Tree Cut Passes edit box.

**StrongBranching** is a logical value corresponding to the Use Strong Branching check box. If TRUE, If FALSE, no action is taken.

**LiftAndCoverCuts** is a logical value corresponding to the Lift and Cover check box. If TRUE, Lift and Cover cuts are generated. If FALSE, no cuts of this type are generated.

**RoundingCuts** is a logical value corresponding to the Lift and Cover check box. If TRUE, Rounding cuts are generated. If FALSE, no cuts of this type are generated.

**KnapsackCuts** is a logical value corresponding to the Knapsack check box. If TRUE, Knapsack cuts are generated. If FALSE, no cuts of this type are generated.

**GomoryCuts** is a logical value corresponding to the Gomory check box. If TRUE, Gomory cuts are generated. If FALSE, no cuts of this type are generated.

**ProbingCuts** is a logical value corresponding to the Probing check box. If TRUE, Probing cuts are generated. If FALSE, no cuts of this type are generated.

**OddHoleCuts** is a logical value corresponding to the Odd Hole check box. If TRUE, Odd Hole cuts are generated. If FALSE, no cuts of this type are generated.

**MirCuts** is a logical value corresponding to the Mixed Integer Rounding check box. If TRUE, Mixed Integer Rounding cuts are generated. If FALSE, no cuts of this type are generated.

**TwoMirCuts** is a logical value corresponding to the Two Mixed Integer Rounding check box. If TRUE, Two Mixed Integer Rounding cuts are generated. If FALSE, no cuts of this type are generated.

**CliqueCuts** is a logical value corresponding to the Clique check box. If TRUE, Clique cuts are generated. If FALSE, no cuts of this type are generated.

**FlowCoverCuts** is a logical value corresponding to the Flow Cover check box. If TRUE, Flow Cover cuts are generated. If FALSE, no cuts of this type are generated.

**RedSplitCuts** is a logical value corresponding to the Reduce and Split check box. If TRUE, Reduce and Split cuts (variants of Gomory cuts) will be generated.

**SOSCuts** is a logical value corresponding to the Special Ordered Sets check box. If TRUE, cuts for Special Ordered Sets will be generated.

**LocalHeur** is a logical value corresponding to the Local Search Heuristic check box. If TRUE, the Local Search Heuristic is used. If FALSE, no action is taken.

**RoundingHeur** is a logical value corresponding to the Rounding Heuristic check box. If TRUE, the Rounding Heuristic is used. If FALSE, no action is taken.

---

## SolverLimGet

Returns Limit Option settings for the Evolutionary Solver problem (if any) defined on the specified sheet. These settings are entered on the Limit Options dialog tab for the Evolutionary Solver.

*VBA Syntax*

**SolverLimGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want. The following settings are specified on the Limit Options dialog tab.

TypeNum	Returns
1	The Max Subproblems value (as a decimal number)
2	The Max Feasible Sols value (as a decimal number)
3	The Tolerance value (as a decimal number)
4	The Max Time w/o Improvement value (as a decimal number)
5	TRUE if the Solve Without Integer Constraints check box is selected; FALSE otherwise

**SheetName** is the name of a worksheet that contains the Solver problem for which you want information. If **SheetName** is omitted, it is assumed to be the active sheet.

---

## SolverLimOptions

Equivalent to choosing Solver... from the Tools menu, choosing the Options button in the Solver Parameters dialog box when the Evolutionary Solver is selected in the Solver Engine dropdown list, then choosing the Limit Options button in the Solver Options dialog. Specifies Limit Options for the Evolutionary Solver.

*VBA Syntax*

**SolverLimOptions (MaxSubproblems:=, MaxFeasibleSols:=, Tolerance:=, MaxTimeNoImp:=, SolveWithout:=)**

The arguments correspond to the options on the Limit Options dialog tab. If an argument is omitted, the Solver maintains the current setting for that option. If any of the arguments are of the wrong type, the function returns the #N/A error value. If all arguments are of the correct type, but an argument has an invalid value, the function

returns a positive integer corresponding to its position. A zero return value indicates that all options were accepted.

**MaxSubproblems** must be an integer greater than zero. It corresponds to the Max Subproblems edit box.

**MaxFeasibleSols** must be an integer greater than zero. It corresponds to the Max Feasible Sols (Solutions) edit box.

**Tolerance** is a number between zero and one, corresponding to the Tolerance edit box. This argument works in conjunction with the MaxTimeNoImp argument below.

**MaxTimeNoImp** is a number corresponding to the Max Time w/o Improvement edit box. This argument works in conjunction with the Tolerance argument above to determine when the Evolutionary Solver will stop with the message “Solver cannot improve the current solution.”

**SolveWithout** is a logical value corresponding to the Solve Without Integer Constraints check box. If TRUE, the Evolutionary Solver ignores any integer constraints and solves the “relaxation” of the problem. If FALSE, the Solver uses the integer constraints in solving the problem.

---

## SolverLPGet

Returns Simplex LP or LP/Quadratic Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the Simplex LP or LP/Quadratic Solver is selected in the Solver Engine dropdown list.

*VBA Syntax*

**SolverLPGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want. The following settings are specified in the Simplex LP or LP/Quadratic Solver Options dialog box.

TypeNum	Returns
1	The Max Time value (as a number in seconds)
2	The Iterations value (max number of iterations)
3	The Precision value (as a decimal number)
4	The Simplex LP Pivot Tolerance (as a decimal number)
5	The Simplex LP Reduced Cost Tolerance (as a decimal number)
6	TRUE if the Show Iteration Result check box is selected; FALSE otherwise
7	TRUE if the Use Automatic Scaling check box is selected; FALSE otherwise
8	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
9	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise.
10	A number corresponding to the Derivatives group selection: 1 = Forward 2 = Central
11	The LP/Quadratic Primal Tolerance (as a decimal number)

- 12           The LP/Quadratic Dual Tolerance (as a decimal number)
- 13           TRUE if the Do Presolve check box is selected; FALSE otherwise.

The return value for **TypeNum** = 4 and 5 is supported only for the Simplex LP Solver. The return value for **TypeNum** = 10 through 13 is supported only for the LP/Quadratic Solver. **SheetName** is the name of a worksheet that contains the Solver problem for which you want information. If **SheetName** is omitted, it is assumed to be the active sheet.

---

## SolverLPOptions

Equivalent to choosing Solver... from the Tools menu and then choosing the Options button in the Solver Parameters dialog box when the Simplex LP or LP/Quadratic Solver is selected in the Solver Engine dropdown list. Specifies options for the Simplex LP and LP/Quadratic Solvers.

*VBA Syntax*

**SolverLPOptions (MaxTime:=, Iterations:=, Precision:=, PivotTol:=, ReducedTol:=, StepThru:=, Scaling:=, AssumeNonNeg:=, BypassReports:=, Derivatives:=, PrimalTolerance:=, DualTolerance:=, Presolve:=)**

The arguments correspond to the options in the Solver Options dialog box. The PivotTol and ReducedTol options are available only for the Simplex LP Solver; the Derivatives, PrimalTolerance, DualTolerance, and Presolve options are available only for the LP/Quadratic Solver. If an argument is omitted, the Solver maintains the current setting for that option. If any of the arguments are of the wrong type, the function returns the #N/A error value. If all arguments are of the correct type, but an argument has an invalid value, the function returns a positive integer corresponding to its position. A zero return value indicates that all options were accepted.

**MaxTime** must be an integer greater than zero. It corresponds to the Max Time edit box.

**Iterations** must be an integer greater than zero. It corresponds to the Iterations edit box.

**Precision** must be a number between zero and one, but not equal to zero or one. It corresponds to the Precision edit box.

**PivotTol** is a number between zero and one, but not equal to zero or one. It corresponds to the Pivot Tolerance edit box for the Simplex LP Solver.

**ReducedTol** is a number between zero and one, but not equal to zero or one. It corresponds to the Reduced Cost Tolerance edit box for the Simplex LP Solver.

**StepThru** is a logical value corresponding to the Show Iteration Results check box. If TRUE, Solver pauses at each trial solution; if FALSE it does not. If you have supplied **SolverSolve** with a valid VBA function, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

**Scaling** is a logical value corresponding to the Use Automatic Scaling check box. If TRUE, then Solver rescales the objective and constraints internally to similar orders of magnitude. If FALSE, Solver uses values directly from the worksheet.

**AssumeNonNeg** is a logical value corresponding to the Assume Non-Negative check box. If TRUE, Solver supplies a lower bound of zero for all variables without explicit lower bounds in the Constraint list box. If FALSE, no action is taken.



**BypassReports** is a logical value corresponding to the Bypass Solver Reports check box. If TRUE, the Solver will skip preparing the information needed to create Solver Reports. If FALSE, the Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

**Derivatives** is the number 1 or 2 and corresponds to the Derivatives option group for the LP/Quadratic Solver: 1 for Forward and 2 for Central.

**PrimalTolerance** is a number between zero and one, but not equal to zero or one. It corresponds to the Primal Tolerance edit box for the LP/Quadratic Solver.

**DualTolerance** is a number between zero and one, but not equal to zero or one. It corresponds to the Dual Tolerance edit box for the LP/Quadratic Solver.

**Presolve** is a logical value corresponding to the Do Presolve check box. If TRUE, the Solver performs a presolve step before starting the Simplex method that detects singleton rows and columns, removes fixed variables and redundant constraints, and tightens bounds. If FALSE, no action is taken.

---

## SolverOkGet

Returns variable, constraint and objective selections and settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Parameters dialog.

*VBA Syntax*

**SolverOkGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want:

TypeNum	Returns
1	The reference in the Set Cell box, or the #N/A error value if Solver has not been used on the active document
2	A number corresponding to the Equal To option 1 = Max 2 = Min 3 = Value Of
3	The value in the Value Of box
4	The reference in the Changing Cells box (in the Premium Solvers, only the first entry in the Variables list box)
5	The number of entries in the Constraints list box
6	An array of the left hand sides of the constraints as text
7	An array of numbers corresponding to the relations between the left and right hand sides of the constraints: 1 = <= 2 = = 3 = >= 4 = int 5 = bin 6 = dif 7 = soc 8 = src
8	An array of the right hand sides of the constraints as text
9	An array of the entries in the Variables list box as text

- 10        A number corresponding to the Solver engine dropdown list for the currently selected Solver engine:
- 1    = Nonlinear GRG Solver
  - 2    = Simplex or LP/Quadratic Solver
  - 3    = Evolutionary Solver
  - 4    = Interval Global Solver
  - 5    = SOCP Barrier Solver
- In the Premium Solver Platform, other values may be returned for field-installable Solver engines
- 11        A string identifying the currently selected Solver engine:
- "Standard GRG Nonlinear"    = Nonlinear GRG Solver
  - "Standard Simplex LP"        = Simplex LP Solver
  - "Standard LP/Quadratic"      = LP/Quadratic Solver
  - "Standard Evolutionary"      = Evolutionary Solver
  - "Standard Interval Global"   = Interval Global Solver
  - "Standard SOCP Barrier"      = SOCP Barrier Solver
  - "KNITRO Solver"              = KNITRO Solver
  - "Large-Scale GRG Solver"     = Large-Scale GRG Solver
  - "Large-Scale LP Solver"      = Large-Scale LP Solver
  - "Large-Scale SQP Solver"     = Large-Scale SQP Solver
  - "MOSEK Solver Engine"        = MOSEK Solver Engine
  - "OptQuest Solver"            = OptQuest Solver
  - "XPRESS Solver Engine"       = XPRESS Solver Engine
- 12        An array of strings corresponding to the comments associated with each block of constraints
- 13        An array of logical values corresponding to the Report check box associated with each block of constraints (TRUE if the box is checked, FALSE otherwise); no longer used in V7.0
- 14        An array of strings corresponding to the comments associated with each block of variables
- 15        An array of logical values corresponding to the Report check box associated with each block of variables (TRUE if the box is checked, FALSE otherwise); no longer used in V7.0

**SheetName** is the name of a worksheet that contains the Solver problem for which you want information. If **SheetName** is omitted, it is assumed to be the active sheet.

---

## SolverSizeGet

Returns statistics about the size of the currently defined Solver problem, and the problem size limits supported by the currently selected Solver engine. The following settings are “read-only” and appear on the Problem tab in the Solver Options dialog for each Solver engine.

*VBA Syntax*

**SolverSizeGet (TypeNum:=, SheetName:=)**

**TypeNum** is a number specifying the type of information you want:

TypeNum	Returns
1	The number of decision variables in the current problem
2	The number of constraints in the current problem
3	The number of variable bounds in the current problem
4	The number of integer variables in the current problem
5	The maximum number of decision variables supported by the currently selected Solver engine

- 6           The maximum number of constraints supported by the  
            currently selected Solver engine
- 7           The maximum number of variable bounds supported by the  
            currently selected Solver engine
- 8           The maximum number of integer variables supported by the  
            currently selected Solver engine

**SheetName** is the name of a worksheet that contains the Solver problem for which you want information. If **SheetName** is omitted, it is assumed to be the active sheet.



# Index

## A

ABS function ..... 64, 116  
 absolute cell references..... 85, 135  
 Absolute vs. Relative Stop option..... 192  
 academic textbooks..... 18  
 Accuracy option..... 169, 191  
 Active Only option..... 118  
 active worksheet ..... 118  
 Add Constraint dialog ..... 83  
 Add Variable Cells dialog ..... 88  
 add-in function..... 82  
 advanced basis ..... 204  
 Albright, S. Christian ..... 38  
 algebraic form..... 62, 63, 81, 82, 91, 136  
 all real solutions..... 19, 28, 58, 73, 122, 238  
 Allan Waren..... 39  
 alldifferent constraint 17, 29, 35, 59, 72, 76, 89, 90, 161,  
 196, 200, 203, 210  
 Analysis Toolkit..... 24  
 Analyzer option group ..... 115  
 Analyzing and Solving Models ..... 101  
 Answer Report ..... 17, 219, 226, 227, 276  
 array form ..... 135  
 array form of DOTPRODUCT ..... 138  
 array formulas ..... 24, 36, 135, 136, 137  
 Assume Linear Model check box..... 68, 156, 177  
 Assume Non-Negative option ..... 57, 90, 178  
 Assume Stationary option ..... 192  
 Automatic Choice ..... 115  
 automatic differentiation.. 19, 26, 34, 66, 121, 135, 170,  
 182, 186, 187  
 automatic transformation ..... 25

## B

Baker, Kenneth ..... 38, 126  
 balance constraints..... 127  
 Barrier method ..... 71  
 basic variable ..... 186

basis selection ..... 22  
 Bayesian test ..... 72, 160, 166  
 best bound ..... 202  
 Best Values ..... 234  
 BFGS method (Search) ..... 187  
 binary integer variable..... 29, 59, 90, 127, 130, 131, 132,  
 133  
 binding constraints ..... 187, 230  
 biological organisms ..... 74  
 books about optimization ..... 37  
 Bounds Improvement option..... 205  
 bounds on the variables.. 57, 71, 90, 129, 161, 189, 195,  
 196, 228  
 boxes ..... 72, 192  
 Branch & Bound ..... 75, 159, 166, 169, 202, 204, 206  
 Branch & Bound method..... 209  
 Building Large-Scale Models..... 125  
 Building Solver Models ..... 81  
 built-in functions ..... 123  
 Bypass Solver Reports option ..... 179, 181, 221

## C

candidate solutions ..... 195  
 CEILING function ..... 64  
 Central choice (Derivatives)..... 182, 187  
 Changing Cells ..... 54  
 Check Model button..... 50, 62, 103, 105, 106, 107, 109,  
 111, 112, 121, 122, 158, 164, 289  
 CHOOSE function ..... 27, 64  
 Christian Albright..... 38  
 circular reference..... 159  
 classic interval methods ..... 72  
 Classic Interval option..... 193  
 Cliff Ragsdale..... 38  
 clique..... 205  
 Clique Cuts option..... 212  
 CliqueCuts SDK parameter ..... 212  
 clustering ..... 71, 189  
 coefficient..... 61, 62, 139, 230  
 Comment edit box ..... 223  
 Comments in reports ..... 227  
 concave function ..... 60, 64  
 conflicting constraints ..... 89  
**conic optimization** ..... 24, 53  
 Conjugate choice (Search) ..... 187  
 conjugate gradient method ..... 187  
 constant derivative ..... 66  
 constant right hand sides ..... 54, 59, 90  
 constraint handling methods..... 198  
 constraint left hand side... 54, 88, 92, 136, 176, 229, 273  
 constraint propagation..... 72, 193  
 constraint right hand side .54, 85, 89, 92, 136, 176, 227,  
 229, 273  
 constraints ..... 54

adding .....	129
balance .....	127
conflicting .....	89
either-or .....	132
equalities .....	57
fixed-charge .....	131
inequalities .....	57
more about .....	57
overlapping .....	89
ratio .....	130
Constraints button .....	88
contiguous cells.....	85, 87
Continue button.....	150, 175, 179, 285
Controlling the Solver's operation .....	241, 271
convergence in probability.....	72
Convergence option .....	153, 165, 172, 177, 185, 195
Convergence SDK parameter.....	177, 185, 191, 195
convex function.....	32, 56, 58, 59, 60, 61, 64, 123
convex objective .....	56
<b>convex optimization</b> .....	24, 53, 58
convex problem.....	25, 29, 35, 55, 67, 69, 70, 105, 164
convex quadratic .....	63
convexity.....	60, 101
Convexity option.....	105, 107
convexity test .....	25, 105, 123, 164
COUNT function .....	64
crossover.....	74, 77, 195, 197, 199
custom result codes .....	152
customized user interface .....	34
cut generation.....	75, 206
cycling.....	166

## D

Dash Optimization .....	23
data sources.....	128
data warehouses .....	128
Dealing with poor scaling .....	163
decision variables.....	54
Decision Variables and Parameters.....	53
defined names .....	35, 81, 85, 93, 126, 159
definiteness .....	63
degeneracy .....	153
degeneracy, nonlinear .....	22
degenerate problem.....	166
Dense mode .....	117
dense representation.....	68, 136
Dependent % box.....	104, 118
dependent on the decision variables.....	60, 61
dependents analysis.....	114
Dependents Report.....	219, 233
derivative .....	65, 116
constant .....	66
first.....	65
second .....	66, 69

derivative evaluation .....	26
Derivatives .....	64, 187
for Quadratic Solver .....	182
Derivatives SDK parameter.....	187, 196
Desired Model option group .....	106
deterministic methods.....	27, 28, 167
Deterministic Pattern Search .....	197
diagnosing problems.....	36
direct search method.....	27, 74
directed rounding .....	167
discontinuous function.....	23, 63, 64, 169, 170
distance filter.....	198
distance filter.....	74
diversity of population.....	171
DOTPRODUCT function.....	36, 62, 92, 137
dual feasible .....	204
Dual Simplex method.....	34, 75, 204
dual values.....	228
DualTolerance SDK parameter .....	181, 183, 184
During the Solution Process .....	150

## E

Efficiency of Constraint Forms .....	90
either-or constraint .....	132
Elements of Solver Models .....	53
email.....	37
Engines option group .....	111
equations .....	57
Error condition at cell address.....	157, 158
Error in model .....	159
error value .....	91, 156, 157
ESC key .....	
to stop Solver.....	175, 178
Estimates option .....	186, 187
Estimates SDK parameter.....	187
evaluation license .....	152
evolutionary algorithm .....	27, 73, 170
Evolutionary Solver.....	13, 17, 19, 27, 38, 57, 66, 71, 72, 73, 75, 115, 116, 129, 153, 154, 155, 157, 160, 161, 167, 169, 171, 199, 221, 222, 234, 276
Evolutionary Solver Stopping Conditions .....	170
EXAMPLES.XLS .....	91, 219
Excel .....	
Microsoft .....	41, 91, 135, 219, 272
Excel macro language .....	241, 271
Excel recalculation.....	119, 121, 124
Excel Scenario Manager.....	53, 178
external data sources .....	35
external name .....	159

## F

fast problem setup .....	17, 33, 34, 62, 83, 117, 134, 135, 136, 137
--------------------------	---

Fast Setup option .....	117
Feasibility Report.....	17, 30, 31, 33, 36, 149, 155, 203, 210, 220, 223, 226, 231, 277
Feasibility-Bounds Report .....	232, 277
Feasible and Optimal Solutions .....	55
feasible regions .....	70, 73
feasible solution .....	55, 67
field-installable Solver engines	18, 22, 34, 128, 152, 281
fill constraints .....	134
filtered local search.....	74
Filtered Local Search.....	198
finite differencing .....	26, 119, 120, 134, 182, 186, 187
finite precision .....	54, 67, 163, 176, 178
first derivative.....	65
first order methods.....	192
fitness.....	74, 153, 154, 171, 195, 200
Fix Nonsmooth Variables .....	194
Fix Nonsmooth Variables .....	116
Fix Nonsmooth Variables .....	197
fixed-charge constraints .....	131
floating point overflow .....	157
FLOOR function.....	64
Flow Cover Cuts option.....	212, 213
FlowCoverCuts SDK parameter .....	212, 213
formatting .....	81, 228
Forward choice (Derivatives) .....	182, 187
From Algebra to Spreadsheets .....	81
function	
built-in.....	123
unknown.....	157
unsupported.....	157
Function cannot be evaluated.....	162
functions	
of the variables.....	59, 88
Functions to Avoid - Discontinuities .....	64
Further Reading .....	37
Fylstra, Daniel.....	39

## G

Generalized Reduced Gradient .....	39, 70, 186
genetic algorithm .....	27, 73, 77, 170
global optimization	18, 19, 23, 26, 56, 57, 71, 160, 165, 166, 167, 190
globally optimal solution	17, 19, 26, 55, 56, 71, 72, 152, 160, 162, 166, 168, 171, 192
Goal Seek... command .....	54
goal seeking .....	39
Gomory Cuts.....	207
Gomory Cuts option.....	212
Gomory Passes.....	207
GomoryCuts SDK parameter .....	212
good model design.....	125
good solution .....	55, 57, 169, 170
good spreadsheet design .....	126

gradient .....	65, 120
Gradient Local.....	115
gradient-based method .....	27, 73, 74
GRG Method.....	70, 71
GRG Nonlinear Solver .....	13, 19, 26, 34, 56, 170
GRG Solver Options .....	194
GRG Solver stopping conditions.....	165, 185

## H

H.P. Williams.....	39
Help	
Solver .....	13, 36, 81, 135, 149, 166, 173, 219
Hesse, Richard .....	39
Hessian matrix.....	65, 66, 120, 122, 135, 187
Hessian of the Lagrangian .....	69
heuristic stopping rules.....	171
How to Use This Guide.....	35
hull consistency .....	72, 193
hybrid Evolutionary Solver .....	27, 198

## I

IF function.....	27, 64, 132, 169
If You Aren't Getting the Solution You Expect.....	149
Ignore nonsmooth variables .....	117
Implicit Non-Negativity Constraints .....	90
incumbent.....	202, 203
indefinite .....	63, 68
INDEX function.....	24
index sets.....	127
individual selection .....	85, 88, 137, 138
inequalities .....	57
inner solution.....	19, 28, 73, 227, 237
Insert Name Create.....	93, 126
Insert Name Define .....	93, 126, 137
Installation.....	35, 41
installation program .....	42
installing Solver engines .....	50
installing the software .....	42
INT function.....	64
IntCutoff SDK parameter .....	203, 209
integer constraint.....	55, 67, 75, 77, 88, 176
Integer Cutoff option.....	203, 209
Integer Options dialog.....	156, 201, 208, 213, 298, 300
integer programming .....	37, 39, 59, 67, 75, 202
Integer Tolerance option .....	159, 164, 202, 209
integer variables .....	76
Interior Point method .....	71
Interpreter.....	101, 117, 138, 141, 149, 161, 182, 187
Interpreting Dual Values .....	229
Interpreting Range Information.....	230
interval arithmetic .....	26, 122, 168
Interval Branch & Bound .....	72, 192
Interval Global Solver.....	56

Interval Global Solver.....	13, 18, 19, 28
Interval Global Solver.....	57
Interval Global Solver.....	57
Interval Global Solver.....	58
Interval Global Solver.....	64
Interval Global Solver.....	64
Interval Global Solver.....	71
Interval Global Solver.....	72
Interval Global Solver.....	115
Interval Global Solver.....	122
Interval Global Solver.....	129
Interval Global Solver.....	154
Interval Global Solver.....	161
Interval Global Solver.....	162
Interval Global Solver.....	167
Interval Global Solver.....	167
Interval Global Solver.....	192
Interval Global Solver.....	221
Interval Global Solver.....	222
Interval Global Solver.....	237
Interval Global Solver.....	285
Interval Global Solver Options .....	190
Interval Global Solver stopping conditions.....	168
interval gradient .....	192
<i>interval methods</i> .....	28
Interval Newton method.....	72, 193
intervals.....	19, 26, 72, 122, 162, 169, 192, 239
IntTolerance SDK parameter .....	177, 200, 202, 209
Irreducibly Infeasible System .....	231
iterations .....	175, 185
Iterations option .....	154, 175
Iterations SDK parameter .....	175

## J

Jacobian matrix.....	65, 66, 120, 122, 135, 186, 187
----------------------	---------------------------------

## K

Kenneth Baker .....	38, 126
Knapsack Cuts .....	207
Knapsack Cuts option .....	212
Knapsack Passes .....	207
KnapsackCuts SDK parameter.....	211
KNITRO Solver.....	23, 26, 71
Krawczyk operator.....	193
Kuhn-Tucker conditions .....	165

## L

Lagrange Multipliers.....	228
Large-Scale GRG Solver .....	22, 68, 70, 71, 76, 131
Large-Scale LP Solver .....	22, 68
Large-Scale SQP Solver .....	22, 32, 68, 70, 71, 77, 131
Lasdon, Leon .....	39

layout.....	35, 92
Layout and Formatting .....	92
left hand side	
constraints.....	54, 88, 92, 136, 176, 229, 273
Leon Lasdon.....	39
license code .....	43
license record .....	152
Lift and Cover Cuts option.....	211
LiftAndCoverCuts SDK parameter .....	211
Limit Options dialog .....	171, 194, 302
limitations	
global optimization.....	167
non-smooth optimization .....	169
smooth nonlinear optimization .....	164
Limits Report.....	17, 219, 226, 231, 276
Linear and Nonlinear Functions .....	59
Linear and Nonlinear Programming .....	67
linear enclosure form.....	73, 192
Linear Enclosure option .....	193
linear function .....	59, 61, 104, 130, 131, 154, 156, 233
Linear Local Gradient Search.....	197
linear model	
testing for.....	62
linear programming .....	22, 67, 82, 137, 230
solution .....	152
linear variable .....	104
linearity conditions not satisfied .....	150, 156, 177
Linearity Report .....	17, 30, 36, 150, 156, 220, 223, 226, 232, 276
linearity test .....	25, 163, 170
linearized local gradient method .....	74
Load Model .....	214
Local Search Heuristic option .....	213
local search.....	199
Local Search options .....	196, 287, 288
LocalHeur SDK parameter.....	213
locally optimal solution .....	55, 56, 70, 73, 160, 165, 166, 172, 192
Locally Versus Globally Optimal Solutions .....	165
LOOKUP function .....	27, 64
loss of diversity .....	171
Lotus 1-2-3 .....	216
LP Phase II option.....	194
LP Test option.....	193
LP/Quadratic Solver...13, 19, 56, 62, 68, 179, 180, 184,	229
LU decomposition .....	68

## M

macro language .....	241, 271
Macro Recorder.....	34, 36, 271
maintainable models.....	149
management science .....	18, 38
Markowitz .....	19



Markowitz refactorization.....	68
matrix factorization.....	68
Max Feasible Sols option.....	154, 160, 172, 194, 200
Max Feasible Solutions option .....	209
MAX function.....	64, 116
Max Integer Sols option.....	160, 202
Max Subproblems option... 33, 154, 160, 172, 194, 199, 201, 202, 208	
Max Time option .....	157, 175, 194
Max Time w/o Improvement option 154, 168, 171, 191, 200	
maximize the minimum.....	130
Maximum Cut Passes option.....	210
Maximum Values.....	234
MaxIntegerSols SDK parameter .....	200, 202, 208
MaxRootCutPasses SDK parameter .....	210
MaxSubProblems SDK parameter ....	199, 201, 207, 208
MaxTime SDK parameter.....	175
MaxTreeCutPasses SDK parameter.....	210
mean value form .....	192
Mean Values .....	234
memory.....	41, 68, 117, 129, 158
virtual.....	158
Merge Model function .....	280
merging Solver models .....	216
merit filter .....	74, 198
Method options group.....	192
Microsoft Excel 39, 41, 91, 92, 119, 123, 135, 138, 216, 221	
Microsoft Excel Help.....	13
MIN function .....	64, 116
minimize the maximum.....	130
Minimum Values .....	234
MirCuts SDK parameter .....	212
Mixed Integer Rounding Cuts option.....	212
mixed-integer programming.... 67, 75, 77, 200, 202, 209	
solution .....	152
MMULT function .....	36, 136
model analysis .....	18, 24, 33, 35
Model button .....	102
model diagnosis .....	24, 25, 32, 112, 122
model diagnosis exceptions .....	25, 35, 106
model size .....	125
model sparsity.....	101
model statistics.....	103, 108
modeling techniques .....	35, 130
MOSEK Solver.....	23, 26, 29, 59, 69, 114, 131
Multi-area not supported.....	159
multi-level single linkage.....	27, 71
multiple locally optimal solutions.....	57
multiple selection..... 33, 85, 86, 92, 135, 137, 138, 139	
multiple Solver models .....	215
multiple worksheets .....	127
multistart methods..... 19, 27, 56, 57, 71, 129, 160, 161, 165, 166, 167, 188, 190	

MultiStart SDK parameter .....	188
Munirpallam Venkataramanan .....	38
mutation .....	74, 77, 195, 196, 197, 199
Mutation Rate option.....	33, 153, 171, 172, 195

## N

natural selection .....	74
negative definite.....	63, 68
Newton choice (Search) .....	187
NLP (nonlinear programming problem).....	70
no feasible solution .....	155
nonbasic variable.....	186, 228, 229
non-convex function.....	32, 58, 59, 60
non-convex objective .....	56
non-convex problem.....25, 26, 27, 35, 55, 70, 71, 105, 164, 168	
non-convex quadratic .....	63
nondeterministic methods.....	27, 28, 72, 167
nonlinear function .....	59, 62, 63, 154, 186, 232
Nonlinear Gradient Search.....	197
nonlinear GRG Solver.....	39, 229, 283
nonlinear optimization.....	67
solution.....	152, 153
nonlinear problems.....	164
nonlinear programming .....	70
non-negativity.....	57, 90
non-smooth function....23, 57, 59, 64, 66, 116, 161, 169	
non-smooth optimization.....	17, 35, 67
non-smooth problems .....	73, 169
NSP (non-smooth optimization problem).....	67
numerical instability .....	67, 70
numerical tolerances.....	173

## O

Object Browser .....	36
objective function.....	54
Odd Hole Cuts option .....	212
OddHoleCuts SDK parameter .....	212
Office	
Microsoft.....	41
OFFSET function.....	24, 124
OLAP databases .....	128
operations research.....	18, 38
optimal solution.....	36, 55, 152, 202, 219, 230
Optimality conditions .....	165
Optimality Fixing option .....	205
optimization modeling hints .....	35, 127
Options	
global optimization.....	190
Solver .....	90, 149, 163, 179, 180, 182, 184, 190, 194
OptQuest Solver.....	23, 71, 73, 75, 129, 157, 161
orders of magnitude.....	163, 178, 282
Other Functions for Fast Problem Setup .....	136

Other Nonlinear Options.....	186
Outline Reports.....	223
outlining of reports.....	30, 81
Overlapping constraints .....	89

## P

Parameters of your model .....	53
partial derivative .....	65, 66, 119, 122, 134, 186, 187
permutation .....	17, 30, 76
piecewise-linear functions.....	134
Pivot Tolerance.....	179, 180
PivotTables .....	128
Polymorphic Spreadsheet Interpreter.....	18, 24, 34, 41, 119, 121, 135
poorly scaled models .....	32, 101, 129, 153, 163, 165, 166, 178
population of solutions.....	27, 73
Population Report ....	17, 30, 31, 36, 172, 220, 221, 222, 226, 234, 235, 276
Population Size option .....	33, 153, 171, 172, 190, 195
PopulationSize SDK parameter .....	190, 195
portfolio optimization .....	63, 139, 215, 224
positive definite.....	63, 68
Powell, Stephen .....	38, 126
Precision and Integer Constraints .....	176
Precision and Regular Constraints .....	176
Precision option .....	54, 165, 176, 194
Precision SDK parameter.....	176
Premium Solver .....	13, 17, 62
Premium Solver for Education.....	18, 38
Premium Solver Platform. ....	13, 18, 22, 24, 62, 66, 71, 72, 75, 101, 122, 128, 140, 161, 162, 163, 165, 166, 167, 170, 190, 204, 233, 237, 277, 281, 299, 304
Premium VBA Functions .....	272, 291
Preprocessing & Probing .....	34, 75
Presolve SDK parameter.....	181
primal feasible.....	204
primal heuristics.....	75
PrimalTolerance SDK parameter .....	181, 183, 184
Probing.....	205
Probing Cuts option .....	212
Probing/Feasibility option.....	204
ProbingCuts SDK parameter.....	212
problem size.....	32, 156
Problem Size and Numerical Stability .....	67
Problems with Poorly Scaled Models .....	163
progress reporting .....	17, 32
proving optimality.....	56
proving optimality.....	28
proving optimality.....	167
pseudocost branching.....	206
pseudocosts .....	75

## Q

QUADPRODUCT function.....	62, 139
quadratic approximation.....	131
Quadratic choice (Estimates).....	187
quadratic function.....	62, 139, 156
quadratic objective .....	19, 139, 182
quadratic programming.....	19, 22, 23, 67, 68, 135, 139, 229
solution .....	152
Quadratic Solver.....	136, 182
QUADTERM function .....	141
quasi-Newton method.....	27, 74, 187

## R

Ragsdale, Cliff.....	38
random choice .....	27, 28, 167
random number generator.....	190, 196
Random Seed option .....	33, 72, 73, 190, 196
Randomized Local Search.....	197
RandomSeed SDK parameter.....	190, 191, 196
Range Selector .....	87
ranges	
for dual values .....	228, 230
ratio constraint.....	130
readable models.....	91, 149
Reading model settings from VBA.....	272
Recognize Linear Variables option .....	34, 131, 186
RecognizeLinear SDK parameter .....	186
reduced cost fixing .....	75
Reduced Cost Tolerance.....	179, 180
Reduced Costs.....	180, 228, 230
Reduced Gradients .....	228
redundant constraints.....	154
referencing Solver functions in VBA .....	272
Relation dropdown list .....	90
relational databases .....	128
relative cell references.....	85, 135
relaxation.....	33
of integer problem .....	200, 202, 209
repair of a solution .....	74
replacing IF function .....	132
report outlining .....	30
Report Scaling Problems .....	123, 129, 224
Reports	
Solver .....	13, 179, 228, 230, 276
Reports list box .....	179, 221, 222, 223
Require Bounds on Variables option.....	129, 161, 189, 196
Require Smooth.....	198
Require Smooth option.....	116
RequireBounds SDK parameter .....	189, 196, 197
Resolution option .....	191
Restart button .....	150, 179, 201, 285
result codes.....	150, 151
custom .....	152

Return to Solver Parameters Dialog check box 151, 224  
 Richard Hesse ..... 39  
 right hand side  
     constant ..... 54, 59  
     constraints ..... 54, 85, 88, 92, 136, 176, 227, 229, 273  
 rigorous global optimization ..... 167  
 roots of equations ..... 238  
 ROUND function ..... 64  
 rounding and possible loss of solutions ..... 167  
 Rounding Cuts option ..... 211  
 Rounding Heuristic option ..... 213  
 RoundingCuts SDK parameter ..... 211  
 RoundingHeur SDK parameter ..... 213

## S

saddle point ..... 63  
 satisfied  
     constraints ..... 54, 58, 176, 227  
 Save Model ..... 214  
 scalable models ..... 91  
 scaling .. 32, 68, 123, 125, 129, 150, 163, 166, 176, 178, 224, 282, 293  
 Scaling Report ..... 30, 32, 36, 219, 224, 225  
 Scaling SDK parameter ..... 178, 180  
 scatter search ..... 23, 75  
 Scenario Manager ..... 53, 178  
 Search option ..... 186, 187  
 SearchOption SDK parameter ..... 187  
 second derivative ..... 64, 66  
 second order cone constraint . 29, 34, 35, 55, 56, 58, 69, 91  
 second order cone programming.. 18, 20, 28, 29, 56, 58, 69, 152  
 second order methods ..... 72  
 Second Order option ..... 193  
 Selecting the Reports ..... 221  
 selection process ..... 74, 195, 197, 199  
 semi-definite ..... 63, 68  
 seminars ..... 38  
 sensitivity analysis ..... 231  
 Sensitivity Report ..... 17, 36, 219, 226, 228, 276  
 Sequential Quadratic Programming ..... 23, 70  
 Set Cell ..... 54  
 Set Cell Value of ..... 54  
 Set Cell values do not converge ..... 154  
 Setting Up a Model ..... 81  
 sexual reproduction ..... 74  
 Shadow Prices ..... 228, 230  
 Sharpe ..... 19  
 Show Iteration Results option ..... 149, 178  
 Show Trial Solution dialog ..... 150, 154, 155, 282, 284, 293, 295, 297, 304  
 SIGN function ..... 116  
 Simplex LP Solver ..... 34, 55

Simplex method ..... 28, 68, 74, 180  
 Simplex Solver Options ..... 179, 180, 184  
 slack  
     in constraints ..... 227  
 smooth function ..... 56, 64, 104  
 smooth variable ..... 104  
 SOCP Barrier Solver .. 13, 18, 19, 23, 26, 28, 29, 34, 56, 59, 69, 114, 131, 151, 182  
 SOCP Barrier Solver Options ..... 182  
 solution ..... 252  
 Solution time ..... 67  
 Solutions Report ..... 30, 36, 73, 221, 237, 238, 239  
 Solve With  
     Automatic ..... 114, 115  
     Convexity ..... 114  
     Gradients ..... 114, 115  
     No Action ..... 115  
     Structure ..... 114, 115  
 Solve With No Action ..... 117  
**Solve With option group** ..... 113  
 Solve Without Integer Constraints option . 200, 203, 210  
 Solver cannot improve the current solution ..... 153, 168, 171, 200  
 Solver converged in probability ..... 160  
 Solver could not find a feasible solution ..... 155  
 Solver dialogs ..... 83  
 Solver encountered an error computing derivatives .. 161  
 Solver encountered an error value ..... 156  
 Solver engine ..... 22  
 Solver engine compatibility ..... 42  
 Solver engine dropdown list ..... 22, 27, 84, 112  
 Solver engine size limits ..... 32, 156  
 Solver function return values ..... 272  
 Solver has converged to the current solution ..... 153, 171  
 Solver Model dialog .. 31, 32, 35, 62, 101, 233, 287, 289  
 Solver Model VBA Functions ..... 272, 286  
 Solver Models and Optimization ..... 53  
 Solver Options dialog .. 90, 149, 163, 173, 179, 180, 182, 184, 190, 194, 277, 278, 282, 291, 292, 294, 295, 296, 297, 303, 304  
 Solver Parameters dialog ..... 84, 278, 305  
 Solver Platform SDK ..... 21  
 Solver Reports ..... 13, 179, 228, 230, 276  
 Solver Result Message ..... 149, 151, 165  
 Solver Results dialog ..... 33, 85, 149, 150, 151, 179, 221, 223, 276, 284  
 SolverAdd (Form 1) ..... 273  
 SolverAdd (Form 2) ..... 274  
 SolverAdd function ..... 273, 280  
 SolverChange (Form 1) ..... 274  
 SolverChange (Form 2) ..... 274  
 SolverChange function ..... 273  
 SolverDelete (Form 1) ..... 275  
 SolverDelete (Form 2) ..... 275  
 SolverDelete function ..... 273

SolverDependents function .....	291
SolverEVGet function.....	291
SolverEVOptions function.....	292
SolverFinish function.....	276, 284
SolverFinishDialog function .....	277
SolverGet function.....	277
SolverGRGGet function.....	294
SolverGRGOptions function.....	295
SolverIGGet function.....	296
SolverIGOptions function .....	297
SolverIntGet function.....	298
SolverIntOptions function.....	300
SolverLimGet function .....	302
SolverLimOptions function.....	302
SolverLoad function.....	280, 283
SolverLPGet function .....	303
SolverLPOptions function.....	304
SolverModel function .....	287
SolverModelGet function.....	289
SolverOk function.....	17, 274, 280
SolverOkDialog function .....	281
SolverOkGet function .....	305
SolverOptions function .....	280, 282, 284
SolverReset function.....	283
SolverSave function.....	283
SolverSizeGet function .....	306
SolverSolve function.. 17, 150, 151, 155, 271, 272, 284, 285	
SolveWithout SDK parameter .. 200, 203, 204, 205, 210	
SOLVSAMP.XLS.....	13, 82, 85, 135
SOS constraint .....	205
sparse matrix .....	67, 92, 136
Sparse mode.....	117
Evolutionary Solver .....	118
KNITRO Solver.....	118
Sparse option .....	115, 117
sparsity .....	32, 101
special functions .....	116, 157, 162, 170
Special Ordered Set .....	205
Speed 17, 33, 34, 68, 122, 134, 135, 293, 295, 297, 305	
Spreadsheet Detective.....	127
spreadsheet formulas.....	35, 82
spreadsheet modeling hints .....	35, 126
SQP Method .....	70, 71
Standard Deviations.....	234
standard Excel Solver .....	16, 174
standard Excel Solver Help.....	36
Standard VBA Functions .....	272, 273
Stephen Powell .....	38, 126
Stop button.....	150, 179, 284, 285
Stopping conditions	
GRG Solver.....	165, 185
Interval Global Solver.....	168
Stopping Solver	
ESC key .....	175, 178, 284

StrongBranching SDK parameter.....	211
Structure Report .....	30, 31, 36, 62, 106
subproblem	
Branch & Bound.....	75, 203, 209
SUM function.....	135
SUMIF function .....	65
summation notation .....	136
SUMPRODUCT function .....	36, 61, 81, 92, 135
swapping .....	158
symbol table .....	121
system of equations .....	19, 28, 58, 238
system of inequalities .....	19, 28, 29, 237

## T

tabu search.....	23, 75
Tangent choice (Estimates) .....	187
technical support .....	36
testing for linearity .....	62
Tolerance option .....	153, 154, 164, 171, 177, 200
Tolerance Option and Integer Constraints.....	164
Too many adjustable cells .....	156
Too many constraints .....	156
Too many integer adjustable cells .....	156
Tools Macro Record New Macro.....	271
Tools References .....	272
topographic search .....	71, 189
Topographic Search option .....	189
TopoSearch SDK parameter.....	189, 192, 193
Total Cells box .....	104
Tour of New Features.....	24
Transferring models between spreadsheets .....	215
transformation of non-smooth functions.....	25
Transformation Report .....	30, 36
TRANPOSE function.....	136
Traveling Salesman Problem.....	17, 29, 59
trial license .....	22, 49, 50, 152
true global optimum .....	167
Two Mixed Integer Rounding Cuts option.....	212
TwoMirCuts SDK parameter .....	212

## U

unbounded objective value.....	67
undefined identifier .....	159
uninstalling.....	45
university use.....	18
unknown function.....	157
unproven solution.....	162, 169
unsupported Excel function.....	157
Use Automatic Scaling option... 163, 165, 176, 178, 224	
Use Dual Simplex for Subproblems option .....	204
Use Strong Branching option .....	211
user interface improvements.....	32
user-defined functions .....	24

Using Array Formulas.....	136
Using Defined Names .....	93
Using Integer Constraints .....	90, 91
Using Microsoft Excel Help .....	272
Using QUADPRODUCT.....	139
Using the Macro Recorder.....	271
Using the Premium Solver Platform .....	13
Using the Variables Button.....	87

## V

Value of edit box .....	54
Variable Cells list box .....	274
Variable Reordering option .....	205, 206
Variables and Multiple Selections .....	85
Variables button.....	33, 87, 88, 274
VBA Function Reference.....	272
Venkataramanan, Munirpallam.....	38
Visual Basic.....	34, 36, 241, 271

## W

Waren, Allan.....	39
Watson, John .....	39
Wayne Winston .....	38
What Is Not Possible .....	238
What You Need .....	41
Williams, H.P. ....	39
Winston, Wayne .....	38

## X

XPRESS Solver .....	23, 68, 130
Xpress <sup>MP</sup> .....	23

## Z

Ziena Optimization .....	23
--------------------------	----