Version 2025 Q1 for Excel, Matlab, Java, .NET, COM, Win/Linux

2025 Q1 Plug-in Solver Engines User Guide

For Analytic Solver Desktop, Analytic Solver Cloud and Solver SDK

Large-Scale **LP/QP Solver**
Large-Scale **GRG Solver**
Large-Scale **SQP Solver**
Knitro Solver
MOSEK Solver
Gurobi Solver
XPRESS Solver
OptQuest Solver

How to Order

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

Contents

Start Here: 2025 Q1 Essentials

Getting the Most from This User Guide

Installing Analytic Solver Cloud

Analytic Solver V2025 Q1 includes our next-generation offering, Analytic Solver Cloud – usable in the latest versions of desktop Excel for Windows and Macintosh, and in Excel for the Web (formerly Excel Online). Analytic Solver Cloud is divided into **two** add-ins that work closely together (since a JavaScript add-in currently can have only one Ribbon tab): the **Analytic Solver** add-in builds optimization and simulation models, and the **Data Science** add-in (formerly known as the Data Mining add-in) builds data science or forecasting models.

Both the Analytic Solver and Data Science add-ins support existing models created in previous versions of Analytic Solver. Your license for Analytic Solver will allow you to use Analytic Solver Desktop in desktop Excel or Analytic Solver Cloud in either desktop Excel (latest version) *or* Excel for the Web.

To use the Analytic Solver and Data Science add-ins, **you must first "insert" them** for use in your licensed copy of desktop Excel or Excel for the Web, while you are logged into your Office 365 account. Once you do this, the Analytic Solver and Data Science tabs will appear on the Ribbon in each new workbook you use. See the Analytic Solver User Guide for more information.

Installing the Software

In the past, to use the Solver Engines, you had to have Analytic Solver Platform, Risk Solver Platform, Premium Solver Platform, or Solver SDK installed. However, starting with V2017, all Large Scale Solver Engines for desktop Excel are now installed via the SolverSetup program, all Large Scale Solver Engines for the SDK are installed via the SDKSetup (for 32 bit) or SDKSetup64 (for 64 bit) program, and all Large Scale Solver Engines for Analytic Solver Cloud are installed via the Microsoft Store.

To install Analytic Solver Comprehensive or Analytic Solver Optimization and the Frontline Solver Engines to work with any supported version of desktop *Microsoft Excel*, simply run the program **SolverSetup.exe**, which installs all of the Solver and Engine programs, Help, User Guides, and example files in compressed form for both 32-bit and 64-bit Excel. SolverSetup.exe checks your system, detects what version of Office you are running (32-bit or 64-bit) and then downloads and runs the appropriate Setup program version. The chapter "Installation and Licensing" in the Analytic Solver User Guide covers installation step-by-step.

The same is true if you are using Solver SDK, simply run the program **SDKSetup.exe** (for 32-bit) or **SDKSetup64.exe** (for 64-bit), to install all of the Solver and Engine programs, Help, User Guides, and example files in compressed form for 32-bit or 64-bit SDK, respectively. The chapter "Installation and Licensing" in the Solver SDK User Guide covers installation step-by-step.

Upgrading from Earlier Versions

Solver Engines 2025 Q1 work with the Platform 2025 Q1 products. You can upgrade your Platform product as well as your external engine(s) all-at-once. Solver Engines 2025 Q1 are **upward compatible** with earlier versions and generally offer better performance – your existing applications should work as-is with the new Solver Engines.

Finding the Examples

Use Help – Examples on the Ribbon to open workbooks with a list of examples you can open by clicking hyperlinks. In Solver SDK, if you're using Microsoft Visual Studio, browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK\Examples, then choose your programming language (C, C#, C++ or VB.NET) and choose the example you'd like to open in Visual Studio. If you are using Visual Basic 6, Java or MATLAB, navigate to the installation folder (for example C:\Program Files\Frontline Systems\Analytic Solver SDK\SolverSDK \Examples).

Using Existing Models and Applications

In Analytic Solver Cloud or Desktop, just **open your existing workbook**, developed in any previous version of Analytic Solver Platform, Risk Solver Platform, Premium Solver Platform, Risk Solver, Premium Solver, or the standard Excel Solver. Your model should appear in the Task Pane; just click the Optimize or Simulate button.

In Solver SDK, **open your existing project or solution** and rebuild your application. Note that Solver SDK 2021 supports **Microsoft .NET V2017 or V2019.**

Using Existing VBA Macros

In Analytic Solver Comprehensive and Analytic Solver Optimization (Desktop only), **macros using the Object-Oriented API** such as Problem.Solver.Optimize should work as-is, provided that you use Tools References in the VBA Editor to set or change the reference to **Analytic Solver Platform 2025 Q1 Type Library**.

Standard Excel Solver macros such as SolverOK and SolverSolve should work as-is, provided that you use Tools References in the VBA Editor to set or change the reference to **Solver**. Note: You must use the Solver located in C:\Program Files\Microsoft Office X\Root\OfficeX\Library\Solver where X is the Microsoft Excel version number.

Choosing a Solver Engine

The chapter "Using the Solver Engines" provides guidance about choosing and using a Solver Engine appropriate for your problem, and describes the special capabilities of the Solver Engines (that you typically won't find in other optimization software). The Appendix "Solver Engine Methodologies" describes the technical methods used in many of the Solver Engines.

Getting and Interpreting Results

The chapter “Solver Result Messages” documents in detail the meaning of the Solver Result Messages – both standard and Solver Engine-specific messages. You’ll also find the integer codes returned by the **Problem.Solver.OptimizeStatus** property and the **SolverSolve** function for each message, which can be tested in your VBA or other programming language code.

Using Solver Engine Options

The chapter “Solver Engine Options” documents in detail the various options and parameters you can set for each plug-in Solver Engine, interactively in Analytic Solver Platform, in your code in VBA and in Solver SDK.

Programming the Solver Engines

The chapter “Programming the Solver Engines” describes how to select a Solver Engine and set its options and parameters, using VBA in Analytic Solver Desktop (only) or using another programming language with Solver SDK. The “legacy” SolverXxx functions for Solver Engine options are also documented here.

Using the Plug-in Solver Engines

Introduction

Thank you for using Frontline Systems' large-scale Solver Engines. These Solver Engines "plug into" Frontline's Desktop and Cloud products: Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK. They 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. They are available in both 32-bit and 64-bit versions.

Analytic Solver Comprehensive and Analytic Solver Optimization for both Desktop and Coud, are fully compatible upgrades for the Solver bundled with Microsoft Excel, which was developed by Frontline Systems for Microsoft. They include a wide spectrum of enhancements to the standard Excel Solver, plus five bundled Solver engines – the LP/Quadratic Solver, nonlinear GRG Solver, Evolutionary Solver, SOCP Barrier Solver, and the Interval Global Solver. Solver SDK is a powerful, flexible software development kit that makes the first four of these Solver engines available for use in your application written in C/C++ or C#, Visual Basic or VB.NET, MATLAB, Java, or another language.

Using Solver Engines with Microsoft Excel

When the Solver Engines are used in Microsoft Excel (Online or Desktop), they integrate seamlessly with Analytic Solver Comprehensive and Analytic Solver Optimization – to use one, you simply select the Solver Engine by name in the dropdown list in the Task Pane Engine tab. They produce reports as Excel worksheets, like the bundled Solver engines; they recognize common Solver options and provide their own options in the Task Pane Engine tab; and they can be controlled by Excel VBA code (Analytic Solver Desktop only) in your custom applications. With a free trial license, you can evaluate how well they perform on a challenging Solver model that you have developed.

Using Solver Engines with Solver SDK

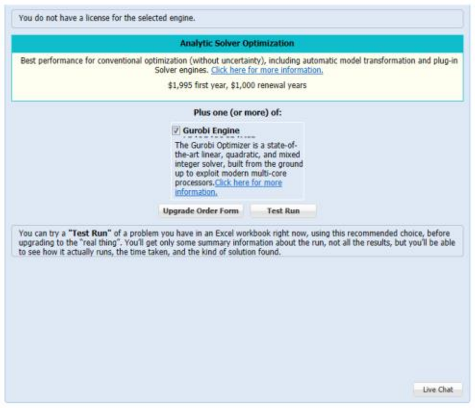
The Solver Engines also integrate seamlessly with Solver SDK, enabling you to build custom programs in C/C++ or C#, Visual Basic or VB.NET, MATLAB, Java, ASP or ASP.NET, that work with any Microsoft Win32/Win64, COM or .NET application. You can either build your model as an Excel workbook and load that workbook into your SDK-based application at runtime (without requiring Microsoft Excel), or you can write a program that implements your model, computing values (and optionally derivatives) for your objective function and constraints, given values for the decision variables.

Solver SDK offers both an easy to use procedural application programming interface (API), with functions that you call to define and solve an optimization problem, and an even easier to use, high-level object-oriented API, where you define objects such as a Problem, Solver, Model, Variable and Function, and set properties and call methods to solve your problem.

If you want to distribute your application, including Solver SDK and Frontline's Solver engines, to other users – in your own company or commercially to outside companies – or make your application available via an Intranet server, Web server or Web service, you can do so easily with a runtime license agreement from Frontline Systems. This makes available discounted runtime licenses for the SDK and Solver Engines you want to include with your application. Deployment is straightforward – typically involving just one to three files – and you can use your own license management system, or license management software provided by Frontline Systems. For more information, please contact Frontline Systems at (775) 831-0300 or by email at **info@solver.com**.

Choosing the Best Plug-in Solver Engine

The tables below give a high level overview of what each Engine is best suited for. If you are using desktop Analytic Solver, an easy way to determine the best engine choice is to select a specific external engine, from the Engine drop down menu on the Engine tab of the Solver Task Pane, for which you do not have a license. The Product Wizard will appear and recommend the selected engine, and allow you to solve your model using this engine. Once Solver has finished solving, you will have the option to purchase the product.


You do not have a license for the selected engine.

Analytic Solver Optimization

Best performance for conventional optimization (without uncertainty), including automatic model transformation and plug-in Solver engines. Click here for more information.

$1,995 first year, $1,000 renewal years

Plus one (or more) of:

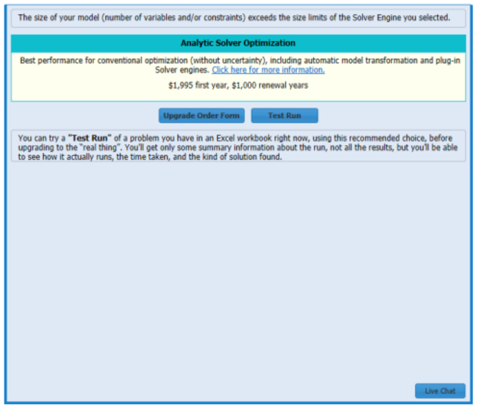
✓ **Gurobi Engine**

The Gurobi Optimizer is a state-of-the-art linear, quadratic, and mixed integer solver, built from the ground up to exploit modern multi-core processors. Click here for more information.

You can try a **"Test Run"** of a problem you have in an Excel workbook right now, using this recommended choice, before upgrading to the "real thing". You'll get only some summary information about the run, not all the results, but you'll be able to see how it actually runs, the time taken, and the kind of solution found.

When you click "Test Run", the Product Wizard will immediately run the optimization or simulation model using the recommended product.   (Only summary information will be available.) At this point, you can purchase the recommended product(s), or close the dialog.

The same behavior will occur whenever you run a simulation or optimization model that contains too many decision variables/uncertain variables or constraints/uncertain functions for the selected engine, the Product Wizard will automatically appear and recommend a product that *can* solve your model.


The size of your model (number of variables and/or constraints) exceeds the size limits of the Solver Engine you selected.

Analytic Solver Optimization

Best performance for conventional optimization (without uncertainty), including automatic model transformation and plug-in Solver engines. Click here for more information.

$1,995 first year, $1,000 renewal years

You can try a **"Test Run"** of a problem you have in an Excel workbook right now, using this recommended choice, before upgrading to the "real thing". You'll get only some summary information about the run, not all the results, but you'll be able to see how it actually runs, the time taken, and the kind of solution found.

Frontline's Large Scale Solver Engines

See the table below for problem types and limits. each engine is specially suited to solve.

By Engine	Bits	Limits	Especially Suited For
	32-bit/64-bit	Var x Const	
Large Scale LP/QP Solver			
Standard Version	Both	32000 x 32000	LP/QP/MIP
Extended Version	Both	Unlim x Unlim	LP/QP/MIP
Large Scale GRG Solver			
Standard Version	Both	4000 x 4000	Smooth nonlinear
Extended Version	Both	12000 x 12000	Smooth nonlinear
Large Scale SQP Solver	Both	Unlim x Unlim	Smooth nonlinear, non-smooth
Knitro	Both	Unlim x Unlim	Smooth nonlinear
MOSEK	Both	Unlim x Unlim	LP/QP, QCP, SOCP & NLP
Gurobi			
LP/MIP	Both	Unlim x Unlim	LP/MIP
LP/QP/MIP	Both	Unlim x Unlim	LP/QP/MIP
XPRESS			
LP/MIP	Both	Unlim x Unlim	LP/MIP
LP/QP/MIP	Both	Unlim x Unlim	LP/QP/MIP
OptQuest	Both	5000 x 1000	Non-smooth

Most of the engines can also handle a wider variety of problems than shown above so the By Problem table below, and the Summary Description for each solver engine in the next chapter, may be helpful as well.

By Problem Type	Best Choices	Alternatives
Linear/MIP	Gurobi, XPRESS, LS LP/QP	MOSEK
Quadratic		
QP Convex	Gurobi, XPRESS, LS LP/QP	MOSEK
QP Non-convex	Gurobi, LS SQP, Knitro, LS LP/QP	LS GRG
Quadratically Constrained		
QCP Convex	Gurobi, MOSEK, Knitro, LS GRG	LS SQP
QCP Non-convex	Gurobi, Knitro, LS GRG, LS SQP	
QCP (unknown convexity)	Gurobi, Knitro, LS GRG, LS SQP	

Second Order Cone	Gurobi, MOSEK, Knitro, LS GRG	LS SQP
Smooth Nonlinear		
NLP Convex	Gurobi, Knitro, LS GRG, LS SQP	
NLP Non-convex	Gurobi, Knitro, LS GRG, LS SQP	
NLP (unknown convexity)	Gurobi, Knitro, LS GRG, LS SQP	
Non-Smooth	OptQuest, LS SQP	LS GRG
Simulation-Optimization	OptQuest, LS SQP	LS GRG

For large-scale linear programming and quadratic programming problems, you can choose the Gurobi Solver, Large-Scale LP/QP Solver, MOSEK Solver or XPRESS Solver. All of these Solvers handle linear mixed-integer problems, but the Gurobi Solver and XPRESS Solver offer **by far the best** performance on difficult MIP problems.

For conic optimization (SOCP or second-order cone programming) problems, the Gurobi, MOSEK, and Xpress Solvers are the leading choice, though the nonlinear Solver Engines can also handle cone constraints.

For large-scale nonlinear programming problems, you can choose the Gurobi Solver, Large-Scale GRG Solver, Large-Scale SQP Solver, or Knitro Solver. Each of these Solvers is effective on large, sparse problems, but the Knitro Solver is exceptionally powerful for smooth convex and nonconvex nonlinear problems. The LSSQP Solver is especially suited for problems where constraints include both linear and nonlinear terms, and it will also handle non-smooth terms with its integrated Evolutionary Solver. The Gurobi Solver can handle all nonlinear models, except t

For non-smooth optimization problems, the OptQuest Solver is an excellent choice.

Summary Description of Each Solver Engine

The Large-Scale LP/QP Solver

Frontline's Large-Scale LP/QP Solver Engine 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. The Large-Scale LP/QP Solver includes a state-of-the-art implementation of the Primal and Dual Simplex methods plus a Quadratic extension, and uses a sparse representation of the LP matrix to handle large problems in limited memory, advanced matrix factorization and updating methods to maintain numerical accuracy, and a Branch and Cut method for solving LP/MIP problems. It is offered in two versions: A Standard Edition, handling problems of up to 32,000 variables and 32,000 constraints; and an Extended Edition, handling very large problems with no fixed limits on variables and constraints (it has been used to solve problems with millions of variables).

V2025 Q1 features a new, higher performance version of the Large Scale LP/QP Solver. Most models will solve **faster – sometimes *much* faster** with this new version, both for LP and QP (linear programming and quadratic programming) models, and for LP and QP models with integer constraints. In addition, this engine now supports **quadratic constraints**, with or without a quadratic objective.

The Large-Scale GRG Solver

Frontline’s Large-Scale GRG Solver Engine is designed to solve smooth nonlinear problems much larger than the 1000 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. Thanks to multi-core automatic differentiation in Analytic Solver Comprehensive and Analytic Solver Optimization, large nonlinear problems can be solved in far less time than ever before.

The Large-Scale SQP Solver

Frontline’s Large-Scale SQP Solver Engine is a state-of-the-art optimizer that is capable of solving linear and quadratic programming problems like those handled by the Large-Scale LP/QP Solver, and smooth nonlinear optimization problems even larger than those handled by the Large-Scale GRG Solver. It is typically faster than the Large-Scale GRG Solver on nonlinear problems, and it handles problems with ten times the “degrees of freedom” (roughly, total variables minus total “active” or binding constraints) of earlier versions. It is especially effective on nonlinear problems with many linear constraints or linear occurrences of variables, since it can exploit information about the model supplied by the PSI Interpreter in Analytic Solver Comprehensive and Analytic Solver Optimization (or by your code in Solver SDK). It will also handle problems with non-smooth or discontinuous terms in the objective or constraints. The Large-Scale SQP Solver uses a Sequential Quadratic Programming method, combined with genetic algorithm methods in its integrated Evolutionary Solver. It has no fixed limit on numbers of variables and constraints.

The Artelys Knitro Solver

Frontline’s Knitro Solver Engine – developed in close cooperation with Ziena Optimization – offers superb performance in solving smooth nonlinear problems, with size limited only by time and memory. The Knitro Solver often outperforms other Frontline Solvers, and nonlinear optimizers from other vendors, on large-scale smooth nonlinear problems with thousands of “degrees of freedom.” The Knitro Solver is the leading implementation of new state-of-the-art interior point nonlinear methods, and it also includes state-of-the-art “active set” SLQP (Sequential Linear / Quadratic Programming) methods that perform very well on highly constrained problems with fewer degrees of freedom. It has no fixed limit on numbers of variables and constraints.

The MOSEK Solver

Frontline’s MOSEK Solver Engine – developed in close cooperation with MOSEK ApS – offers breakthrough performance in solving linear (LP), quadratic (QP), convex quadratically constrained (QCP) and convex second order cone programming (SOCP). Its performance is competitive with the Large-Scale LP/QP and even the Gurobi and XPRESS Solvers on large-scale LP and QP problems, and unlike these other Solvers, it can handle convex quadratic constraints and convex second order cone constraints – solving such ‘nonlinear’ problems with speed and reliability comparable to LP and QP problems. The MOSEK Solver includes both Simplex and interior point methods – the latter is a state-of-the-art implementation of the homogeneous self-dual method, as described in the Appendix under “The MOSEK

Solver Methodology." A license for Mosek handles very large problems with no fixed limits on variables and constraints.

The Gurobi Solver

Frontline's Gurobi Solver Engine, developed in close cooperation with Gurobi Optimization, is the **fastest** linear programming (LP), quadratic programming (QP), and mixed-integer linear programming (LP/MIP) optimizer available today, according to several independent third-party benchmarks. The Gurobi Solver, developed by three former leaders of the CPLEX development team, has been engineered from the ground up to **exploit multi-core processors** more effectively than other Solvers. It has no fixed limit on problem size, and has been used to solve problems with millions of variables and constraints. There are two versions available, one for LP/MIP problems and another which also solves QP problems.

Analytic Solver V2025 Q1 supports mixed-integer nonlinear optimization problems, including those that are often non-convex. These problems are common in industries such as specialty chemicals, hydroelectric power, and certain aeronautics and space applications. Unlike other nonlinear solvers, such as LSGRG, LSSQP, and KNITRO, Gurobi uses a different approach that can provide globally optimal solutions—provided the model is formulated using supported mathematical operators, plus addition, subtraction, multiplication, division, square, square root, exponential, logarithmic, and similar functions. No additional effort is required to define your nonlinear model for the Gurobi Solver; Frontline's Interpreter converts your existing Excel formulas into calculations compatible with the Gurobi Solver.

The XPRESS Solver

Frontline's XPRESS Solver Engine, developed in close cooperation with FICO, Inc., brings the lightning-fast performance, and virtually unlimited problem solving capacity of the Xpress^{MP} mixed-integer linear o Analytic Solver V2025 Q1 supports mixed-integer nonlinear optimization problems, including those that are often non-convex. These problems are common in industries such as specialty chemicals, hydroelectric power, and certain aeronautics and space applications. Unlike other nonlinear solvers, such as LSGRG, LSSQP, and KNITRO, Gurobi uses a different approach that can provide globally optimal solutions—provided the model is formulated using specific algebraic expressions. These expressions must involve only a limited set of mathematical operators, including addition, subtraction, multiplication, division, square, square root, exponential, logarithmic, and similar functions. No additional effort is required to define your nonlinear model for the Gurobi Solver.

ptimizer to Excel, MATLAB and Java users. The XPRESS Solver can also solve quadratic programming (QP) problems, and mixed-integer quadratic (QP/MIP) problems. It offers more than 60 Solver Options, described in "XPRESS Solver Options" in the chapter "Solver Options." It has no fixed limit on numbers of variables and constraints.   There are two versions available, one for LP/MIP problems and another which also solves QP problems. Please note that special considerations apply to distribution of the XPRESS Solver with a custom application; contact Frontline Systems for details.

The OptQuest Solver

Frontline's OptQuest Solver Engine, developed in close cooperation with OptTek Systems, is designed to work with all types of models. Your Excel model or custom program can use any standard or user-written (numeric) functions – even discontinuous functions such as IF, CHOOSE, LOOKUP, and COUNT that cause difficulty for the Large-Scale GRG, Large-Scale SQP and Knitro 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 later in this chapter under "The OptQuest Solver Methodology." It supports up to 5,000 variables and 1,000 constraints, though the practical size of problems that can be solved to near-global optimality may be less than these limits.

Special Capabilities of the Solver Engines

Each of the Solver Engines offers more than just classical linear programming or nonlinear optimization. By leveraging the capabilities of the Analytic Solver Comprehensive, Analytic Solver Optimization and the Solver SDK as well as their own strengths, many of these Solvers can be used to tackle conic optimization, robust optimization, simulation optimization, and global optimization problems, analysis of infeasible problems, mixed-integer nonlinear problems, and even basic constraint programming problems.

Conic Optimization with Solver Engines

In all Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK, Frontline Systems is first to offer broad support for conic optimization in a commercial product line. Second order cone programming (SOCP) problems can be solved with the SOCP Barrier Solver or the MOSEK Solver, with remarkable speed. In addition, second order cone constraints can be used in linear or nonlinear problems that are solved with the standard GRG, Large-Scale GRG, Large-Scale SQP, and Knitro Solvers. Both standard and "rotated" second order cone constraints may be specified. Conic optimization – the natural generalization of linear and quadratic optimization – has many applications in finance and engineering design.

Robust Optimization with Solver Engines

In Analytic Solver Comprehensive and Analytic Solver Optimization, Frontline Systems is again first to offer general-purpose support for robust optimization in a commercial product line. All of Frontline's Solver Engines can be applied to solve robust counterpart problems, created *automatically* by Analytic Solver, since these are normally linear, quadratic, or second-order cone programming problems.

The MOSEK Solver is an especially good match for Analytic Solver Comprehensive or Optimization, because it can solve – with high performance – problems generated by Analytic Solver's *Robust Counterpart Transformation* for all four choices of norms, including the L2 norm (which creates large-scale SOCP robust counterpart problems).

Simulation Optimization with Solver Engines

Frontline Systems offers "best in class" support for simulation optimization, using the Solver Engines with Analytic Solver Comprehensive, Analytic Solver Optimization with Analytic Solver Simulation, and Solver SDK. You can solve simulation optimization problems up to *hundreds of times faster* than competitive products for Microsoft Excel. Solver SDK includes both optimization and Monte

Carlo simulation capabilities in one package – it provides pre-written examples of simulation optimization in C++, C#, Visual Basic, VB.NET, Java, and MATLAB.

The Large-Scale GRG, Large-Scale SQP, Knitro, and OptQuest Solvers can be applied to simulation optimization problems. The OptQuest Solver is an especially popular choice for this purpose, since it can find solutions to difficult non-smooth optimization problems that can arise in simulation optimization.

Global Optimization with Solver Engines

Most nonlinear optimizers are guaranteed only to find a *locally optimal* solution to a nonconvex problem. Imagine a graph of the objective function with "hills" and "valleys:" Such optimizers will typically find the peak of a hill near the starting point you specified (if maximizing), but they 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 or need to find a *globally optimal* solution. Frontline's Solver Engines offer you several different approaches to global optimization.

The OptQuest Solver, like the Evolutionary Solver bundled in Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK, is designed from the ground up to search for globally optimal, or near-optimal solutions of problems with *non-smooth* functions. It uses a search strategy that moves aggressively past locally optimal solutions towards better solutions, ultimately towards the globally optimal solution.

You can also use the Large-Scale SQP Solver engine to solve problems with non-smooth functions, using its integrated Evolutionary Solver. This Solver Engine is often the best choice for large problems that are mostly smooth or even linear, but include some non-smooth terms in the objective or constraints.

Using an alternative approach, the bundled nonlinear GRG Solver, the Large-Scale GRG Solver, the Large-Scale SQP Solver, and the Knitro Solver can make use of multistart methods for global optimization: They can be run automatically many times from judiciously chosen starting points, and the best solution found (the "highest peak" if maximizing) will be returned as the best estimate of a globally optimal solution. Using the multistart methods is as simple as setting the Multistart Search option to True in the Task Pane Engine tab, or setting a property or calling an API function in the Solver SDK.

Analysis of Infeasible Problems

When the normal optimization process finds no feasible solutions, all Solver Engines except those aimed at non-smooth problems (the Evolutionary and OptQuest Solvers) can be used to analyze your model and find an Irreducibly Infeasible Subset (IIS) of the constraints. The "IIS Finder" in Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK will automatically run the Solver Engine on problems with different subsets of the constraints to isolate an IIS. The results can be displayed in the Excel products' Feasibility Report, or obtained via properties or API functions for further use in your application program.

Integer, Semi-Continuous, and Alldifferent Variables

Each of the Solver Engines handles general integer and binary integer variables. Binary or 0-1 integer variables are often used to represent yes/no decisions in an optimization model. Integer variables make it possible to model and solve a wide range of problems that could not be handled with conventional continuous variables.

You can also specify that a decision variable is *semi-continuous*. At the optimal solution, a semi-continuous variable is either zero, or a continuous value in a range that you specify. Semi-continuous variables can serve some of the common purposes of binary integer and continuous variables, and they can be handled very efficiently in the solution process. All Solver engines support semi-continuous variables, *except* the Knitro Solver. .

You can also specify that a set of integer variables must be "alldifferent." Such variables will then have integer values from 1 to N (the number of variables), all of them *different* at the solution. You can use such "alldifferent" constraints to model problems involving ordering or permutations of choices, such as the well-known Traveling Salesman Problem.

The Solver Engines take very different approaches to solving problems containing "alldifferent" constraints – so you can model the problem in a high-level way, and try a variety of Solver engines to see which one yields the best performance.

Standard and Solver Engine-Specific Reports

In addition to the optimal solution (final values for the decision variables) found for your problem, the Solver Engines offer further information in the form of reports. In Analytic Solver Comprehensive and Analytic Solver Optimization reports are produced as *Excel worksheets* that can be read, modified, or used as the basis for further calculations in Excel or VBA. In both Excel and Solver SDK, the report information is returned via object properties, for further analysis by your application program.

Analytic Solver Comprehensive and Analytic Solver Optimization offer eight types of standard reports for Solver engines: The Answer, Sensitivity, and Limits Reports (also present in the standard Excel Solver), and the new Scaling Report, Linearity Report, Feasibility Report, Solutions Report, and Population Report. In addition, Analytic Solver's PSI Interpreter offers the Structure Report and Transformation Report. In Solver SDK, you can access the information found in the Answer, Sensitivity, Linearity, Feasibility, Solutions, and Population Reports.

The Large-Scale LP/QP Solver and the XPRESS Solver support the first seven types of reports, and the Large-Scale GRG Solver, Large-Scale SQP Solver, and Knitro Solver support all of these except the Linearity Report. The MOSEK Solver supports the Answer, Sensitivity, Limits, Scaling, and Solutions Reports. The OptQuest Solver supports the Answer, Scaling, Solutions, and Population Reports. The Analytic Solver User Guide describes the standard reports in more detail, and provides examples.

Programming the Solver Engines

Analytic Solver Comprehensive and Analytic Solver Optimization and the Solver Engines are fully programmable from VBA (Visual Basic Application Edition) in Excel. Two comprehensive APIs (Application Programming Interfaces) are supported: a **legacy VBA macro interface** that is upward compatible from the Excel Solver, and a **new object-oriented interface** that is compatible with Frontline's Solver SDK, and is easy to use from C# and VB.NET.

If you prefer to deliver your application to end users in Excel, this means you can easily build an application using a bundled Solver or a plug-in, large-scale Solver

Engine, hide the Solver user interface, and (using VBA) present a customized user interface for your end users.

Using Your Excel Solver Model – Outside Excel

If you prefer to deliver your application to end users **outside** Excel – for example on an Intranet or Web server, or even on Linux instead of Windows – ***you can still build your optimization model in Excel!*** Solver SDK can load your Excel workbook, interpret the model you've defined in Excel, enable you to control this model – using object-oriented API calls almost identical to the ones in VBA – and solve the model with very high performance.

You can obtain new data from a database, another application or the end user, use it to update your Excel model, and solve a new problem instance. And you can do all of this in a self-contained Windows program that's very easy to deploy.

With just a little care, you can create your SDK-based application so it can work with updated and improved versions of your Excel workbook model, or even with several different Excel workbooks, without having to be modified or even recompiled! This makes for a very flexible and powerful deployment solution.

At any point if you wish, you can rewrite your Excel formulas in C++, C#, VB.NET, Java or other programming language code, and move your entire optimization or simulation model into your SDK application.

Installation and Licensing

What You Need

The Solver engines covered in this Guide are Dynamic Link Libraries on Windows, and Shared Libraries on Linux. They do not run as “stand-alone” programs; instead they are designed to “plug into” Analytic Solver Comprehensive (Desktop or Cloud), Analytic Solver Optimization (Desktop or Cloud) or Solver SDK.

You can use Analytic Solver Cloud in Excel for the Web through a Web browser (such as Edge, Chrome, Firefox or Safari), without installing anything else. This is the simplest and most flexible option, but it requires a constant Internet connection.

To use Analytic Solver Cloud in Excel Desktop on a PC or Mac, you must have a current version of Windows or iOS installed, and **you will need the latest Excel version installed via your Office 365 subscription** – older non-subscription versions, even Excel 2019, do not have all the features and APIs needed for modern JavaScript add-ins like Analytic Solver Cloud.

To use Analytic Solver Desktop (Windows PCs only), you must have first installed Microsoft Excel 2013, 2016, 2019, or the latest Office 365 version on Windows 10, Windows 8, Windows 7, or Windows Server 2019, 2016 or 2012. (Windows Vista or Windows Server 2008 may work but are no longer supported.). It’s not essential to have the standard Excel Solver installed.

To install Analytic Solver Desktop and all Large Scale engines to work with any version of Microsoft Excel, simply run the program **SolverSetup.exe**, which installs all of the Solver program, Help, User Guide, and example files in compressed form for both 32-bit and 64-bit Excel. SolverSetup.exe checks your system, detects what version of Office you are running (32-bit or 64-bit) and then downloads and runs the appropriate Setup program version. See the chapter "Installation and Licensing" within the Analytic Solver User Guide for step-by-step installation instructions.

To install Solver SDK and all Large Scale engine, simply run either **SDKSetup.exe** (for 32-bit) or **SDKSetup64.exe** (for 64-bit), to install all of the Solver and Engine programs, Help, User Guides, and example files in compressed form for 32-bit or 64-bit SDK, respectively. The chapter “Installation and Licensing” in the Solver SDK User Guide covers installation step-by-step.

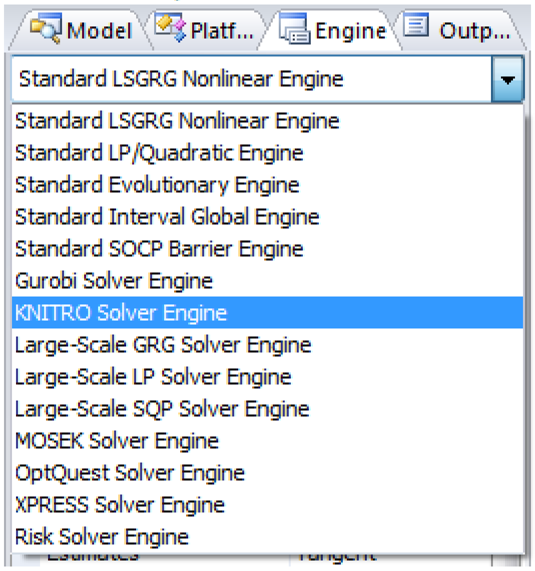
Solver engines will run on the same hardware and system software configuration that you’ve used to run the related Analytic Solver Desktop product. If you solve very large models, however, performance may depend on the amount of main memory (RAM) in your system. Large models with many integer constraints can take substantially more time to solve, and require more memory than models without such constraints. If you are solving in the Cloud, the amount of RAM your system contains is not an issue.

Solver SDK is most often used with Microsoft Visual Studio, but it can also be used with Sun Java, MATLAB, Visual Basic 6.0, and older versions of Visual Studio. For installation instructions for Solver SDK see the "Installation and Licensing" chapter within the Solver SDK User Guide.

Solver Engines 2025 Q1 work with Analytic Solver V2025 Q1 (or Solver Engines V2025 Q1 work with Solver SDK V2025 Q1). If you've been using an earlier version, you'll need to upgrade to Version 2025 Q1 at the same time that you upgrade your existing license for Analytic Solver Comprehensive and Analytic Solver Optimization. Solver Engines 2025 Q1 are **upward compatible** with earlier versions and generally offer better performance – your existing applications should work as-is with the new Solver Engines. If you are using Analytic Solver Cloud, you will always be using the most recent version of our software.

Using the Solver Engines

To use the Solver Engines after installation with Analytic Solver Desktop or Cloud, simply start Microsoft Excel (Online or Desktop) and click the Analytic Solver tab on the Ribbon. In the Task Pane Engine tab, click the Solver Engine dropdown list.


Model
Platf...
Engine
Outp...
Standard LSGRG Nonlinear Engine
Standard LSGRG Nonlinear Engine
Standard LP/Quadratic Engine
Standard Evolutionary Engine
Standard Interval Global Engine
Standard SOCP Barrier Engine
Gurobi Solver Engine
KNITRO Solver Engine
Large-Scale GRG Solver Engine
Large-Scale LP Solver Engine
Large-Scale SQP Solver Engine
MOSEK Solver Engine
OptQuest Solver Engine
XPRESS Solver Engine
Risk Solver Engine

If you select one of the new Solver Engines from the dropdown list, it will be used when you next run an optimization, and its Solver Options will appear on the Engine tab, where you can examine or change them.

To use a Solver Engine with Solver SDK, simply *add* it to the Engines collection and then *select* it as the current Solver engine, as described in the SDK User Guide. For example, you could write in C++:

```
CProblem myProb;
CEngine myEngine ("Knitro Solver", "Knitroeng.dll");
myProb.Engines.Add (myEngine);
```

to add the Knitro Solver Engine to myProb's Engines collection, and:

```
myProb.Engine = myProb.Engines["Knitro Solver"];
myProb.Solver.Optimize();
```

to select the Knitro Solver and use it to solve a problem:

Working with Licenses in Versions 2025 Q1

The Analytic Solver products V2025 Q1 and plug-in Solver Engines V2025 Q1 use a new license manager that simplifies the handling of licenses and license codes for both you and Frontline Systems.

A **license** is a grant of rights, from Frontline Systems to you, to use our software in specified ways. As we learned in the last chapter, when you run the SolverSetup program to install Analytic Solver Desktop, software for the full Analytic Solver product family is installed. However, the product features you see in Analytic Solver Desktop and Analytic Solver Cloud, and the size of models you can solve, depends on the license you have. Analytic Solvers V2025 Q1 uses a **licensing system**, developed in V2012, that offers you more flexible ways to use the software, both desktop and cloud.

Analytic Solver - Licenses Tied to You, Not Your Computer

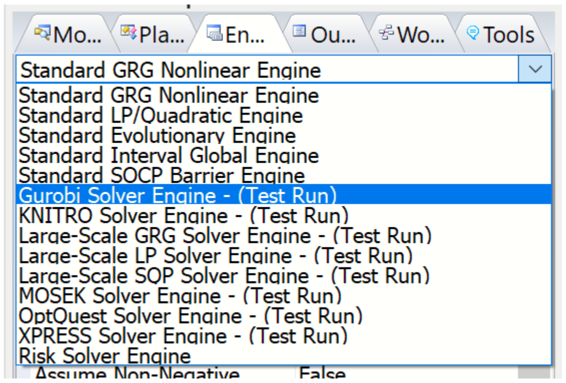
When you first use Analytic Solver either in the cloud or on a new computer, **you will be prompted to login**. Enter the **email address** and **password** that you used to register on Solver.com. Once you've done this in Analytic Solver Desktop, your identity will be "remembered," so you won't have to login every time you start Excel, and go to one of the Analytic Solver tabs. If you are using Analytic Solver Cloud, you will need to login at the beginning of each session.

You can login and logout at any time, using **Login/Logout** on the License menu. If you share use of a single physical computer with other Analytic Solver users, be careful to **login** with your own email and password, and **log out when you're done** – if you don't, other users could access private files in your cloud account, or use up your allotted CPU time or storage.

When you move from one computer to another, you should **log out** on one and **log in** on the other. As a convenience, if you log in to Analytic Solver on a new computer when you haven't logged out on the old computer, Analytic Solver will let you know, and offer to automatically log you out on the other computer.

Taking Solver Engines for a Test Run

Frontline Systems allows you to **Test Run** models with all Large-Scale Solver Engines. (You don't get full results for all variables and constraints, but you do get the final objective value and solution time.) To make it clear what your current license does and doesn't include, in V2025 Q1 you'll see "(Test Run)" next to the names of optional Solver Engines in the Task Pane.



Standard GRG Nonlinear Engine

Standard GRG Nonlinear Engine
Standard LP/Quadratic Engine
Standard Evolutionary Engine
Standard Interval Global Engine
Standard SOCP Barrier Engine
Gurobi Solver Engine - (Test Run)
KNITRO Solver Engine - (Test Run)
Large-Scale GRG Solver Engine - (Test Run)
Large-Scale LP Solver Engine - (Test Run)
Large-Scale SOP Solver Engine - (Test Run)
MOSEK Solver Engine - (Test Run)
OptQuest Solver Engine - (Test Run)
XPRESS Solver Engine - (Test Run)
Risk Solver Engine

Assume Non-Negative False

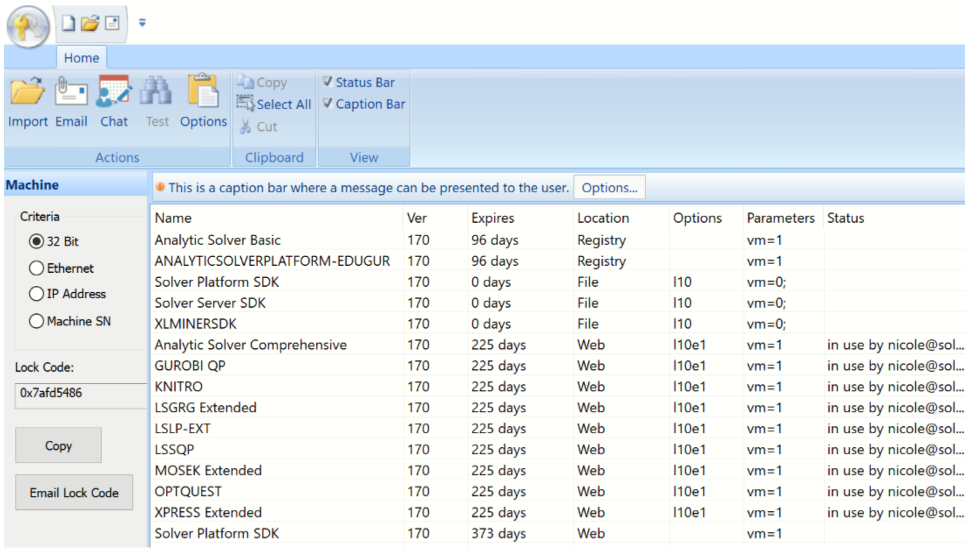
Upgrading from Earlier Analytic Solver Versions

A new license will not be required for V2025 Q1 if you're upgrading from Analytic Solvers V2017/V2018. Old license codes for V2016 and earlier have no negative effect in 2025 Q1. If they exist in the obsolete Solver.lic file (located at

C:\ProgramData\Frontline Systems), they will be ignored. A new license will be issued at the time of purchase.

Managing Solver Licenses Outside Excel

When you install the SDK Platform product, the **SolverLicMan** utility program is also installed. SolverLicMan.exe is installed at C:\Program Files\Frontline Systems\Engines (substitute "Program Files (x86)" for "Program Files" if you've installed 32-bit SDK Platform on a 64-bit operating system). Use this utility to manage all of your licenses in one convenient place.



Criteria
[✓ 32 Bit] Ethernet IP Address Machine SN

Lock Code: 0x7afd5486

This is a caption bar where a message can be presented to the user. Options...

Name	Ver	Expires	Location	Options	Parameters	Status
Analytic Solver Basic	170	96 days	Registry		vm=1	
ANALYTICSOLVERPLATFORM-EDUGUR	170	96 days	Registry		vm=1	
Solver Platform SDK	170	0 days	File	l10	vm=0;	
Solver Server SDK	170	0 days	File	l10	vm=0;	
XLMINERSDK	170	0 days	File	l10	vm=0;	
Analytic Solver Comprehensive	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
GUROBI QP	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
KNITRO	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
LSGRG Extended	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
LSLP-EXT	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
LSSQP	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
MOSEK Extended	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
OPTQUEST	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
XPRESS Extended	170	225 days	Web	l10e1	vm=1	in use by nicole@sol...
Solver Platform SDK	170	373 days	Web		vm=1	

Solver Result Messages

If You Aren’t Getting the Solution You Expect

This chapter documents the Solver Result Messages that can be returned when you optimize a model, and discusses some of the characteristics and limitations of the Solver Engines. You should read this chapter in conjunction with the Analytic Solver User Guide chapter “Getting Results: Optimization.”

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, nonlinear or non-smooth optimization problem – as discussed in depth in the chapter “Mastering Conventional Optimization Concepts” in the Analytic Solver User Guide. If you then build your model in a *well-structured, readable and efficient form,* diagnosing problems should be relatively easy. But at times you may be “surprised” by the results you get from your Solver engine.

If the Solver stops or returns with a solution (set of values for the decision variables) 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 Task Pane Output tab, or the corresponding value of **Problem.Solver.OptimizeStatus** in VBA (value of **SolverSolve** if you’re using the legacy VBA functions) or Solver SDK. Users sometimes contact Frontline Systems about “wrong solutions”, but they don’t know which result they received – this is crucial to diagnosing the problem. **Read carefully the discussion** of your Solver Result Message or OptimizeStatus code in the following sections.
- In Analytic Solver Comprehensive and Analytic Solver Optimization, **review the solution log messages in the Task Pane Output tab**. Set Task Pane Platform tab General group **Log Level** option to **Verbose** before you solve, to obtain maximum information from the solution log.
- In Analytic Solver Comprehensive and Analytic Solver Optimization, **examine the available optimization reports** – notably the Linearity Report, Structure Report, Feasibility Report, and Scaling Report. In Solver SDK, you can call the **Problem.Model.DependCheck** or **DependTest** method to obtain the same information as the Linearity Report, and call the **Problem.Solver.IISFind** method to obtain the same information as the Feasibility Report.
- **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 Analytic Solver Comprehensive and Analytic Solver Optimization, many messages from the **PSI 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 PSI Interpreter, or else you'll have to set the Task Pane Optimization Model group Interpreter option to Excel Interpreter.

- In Solver SDK, the object-oriented API will raise a specific **SolverException**, and the procedural API will return specific exception codes, when the Solver detects errors or inconsistencies in your use of SDK properties, methods or API functions. Your application code should include **try/catch blocks for SolverExceptions**, or tests for nonzero return values from procedural API function calls.
- For a detailed look at the solution process, set the Show Iteration Results option to True and re-solve, or in VBA or Solver SDK, define an **Evaluator** to be called on every iteration, and inspect the current trial solution in your Evaluator code. The iterations show you the path taken towards the solution.
- Consider the impact of a **poorly scaled model**, the **role of the Tolerance option** for integer problems, and any limitations or special considerations for your Solver Engine as outlined in this chapter.

Standard Solver Result Messages

The bundled Solver engines in Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK, and the plug-in Solver Engines covered by this Guide, return the standard 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, which are documented later in this section.

In addition, the Large-Scale GRG Solver, Large-Scale SQP Solver, Knitro Solver, MOSEK Solver, Gurobi Solver, XPRESS Solver, and OptQuest Solver each return certain engine-specific result codes and display related Solver Result Messages in special circumstances. These are described in sections for each Solver Engine.

-1. A licensing problem was detected, or your trial license has expired.

This message appears if a product cannot find its licensing information, if the licensing information is invalid, or if you have a time-limited evaluation license that has expired. You need a license for both Analytic Solver Comprehensive, Analytic Solver Optimization, *or* Solver SDK *and* the plug-in Solver Engine you are using. In Excel, *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 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 the type of problem you are solving, as outlined below. In general, for smooth *convex* problems (including all linear and certain quadratic problems), the Solver has found a globally optimal solution, but for *non-convex* problems, this message guarantees only a locally optimal solution.

If you are using the Large-Scale LP/QP Solver, Large-Scale SQP Solver, MOSEK Solver, XPRESS Solver or Gurobi Solver to solve a linear or convex quadratic problem with no integer constraints, or you are using the MOSEK Solver to solve an SOCP with no integer constraints, the Solver has found the *globally* optimal solution: There is *no* other solution satisfying the constraints which has a better value for the objective. It is possible that there are other solutions with the *same* objective value, but all such solutions are linear combinations of the current decision variable values.

If you are using the Large-Scale GRG Solver, Large-Scale SQP Solver, or Knitro Solver to solve a *smooth nonlinear* optimization problem with no integer constraints, the Solver has found a *locally* optimal solution in the neighborhood of the starting values of the variables: 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. You can assume that the solution is globally optimal only if you know that the problem is convex; otherwise there *may* be other sets of values for the variables, far away from the current values, that yield better values for the objective and still satisfy the constraints.

If you are solving a *mixed-integer* programming problem (any problem with integer constraints), this message means that the Solver has found a solution satisfying the constraints (including the integer constraints) with the best possible objective value – it has 'proved optimality' by searching for all possible alternative integer solutions, finding none better. If the problem (without the integer variables) is convex, the true integer optimal solution has been found. If the problem is non-convex, the Solver has found the best of the locally optimal solutions found for subproblems by the Large-Scale GRG, Large-Scale SQP, or Knitro Solver.

If you are using the OptQuest Solver, this message means that the best solution found so far is available. Since it is designed to handle complex, non-smooth functions, the OptQuest Solver can *prove* optimality only in the (rare) case where all variables are integer and have relatively tight lower and upper bounds, making complete enumeration of all trial solutions possible.

1. Solver has converged to the current solution. All constraints are satisfied.

This means that the Large-Scale GRG Solver, Large-Scale SQP Solver, Knitro Solver, or MOSEK Solver has found a series of "best" solutions that satisfy the constraints, and that are very similar to each other; however, no single solution strictly satisfies the Solver's test for optimality. The Convergence tolerance in the Task Pane Engine tab controls how similar the solutions must be. More precisely, the Large-Scale GRG Solver stops with this message if the *absolute value of the relative change in the objective function* is less than this tolerance. The Large-Scale SQP Solver stops if the *maximum normalized complementarity gap of the variables* is less than this tolerance, for the last few iterations. The Knitro Solver stops if no further progress can be made and the optimality conditions are satisfied to within a factor of 100. A *poorly scaled* model is more likely to trigger this stopping condition, even if the Use Automatic Scaling option in the Task Pane Engine tab is set to True. If you are sure that your model is well scaled, you should consider why it is that the objective function is changing so slowly. For more information, see the discussion of "Large-Scale Nonlinear Solver Stopping Conditions" below.

When the Evolutionary Solver in the Large-Scale SQP Solver Engine 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 Task Pane Engine tab. 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.

2. Solver cannot improve the current solution. All constraints are satisfied.

This means that the Large-Scale GRG Solver, Large-Scale SQP Solver, or Knitro Solver has found solutions that satisfy the constraints. However, the respective Solver 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. This message rarely occurs. It means that the Solver has encountered numerical accuracy or stability problems in optimizing the model, and it has tried all available methods to overcome the numerical problems, but cannot reach an optimal solution. One possibility worth checking is that some of your constraints are redundant, and should be removed. For more information, see the discussion of "Large-Scale Nonlinear Solver Stopping Conditions" below.

When the Evolutionary Solver in the Large-Scale SQP Solver Engine 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 in the Task Pane Engine tab, in the amount of time specified by the Max Time without Improvement option in the same option group. 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.

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 by the Iterations option in the Task Pane Engine tab *and* (ii) you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. For the Gurobi Solver, when its Barrier (interior point) algorithm is used, this message appears when the Barrier Iteration Limit is exceeded.

You may increase the value of the Iterations option, 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 unlikely to appear.

4. The objective (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 objective may take on negative values without limit unless this is prevented by the constraints or bounds on the variables. Set the Assume Non-Negative option in the Task Pane Engine tab to True to impose >= 0 bounds on all variables.

If the objective is a linear function of the decision variables, it can *always* be increased or decreased without limit (picture it as a straight line), so the Solver will seek the extreme value that still satisfies the constraints. If the objective is a nonlinear function of the variables, it may have a "natural" maximum or minimum (for example, =A1*A1 has a minimum at zero), or no such limit (for example, =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 in the Large-Scale SQP Solver Engine and the OptQuest Solver *never* display this message, because they have no way of systematically increasing (or decreasing) the objective function, which may be non-smooth. If you have forgotten a constraint, these Solvers *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*. Generally speaking, if you are solving a linear problem with any Solver Engine, or solving a convex nonlinear problem with a Solver Engine other than the OptQuest Solver, and the model is well scaled, then the Solver has determined for *certain* that there is no feasible solution. But the interior point methods used in the Knitro Solver may sometimes have difficulty determining feasibility if there are many equality constraints, especially if the model is not well scaled.

If you are using the Evolutionary Solver in the Large-Scale SQP Solver Engine or the OptQuest Solver and the model has nonlinear constraints, or if you are using the Large-Scale GRG Solver, Large-Scale SQP Solver, or Knitro Solver, the Solver was unable to find a feasible solution; however it is possible that there is a feasible solution outside of the region(s) searched. In general, the Solver's search depends heavily on the starting point (i.e. the initial values of the variables); if you start this Solver from a very different starting point, it might find a feasible solution.

If you are solving a *mixed-integer* programming problem (any problem with integer constraints), this message means that there are no solutions that satisfy all of the constraints, including the integer constraints on variables. You can try solving the "relaxation" of the original problem (which ignores the integer constraints), to see if a feasible solution to this simplified problem can be found. In VBA and Solver SDK, you can do this by calling Problem.Solver.Optimize with argument Solve_Type_NoIntegers.

If you are solving a problem with chance constraints using simulation optimization, this message means that the Solver could find no solution that satisfies these constraints to the chance measures (such as 95%) that you specified. If you 'relax' the chance measures (to say 90%) and solve again, it's possible that a feasible solution will be found. For robust optimization, see result codes 26 through 29.

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. <= instead of >=) on an otherwise appropriate constraint. The easiest way to find conflicting constraints in Analytic Solver Comprehensive or Analytic Solver Optimization is to select the Feasibility Report, shown in the **Reports – Optimization** gallery when this Solver Result Message appears, and click OK. In Solver SDK, call Problem.Solver.IISFind and use the OptIIS property, or in the procedural API call SolverOptIISFind to get the same information.

6. Solver stopped at user's request.

In Analytic Solver Comprehensive and Analytic Solver Optimization, this message appears if you press ESC to display the Show Trial Solution dialog, and then click 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. In Solver SDK, this result code is returned if you've defined an Evaluator to be called on each iteration or subproblem, and your Evaluator returned the “user abort” code to its caller.

If you are using a shared network license, it’s also possible – though unlikely – that the license server could “go down” or you might otherwise lose your active license, yielding this message and return code.

7. The linearity conditions required by this Solver engine are not satisfied.

If you are using the Large-Scale LP/QP Solver or the MOSEK Solver, or if you are using the Large-Scale SQP Solver or Knitro Solver and have selected one or both of the options “Treat Objective as Linear” or “Treat Constraints as Linear,” this message appears if the Solver’s numeric tests to ensure that the objective and constraints are indeed linear functions of the decision variables were not satisfied. To understand exactly what is meant by a linear function, read the chapter “Mastering Conventional Optimization Concepts” in the Analytic Solver User Guide.

If you receive this message, examine your formulas or program statements for the objective and constraints, looking for nonlinear or non-smooth functions or operators applied to the decision variables. In Analytic Solver Comprehensive and Analytic Solver Optimization, select the Linearity Report, or – even better – select a Structure Report to pinpoint the exact cell formulas that aren’t linear. In Solver SDK, call the Problem.Model.DependCheck (if you’ve loaded an Excel workbook), or the DependTest method if you’ve written your own Evaluator for the objective and constraints, to test your objective and constraints for linearity. In the procedural API, call the SolverModDependTest function.

8. The problem is too large for Solver to handle.

This message – or the more specific message **Too many variable cells**, **Too many constraints**, or **Too many integer variable cells** – appears when the Solver determines that your model is too large for the Solver Engine that you are using. 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.

In Analytic Solver Comprehensive and Analytic Solver Optimization, 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 examining the Current Problem and Engine Limits groups in the Engine tab for that Solver Engine. In Solver SDK, you can access the Solver Engine object’s EngineLimit properties, or call the SolverEngLimit procedural API function.

9. Solver encountered an error value in a target or constraint cell.

In Analytic Solver Comprehensive and Analytic Solver Optimization, 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 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, we recommend that you use only constants and cell references on the right hand sides of constraints. In Solver SDK, this result code is returned if your Evaluator for function values returns a nonzero value to its caller. You’ll have to examine your Evaluator to further diagnose the problem.

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 that lead to arguments outside the domains of your functions – such as a negative value supplied to SQRT. In Solver SDK, this kind of error will usually raise one of the standard numeric exceptions, which you can "catch" in your code. 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 Analytic Solver Comprehensive and Analytic Solver Optimization, when the Polymorphic Spreadsheet Interpreter is used, 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 Sheet1!$A$1," where Sheet1!$A$1 is replaced by the cell address where the error was encountered. Usually, a more specific message will appear. 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.

"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. "Non-smooth special function" may appear if your model uses functions ABS, IF, MAX, MIN or SIGN.

The Evolutionary Solver in the Large-Scale SQP Solver Engine and the 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.

In Analytic Solver Comprehensive and Analytic Solver Optimization, this message appears when (i) the Solver has run for the maximum time (number of seconds) allowed by the Max Time option in the Task Pane Engine tab *and* (ii) you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value of the Max Time option 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. In Solver SDK, you set the “MaxTime” parameter to the number of seconds you want to allow the Solver to run before stopping with this result code.

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. If you are using 32-bit Excel or the 32-bit version of Analytic Solver Comprehensive or Analytic Solver Optimization and you receive this message, you should strongly consider moving to 64-bit Excel 2013 and Analytic Solver, or to the Solver SDK 64-bit version, both of which can address far more memory.

Since Microsoft Windows supports a “virtual memory” that can be larger than your available RAM by swapping data to your hard disk, you may notice that solution times have greatly slowed down, and the hard disk activity light in your PC is flickering during the solution process. In this case, your simplest and least cost solution may be to upgrade your PC with more RAM.

The Polymorphic Spreadsheet Interpreter in Analytic Solver Comprehensive and Optimization and Solver SDK can use a considerable amount of memory, when you solve a problem or analyze a model. You can progressively reduce the memory used by the Interpreter by taking the following actions in order, using the Task Pane Platform tab:

1. Set the Use Internal Sparse Representation option in the Advanced options group to True.
2. Set the Supply Engine with option to Gradients.
3. Set the Optimization Model Interpreter option to Excel Interpreter.

When you use 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.

In Solver SDK, you can either load an Excel workbook model and use the PSI Interpreter, or write your own Evaluator function to compute values for your objective and constraints, without using either Excel or the PSI Interpreter. With this latter option, you may be able to solve a larger problem on the same computer.

Most large problems are *sparse* in that a typical constraint depends on only a small subset of the variables. The large-scale Solver engines described in this User Guide use *sparse* representations of your model, to save memory as well as solution time.

12. *Error condition* at cell *address*. Edit your formulas, or use Excel Interpreter in the Solver Model dialog.

In Analytic Solver Comprehensive and Analytic Solver Optimization, this message appears when the Interpreter first analyzes the formulas in your model, after you click the Optimize button or the green arrow in the Task Pane. *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… commands in Excel. The “labels in formulas” feature was dropped in Excel 2007 and Excel 2010, and the Interpreter does not support this use of labels in formulas. You should define these labels with the Insert Name Define… or Insert Name Create… commands.

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 use the Excel 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 use the 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.

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, objective, constraints, Solver options, etc.) is not in a valid form. An “empty” or incomplete Solver model, perhaps one with no objective and no constraints other than bounds on the variables, can cause this message to appear. You might also receive this message if you are using the *wrong version* of certain Solver files, 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 the Large-Scale LP/QP Solver, Large-Scale SQP Solver, MOSEK

Solver, XPRESS or Gurobi Solver, with a *non-zero value* for the integer Tolerance option in the Task Pane Engine tab (in VBA and Solver SDK, the “IntTolerance” parameter), 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.

The solution found when this message (or result code) appears may actually *be* the true integer optimal solution; however, the Branch & Bound method did not take the extra time to search all possible remaining subproblems to “prove optimality” for this solution. If all subproblems *were* explored (which can happen even with a non-zero Tolerance in some cases), the Solver will produce the message “Solver found a solution. All constraints are satisfied” (result code 0).

15. Stop chosen when the maximum number of integer solutions was reached.

If you are using the Large-Scale LP/QP Solver, Large-Scale SQP Solver, MOSEK Solver, XPRESS or Gurobi Solver on a problem with integer constraints, this message appears when the Solver has found the maximum number of integer solutions (values for the variables that satisfy all constraints, including the integer constraints) allowed by the Max Integer Solutions option in the Task Pane Engine tab (in VBA and Solver SDK, the “MaxIntegerSols” parameter), *and* in Excel, you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value of the Max Integer Solutions option, 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 well-formulated, and whether you can add constraints to “tighten” the formulation. If you are using the LP/QP Solver, try the Aggressive setting for the PreProcessing, Cuts and Heuristics options in the Task Pane Engine tab.

16. Stop chosen when the max number of integer subproblems was reached.

If you are using the Large-Scale LP/QP Solver, Large-Scale SQP Solver, MOSEK Solver, XPRESS or Gurobi Solver on a problem with integer constraints, this message appears when the Solver has explored the maximum number of integer subproblems (each one is a “regular” Solver problem with additional bounds on the variables) allowed by the Max Subproblems option in the Task Pane Engine tab (in VBA and Solver SDK, the “MaxSubProblems” parameter), *and* in Excel, you clicked on the Stop button when the Solver displayed the Show Trial Solution dialog. You may increase the value of the Max Subproblems option, 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 well-formulated, and whether you can add constraints to “tighten” the formulation. If you are using the LP/QP Solver, try the Aggressive setting for the PreProcessing, Cuts and Heuristics options in the Task Pane Engine tab.

17. Solver converged in probability to a global solution.

If you are using the multistart method for global optimization with the Large-Scale GRG Solver, Large-Scale SQP Solver or Knitro Solver (by setting the Global Optimization options in the Task Pane Engine tab, or setting the “MultiStart” parameter in the Solver SDK), 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 (or returned by the Solver SDK) 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 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 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 the multistart method for global optimization with the Large-Scale GRG Solver, Large-Scale SQP Solver, or Knitro Solver, and you have set the Require Bounds on Variables option in the Task Pane Engine tab to True (it is True by default), this message will also appear. You should add the missing bounds and try again. You *must* define bounds on all variables in order to use the OptQuest Solver. For the Evolutionary Solver or multistart method, such bounds are not absolutely required – you can set the Require Bounds on Variables option to False – but they are a practical necessity if you want the Solver to find good solutions in a reasonable amount of time.

In Analytic Solver Comprehensive and Analytic Solver Optimization, you can use the Platform tab Decisions Vars Lower and Upper options to quickly set non-zero bounds on *all* decision variables. In VBA and Solver SDK, you specify individual bounds by setting the LowerBound and UpperBound properties of Variable objects, or by calling the SolverVarLowerBoundSet and SolverVarUpperBoundSet procedural API functions. To set default bounds on *all* variables, set the Problem. Model.Params"DefaultLowerBound" and "DefaultUpperBound" properties.

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 <= or >= 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 <= or >= constraint causes the conflict, remove it and try again.

In VBA and Solver SDK, you specify binary or alldifferent constraints by setting the IntegerType and GroupIndex properties of Variable objects, or by calling the SolverVarIntegerTypeSet and SolverVarGroupIndexSet procedural functions. You will receive result code 19 if these specifications conflict as described above.

20. Lower and upper bounds on variables allow no feasible solution.

This message appears if you've 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 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 problem.

In VBA and Solver SDK, you'll receive result code 20 if you set the LowerBound and UpperBound properties of a Variable object, or call the SolverVarLower-BoundSet and SolverVarUpperBoundSet procedural API functions, where a lower bound is greater than the corresponding upper bound.

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 is being used, and the PSI Interpreter encounters an error when computing derivatives via automatic differentiation. 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 this won't yield an error when you use the Excel Interpreter and the Solver is using finite differencing.

In Solver SDK, this result code appears if your Evaluator for derivatives returns a nonzero value to its caller. You'll have to examine your Evaluator to further diagnose the problem.

22. Variable appears in more than one cone constraint.

This message appears when you solve if the same decision variable appears in more than one cone constraint. You can define as many cone constraints as you want, but each one must constrain a different group of decision variables.

In VBA and Solver SDK, you specify cone constraints by setting the ConeType and ConeIndex properties of Variable objects, or by calling the SolverVarConeTypeSet and SolverVarConeIndexSet procedural API functions. You will receive result code 22 if these specifications conflict as described above.

23. Formula depends on uncertainties, must be summarized or transformed.

This message can appear in Analytic Solver Comprehensive and Analytic Solver Optimization, typically when you are first starting to build optimization models that include uncertainty – consider it part of the "learning experience." You'll be defining constraints or an objective, computed by formulas that depend on uncertain parameters: Each such formula represents an *array* of sample values, one for each realization of the uncertainties. For your model to be well-defined, the objective or constraint must either be *summarized* to a single value (such as a mean or percentile value) or *transformed* into a set of single-valued constraints.

To pinpoint the cell formula(s) where the problem occurs, follow the steps in "What Can Go Wrong, and What to Do About It" in the chapter "Getting Results: Stochastic Optimization" in the Analytic Solver User Guide. To correct the problem, you can (i) use the Ribbon or Task Pane to define the constraint as a *chance* constraint or the objective as an *expected value* or *risk measure* objective, or (ii) edit the formula so that its 'top level' value is computed by a PSI Statistics function such as PsiMean() or PsiPercentile(). It's important to understand *why* you need to summarize or transform a formula that depends on uncertainties: To learn more, read the chapter "Mastering Stochastic Optimization Concepts" in the Analytic Solver User Guide.

24. Excel Interpreter can only handle normal objective and constraints.

In Analytic Solver Comprehensive and Analytic Solver Optimization, this message appears when you solve if you've used the Task Pane Model tab to define a *chance* constraint or an *expected value* or *risk measure* objective, but you've selected the Excel Interpreter in the Task Pane Platform tab. (It doesn't occur in Solver SDK.) The *PSI Interpreter* can handle these types of constraints or objectives, without using PSI Statistics functions on the worksheet. But the Excel Interpreter cannot do this; you must use PSI Statistics functions to *summarize* the array of sample values represented by the objective and each constraint that depends on uncertainty. For example, you can compute a VaR-type chance constraint with a PsiPercentile() function. To learn more, read the chapter "Mastering Stochastic Optimization Concepts" in the Analytic Solver User Guide.

25. Simulation optimization doesn't handle models with recourse decisions.

In Analytic Solver Comprehensive and Analytic Solver Optimization, this message appears when you solve if you've used the Task Pane Model tab to define a recourse decision variable, but you've set the Solve Uncertain Models option to Simulation Optimization in the Task Pane Platform tab. In Solver SDK, it appears if you've set the ModelParam "SolveUncertain" property to 1, and your model includes recourse decision variables. Simulation optimization, as defined in the academic literature and as implemented in the Analytic Solver or Solver SDK products, doesn't support the concept of recourse decision variables. To solve a problem with recourse decisions, you'll need to set the Task Pane Platform tab options or ModelParam "SolveUncertain" property to use stochastic programming and robust optimization methods, both of which *do* support recourse decision variables (or use Automatic mode). To learn more, read the chapter "Mastering Stochastic Optimization Concepts" in the Analytic Solver User Guide.

26. Solver could not find a feasible solution to the robust chance constrained problem.

This message may appear when you solve a model with uncertainty and chance constraints using robust optimization. When you do this, the Solver transforms your original model with uncertainty into a *robust counterpart* model that is a conventional optimization problem without uncertainty.

This message means that the Solver could not find a feasible solution to the robust counterpart problem. It does not *necessarily* mean that there is no feasible solution to the original problem; the robust counterpart is an *approximation* to the problem defined by your chance constraints that may yield conservative solutions which *over-satisfy* the chance constraints.

In Analytic Solver Comprehensive and Analytic Solver Optimization, when this message appears, there *may* be an option to "Auto Adjust Chance Constraints" – a small white button containing a green arrow, as shown below for result code 27. Your simplest course of action is to select this option and click OK. The Solver will then re-solve the problem, automatically adjusting the sizes of robust optimization *uncertainty sets* created for the chance constraints, in an effort to find a feasible solution.

If you don't see the option "Auto Adjust Chance Constraints," this normally means that the automatic improvement algorithm has already been tried (possibly because the "Auto Adjust Chance Constraints" option is set in the Task Pane), and this algorithm was unable to find a feasible solution. In this case, you should proceed as described for result code 5, "Solver could not find a feasible solution:" Look for conflicting constraints, i.e. conditions that *cannot* be satisfied simultaneously, perhaps due to choosing the wrong relation (e.g. <= instead of >=) on an otherwise appropriate constraint.

27. Solver found a conservative solution to the robust chance constrained problem. All constraints are satisfied.

This message may appear when you solve a model with uncertainty and chance constraints using robust optimization. When you do this, the Solver transforms your original model with uncertainty into a *robust counterpart* model that is a conventional optimization model without uncertainty.

The message means that the Solver found an optimal solution to the robust counterpart model, but when this solution was tested against your *original* model (using

Monte Carlo simulation to test satisfaction of the chance constraints), the solution *over-satisfied* the chance constraints; this normally means that the solution is 'conservative' and the objective function value can be further improved.

In Analytic Solver Comprehensive and Analytic Solver Optimization when this message appears, there *may* be an option to "Auto Adjust Chance Constraints" – a small white button containing a green arrow, as shown below. Your simplest course of action is to select this option and click OK. The Solver will then re-solve the problem, automatically adjusting the sizes of robust optimization *uncertainty sets* created for the chance constraints, in an effort to improve the solution.

If you don't see the option "Auto Adjust Chance Constraints," this normally means that the automatic improvement algorithm has already been tried (possibly because the "Auto Adjust Chance Constraints" option is set in the Task Pane).

An alternative course of action is to *manually* adjust the Chance measures of selected chance constraints, and re-solve the problem. The automatic improvement algorithm uses general-purpose methods to find an improved solution; you may be able to do better by adjusting Chance measures based on your knowledge of the problem.

28. Solver has converged to the current solution of the robust chance constrained problem. All constraints are satisfied.

This message may appear when you solve a model with uncertainty and chance constraints using robust optimization, and you've set the "Auto Adjust Chance Constraints" option in the Task Pane Output tab to True, or you've previously used the "Auto Adjust Chance Constraints" option in the Task Pane Output tab. It means that the Solver has found the best 'improved solution' it can; the normal constraints are satisfied, and the chance constraints are satisfied to the Chance level that you specified.

This is usually a very good solution, but it does not rule out the possibility that you may be able to find an even better solution by manually adjusting Chance measures based on your knowledge of the problem, and re-solving.

29. Solver cannot improve the current solution of the robust chance constrained problem. All constraints are satisfied.

This message may appear when you solve a model with uncertainty and chance constraints using robust optimization, and you've set the "Auto Adjust Chance Constraints" option in the Task Pane Output tab to True, or you've previously used the "Auto Adjust Chance Constraints" option in the Task Pane Output tab. It means that the Solver could not find an improved solution that satisfies *all* of the chance constraints to the Chance level that you specified. Typically in this case, some of the chance constraints will be satisfied to the level you specified, or even *over*-satisfied, but others will be *under*-satisfied.

You may find that the solution is acceptable, but if the chance constraints *must* be satisfied to the level you specified, further work will be required. You may be able to find an improved solution by manually adjusting Chance measures based on your knowledge of the problem, and re-solving.

Note: For custom Solver Result Messages and result codes returned by the Interval Global Solver (which is available only in Analytic Solver Comprehensive and Analytic Solver Optimization at present), please consult "Interval Global Solver Result Messages" in the chapter "Solver Result Messages" in the Analytic Solver User Guide.

Large-Scale GRG Solver Result Messages

The Large-Scale GRG Solver engine can return the following standard Solver Result Messages and result codes listed earlier: -1, 0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21 and 22. It can also return the Solver engine-specific messages and result codes listed in this section. If you're using the object-oriented interface in VBA or Solver SDK, the result codes are returned by the Problem.Solver.OptimizeStatus property. If you're using the legacy VBA macro interface in Excel, the result codes are returned by the SolverSolve function.

1017. Insufficient memory for basis matrix. Problem is too big for LSGRG.

This message appears when there are too many nonzero entries in the sparse-matrix representation of the inverse of the basis matrix (used to find feasible solutions to the constraints). Although the problem may be within the limits of variables and constraints supported by the Large-Scale GRG Solver, the *density* of the matrix entries is too great for the this Solver to handle. If you receive this message or the corresponding result code, try the Large-Scale SQP Solver or Knitro Solver, which can usually handle much larger problems than the Large-Scale GRG Solver.

1040. Too many new nonzero Jacobian entries. Choose a different starting point.

The Large-Scale GRG Solver maintains a sparse representation of the Jacobian matrix (i.e. the matrix of partial derivatives of the objective and constraints with respect to the decision variables, used to determine search directions), in which zero partial derivatives are omitted. If the sparsity pattern of the Jacobian matrix is not known when the solution process starts, the Solver constructs the matrix based on the partial derivatives at the starting point you supply. Some partial derivatives that are zero at the starting point may become nonzero as the Solver moves to new trial points. The Large-Scale GRG Solver can accommodate a large, but limited number of these new nonzero partial derivatives; if the limit is exceeded, this message or result code is returned.

You may be able to work around this problem by restarting the Solver at a different point – for example at the ending point reached when this message appears. A better approach is to specify the sparsity pattern of the Jacobian matrix at the beginning of the solution process. In Analytic Solver Comprehensive and Analytic Solver Optimization, the PSI Interpreter can determine the sparsity pattern automatically. In Solver SDK, if you've written your own Evaluator for the objective and constraints, you can specify the sparsity pattern by setting the Model object AllGradDepend property, or by calling the SolverModAllGradDependSet procedural API function. This will ensure that the Solver will not encounter the limit on additional nonzero Jacobian entries.

2009. Variable bounds are conflicting.

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. Hence there can be no feasible solution. This very likely means that you have made a mistake by specifying <= when you want >= or vice versa. You should recheck the bounds you specified for the decision variables to ensure they are consistent and reflect your intentions.

Other Large-Scale GRG Solver Error Messages

On very difficult or ill-formed problems, the Large-Scale GRG Solver may display error messages of the form "LSGRG Solver: ResultCode = *errval*" where *errval* is an internal error code, greater than or equal to 1000. If you see an error message of this form, please contact Frontline Systems technical support.

Large-Scale SQP Solver Result Messages

The Large-Scale SQP Solver engine can return any of the standard Solver Result Messages and result codes listed earlier: -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21 and 22. (Result code 7 can be returned only if both of the options “Treat Constraints as Linear” and “Treat Objective as Linear” are selected.) If you’re using the object-oriented interface in VBA or Solver SDK, the result codes are returned by the Problem.Solver.OptimizeStatus property. If you’re using the legacy VBA macro interface in Excel, the result codes are returned by the SolverSolve function.

Knitro Solver Result Messages

The Knitro Solver engine can return the following standard Solver Result Messages and result codes listed earlier: -1, 0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21 and 22. It can also return the Solver engine-specific messages and result codes listed in this section. If you’re using the object-oriented interface in VBA or Solver SDK, the result codes are returned by the Problem.Solver.OptimizeStatus property. If you’re using the legacy VBA macro interface in Excel, the result codes are returned by the SolverSolve function.

1000. Unable to compute analytic 2nd derivatives. Choose a different Second Derivatives Option.

This message appears if you’ve chosen “Analytic 2nd Derivatives” for the Second Derivatives option (see “Knitro Solver Options” in the chapter “Solver Options”), but the PSI Interpreter is unable to compute analytic second derivatives for your problem functions. This may be due to use of a nonsmooth function, or a function whose first derivative is nonsmooth, in your objective or constraints. You may be able to proceed by choosing “Analytic 1st Derivatives” or “Finite Differences” for the Second Derivatives option. In Solver SDK, you do this by setting the EngineParam "SecondDerivatives" parameter. But you should also consider changing the formulas in your model to eliminate functions whose second derivatives are not defined.

1001. Unable to allocate enough memory for analytic 2nd derivatives. Choose a different Second Derivatives Option.

This message appears if you’ve chosen “Analytic 2nd Derivatives” for the Second Derivatives option (see “Knitro Solver Options” in the chapter “Solver Options”) but the Solver is unable to allocate enough memory to compute or store analytic 2nd derivative information (which can take a considerable amount of space). As a first step in Analytic Solver Comprehensive and Analytic Solver Optimization, in the Task Pane Platform tab, ensure that the Advanced group Use Internal Sparse Representation option is set to True. If this message persists, try using “Analytic 1st Derivatives” or “Finite Differences” for the Second Derivatives option. In Solver SDK, you do this by setting the EngineParam "SecondDerivatives" parameter.

Other Knitro Solver Error Messages

In very exceptional cases, the Knitro Solver may display error messages of the form “An internal error occurred…,” or return a result code greater than 1001 via the SolverSolve function, or the Problem.Solver.OptimizeStatus property, or the SolverOptimizeStatus API function in the Solver SDK. If you encounter such an error message or result code, please contact Frontline Systems.

MOSEK Solver Result Messages

The MOSEK Solver engine can return the following standard Solver Result Messages and result codes listed earlier: -1, 0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, 21 and 22. It can also return the Solver engine-specific messages and result codes listed in this section. If you're using the object-oriented interface in VBA or Solver SDK, the result codes are returned by the Problem.Solver.OptimizeStatus property. If you're using the legacy VBA macro interface in Excel, the result codes are returned by the SolverSolve function.

1001. MOSEK Solver requires Solve With Structure or Automatic.

This message appears if you attempt to solve a problem with the MOSEK Solver, and in the Task Pane Platform tab, the Optimization Model group Interpreter option is set to Excel Interpreter, or the Advanced group Supply Engine With option is set to Gradients. This Solver Engine uses diagnostic information from the PSI Interpreter that is available only if the Interpreter option is set to PSI Interpreter, and the Supply Engine With option is set to Structure, Convexity or Automatic. In Solver SDK, you must either load an Excel workbook and use the PSI Interpreter, or else provide diagnostic information by setting the Model object AllGradDepend property, or calling the SolverModAllGradDependSet procedural API function, to use this Solver Engine.

1002, 1293, 1234, 1295, 1296. The problem cannot be solved and is likely nonconvex.

This message appears if the MOSEK Solver algorithms detect numerical error conditions that make it impossible to proceed with the solution process. In most cases, this means that one or more problem functions are non-convex. The MOSEK Solver can be used only on smooth convex (linear, quadratic, QCP or SOCP) problems.

You can try to solve the problem with the Large-Scale GRG, Large-Scale SQP or Knitro Solvers, all of which are designed to handle non-convex problems. But you should also consider why some of your problem functions may be non-convex. In Analytic Solver Comprehensive and Analytic Solver Optimization, you may gain further information by selecting **Optimize – Analyze Without Solving** (ensure that the Diagnosis group Intended Model Type option is set to Linear, and the Advanced group Supply Engine With option is set to Convexity). Then select **Reports – Optimization – Structure** and examine the report, looking at the rightmost column which gives information on the convexity of individual problem functions.

Gurobi Solver Result Messages

The Gurobi Solver can return the following standard Solver Result Messages and result codes listed in the last section: -1, 0, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, and 21. It can also return the Solver engine-specific message and result code listed in this section. If you're using the object-oriented interface, the result codes are returned by the Problem.Solver.OptimizeStatus property. If you're using the legacy VBA macro interface in Excel, the result codes are returned by the SolverSolve function.

In addition to Solver Result message, additional details can be found in the Output tab of the Solver Task Pane. The error message, "Unsupported Gurobi [operation] at [cell/line] may appear when an error has occurred during the solution processes. The term, [operation], can refer to one of the following unsupported functions (ACOS, ASIN, ATAN, ACOT), or "array/nested forms" which can be caused by an unsupported array formula, an unsupported “spill” or an excessively long nested/cascading formula.

Note: The Gurobi Solver's NLP functionality does not support array formulas, i.e. those where a formula is applied to a column range computing an array (rather than a scalar). In other words, it does not support functions that result in "spills." This limitation does not apply when solving linear or quadratic models.

1001. Integer Cutoff makes problem infeasible.

This message appears if you entered an inappropriate value in the Integer Cutoff option in the Task Pane Engine tab Integer group (i.e. a value larger than the optimal objective if maximizing, or smaller than the optimal objective if minimizing).

1002. Model is unbounded or infeasible.

This message appears when the Gurobi Solver determines, in its *presolve* phase, that the model is either unbounded or infeasible. Because the main optimization process is never started in this case, the Solver Engine cannot return result codes 4 or 5.

1004. Solver ran into numerical difficulties.

This message is very rare, but it can appear in situations where the problem (represented by the LP coefficient matrix and bounds on the variables and constraints) is numerically unstable, and the Gurobi Solver’s sophisticated methods for overcoming this instability are not sufficient. If you encounter this error message or result code, please contact Frontline Systems Technical Support (support@solver.com).

1005. Unable to satisfy optimality tolerances; a sub-optimal solution is available.

This message appears – typically for a model that is numerically unstable – when the Gurobi Solver has come “close” to a solution, but is unable to satisfy its optimality tolerances. Final values of the objective and variables are available in the usual way.

This result may also occur when solving a nonlinear model with the Gurobi Solver. It typically indicates the presence of an unsupported x^y expression. Such an expression is only valid when either x or y is constant, or if presolve can determine that one of them is constant. The following forms are allowed, where b is a constant:

- x^b
- b^x

As a workaround, the change of base rule can be applied: $x^y = b^{y \log_b(x)}$, though this does require x > 0.

1006. Quadratic objective is not positive semi-definite.

This message appears when you are solving a quadratic programming problem, and the Hessian of the quadratic objective is not strictly positive semi-definite. When this is true, the Gurobi Solver cannot find a solution because the problem is most likely nonconvex. The Gurobi Solver tries to add small perturbations to correct for small positive semi-definite violations. For your model, the required perturbations are larger than the required tolerance that is controlled by the ‘PSD Tolerance’ parameter.

10001. Out of memory.

This message can sometimes appear for very large, difficult LP/MIP problems. It means that the Branch & Bound tree has exhausted all available main memory. Your best immediate course of action is to set the Task Pane Engine tab MIP group **Node File Start** option to a low value such as 1 (for 1GB of memory), or even a fractional value. This will cause the Gurobi Solver to write the Branch & Bound tree to a disk file (the “Node File”) once it reaches 1GB in size. This should allow you to solve the problem, at some cost in extra solution time. But it may be a good idea to upgrade your PC with more RAM to solve such problems.

XPRESS Solver Result Messages

The XPRESS Solver can return the following standard Solver Result Messages and result codes listed in the last section: -1, 0, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, and 21. It can also return the Solver engine-specific message and result code listed in this section. If you’re using the object-oriented interface, the result codes are returned by the Problem.Solver.OptimizeStatus property. If you’re using the legacy VBA macro interface in Excel, the result codes are returned by the SolverSolve function.

1001. Objective worse than Cutoff.

This message appears if (i) the Dual Simplex method (or the Default method, which is Dual Simplex) is selected for the Algorithm to Use option in the Task Pane Engine tab General group, and (ii) you entered an inappropriate value in the Integer Cutoff option in the Integer group (i.e. a value larger than the optimal objective if maximizing, or smaller than the optimal objective if minimizing).

This message appears only if the LP *relaxation* of the mixed-integer problem yields an objective worse than the value entered for the Integer Cutoff option. On mixed-integer programming problems, if the objective of the relaxation is better than the Integer Cutoff value, then the Branch & Bound method will proceed with the search for integer solutions, cutting off the search on any node of the B&B tree whose objective is worse than the Integer Cutoff. So, if the Integer Cutoff value does not “cut off” the solution of the *relaxation*, but it does “cut off” the search that would have yielded the optimal *integer* solution, you will instead receive the message “Solver could not find a feasible solution.”

Also bear in mind that this message will appear only if the Dual Simplex method is used; if the Primal Simplex or Newton Barrier method is used, and the Integer Cutoff value is inappropriate, you will always receive the message “Solver could not find a feasible solution.” Hence, you should take care to ensure that the value you enter for the Integer Cutoff option is no better than the objective of a *known integer solution* to the *current problem.*

1002. Global search incomplete – no integer solution was found.

This message appears if you are solving a mixed-integer quadratic programming problem, and the Hessian of the quadratic objective is not strictly positive definite. When this is true, the QP subproblems solved as part of the Branch & Bound process may not have *unique* optimal solutions, and hence the Branch & Bound search cannot guarantee an exhaustive evaluation of all possible integer solutions. If no integer feasible solution has been found by the time the search is terminated, this message appears; otherwise the following message appears.

1003. Global search incomplete – an integer solution was found.

This message appears if you are solving a mixed-integer quadratic programming problem, and the Hessian of the quadratic objective is not strictly positive definite. When this is true, the QP subproblems solved as part of the Branch & Bound process may not have *unique* optimal solutions, and hence the Branch & Bound search cannot guarantee an exhaustive evaluation of all possible integer solutions. If at least one integer feasible solution has been found by the time the search is terminated, the best such solution is returned by the Solver, and this message appears.

1004. Quadratic Objective is not positive definite.

This message appears if you are solving a quadratic programming problem, and the Hessian of the quadratic objective is not strictly positive definite. When this is true, the XPRESS Solver cannot guarantee an optimal solution. If the quadratic objective is positive semidefinite, the Solver will normally find one of many solutions with the same best objective value. If the quadratic objective is indefinite, the Solver may not find an optimal solution at all (this is a difficult global optimization problem).

1005. Model does not appear to be convex.

This message appears when you are solving a quadratic programming problem, and the quadratic objective function is not convex. You can turn off the built-in convexity checking algorithm for the Xpress Solver by setting the 'Check Convexity' parameter to False, and try again. With this change, since your problem is not convex, the Xpress Solver might not find an optimal solution at all since this is a difficult global optimization problem.

OptQuest Solver Result Messages

The OptQuest Solver engine can return the following standard Solver Result Messages and result codes listed in the last section: -1, 0, 3, 5, 6, 8, 9, 10, 11, 12, 13, 18, 19, 20, 21 and 22. It can also return the Solver engine-specific message and result code listed in this section. If you're using the object-oriented interface in VBA or Solver SDK, the result codes are returned by the Problem.Solver.OptimizeStatus property. If you're using the legacy VBA macro interface in Excel, the result codes are returned by the SolverSolve function.

1001. Multiple groups of "alldifferent" variables.

This message appears if your model includes more than one "alldifferent" constraint, which defines a group of integer variables that are required to be all different at the solution. The OptQuest Solver engine currently supports only one such group per problem. To use the OptQuest Solver, you must re-formulate your model so that only one "alldifferent" constraint is defined.

5002. Optimal solution found.

This message appears when the OptQuest Solver was able to enumerate and evaluate all possible solutions to your model. The OptQuest Solver has systematically explored the feasible solution space and found the globally optimal solution. There is no other solution satisfying the constraints that has a better value for the objective.

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

The Large-Scale LP/QP Solver, Gurobi Solver, Large-Scale GRG Solver, Large-Scale SQP Solver, and Knitro Solver all offer a Use Automatic Scaling option. The MOSEK Solver offers four choices: No automatic scaling, conservative scaling, aggressive scaling, and automatic determination of scaling. The XPRESS Solver offers a wide range of options for automatic scaling of your model, in the Scaling option group on the Task Pane Engine tab. When you use these options, 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. *Note however that calculations performed by your formulas in spreadsheet cells, or by expressions or statements in your custom program, with values of very different magnitudes, may result in loss of accuracy that is outside the control of the Solver.*

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 Excel worksheet or in your custom program may range from perhaps 1 to 1,000.

If you are using Analytic Solver Comprehensive and Analytic Solver Optimization 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 a **Scaling Report** when available from **Reports – Optimization** on the Ribbon. If you are using Solver SDK with an Evaluator written in code, you’ll have to analyze by hand your program statements that calculate values for the objective and constraints for poorly scaled results.

Because automatic scaling will compensate for large differences in the magnitudes of the objective and constraints, we recommend that you use Automatic Scaling for most of your Solver problems. In some cases, however, *no scaling* may yield better results than automatic scaling. If you receive one of the Solver Result Messages mentioned above, you may wish to try solving the model with no scaling.

The Integer Tolerance Option and Integer Constraints

When you solve a mixed-integer programming problem (any problem with integer constraints) using the Large-Scale LP/QP Solver, Large-Scale GRG Solver, Large-Scale SQP Solver, Knitro Solver, Gurobi Solver, or MOSEK Solver, all of which employ the Branch & Bound method, the solution process is governed by the Integer Tolerance option on the Task Pane Engine tab. Since the *default setting of the Integer Tolerance option is non-zero* for some Solver Engines, the Solver stops when it has found a solution satisfying the integer constraints whose objective is *within the tolerance* of the true integer optimal solution. Therefore, you may know of or discover an integer solution that is “better” than the one found by the Solver.

The reason the default setting of the Integer Tolerance option may be non-zero 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 returned by the Solver is close to the true integer 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 Analytic Solver Comprehensive and Analytic Solver Optimization you can do this on the Task Pane Engine tab. In VBA and Solver SDK, you do this by setting the value of the “IntTolerance” parameter.

Limitations on Non-Convex Optimization

As discussed in the chapter “Mastering Conventional Optimization Concepts” in the Analytic Solver User Guide, non-convex problems are far more difficult to solve than convex (linear, quadratic or nonlinear) problems, and there are fewer guarantees about what the Solver can do. This section refers to the Large-Scale GRG Solver as LSGRG, and to the Large-Scale SQP Solver as LSSQP.

How do you know whether your nonlinear model is convex or non-convex? With other software, your only choice is to use the mathematical properties of your problem functions to attempt to prove convexity or non-convexity. Few users have the time or the background to do this. But anyone can use the automatic convexity test in Analytic Solver Comprehensive and Analytic Solver Optimization by simply selecting **Optimize – Analyze Without Solving** from the Ribbon.

The convexity test in Analytic Solver Comprehensive and Analytic Solver Optimization does not always return a definitive answer – the test would require time *exponential* in the number of variables and constraints to guarantee such an answer. If the convexity test does not return a definitive answer, you should assume that your model is non-convex, and follow the advice in this section, unless you do have the time and background to develop an analytic proof of convexity.

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.” You can do this manually, or you can set the global optimization options for the LSGRG, LSSQP or Knitro Solver, to activate the multistart method which will automatically run the Solver from multiple starting points.

Large-Scale GRG, SQP, and Knitro Solver Stopping Conditions

It is helpful to understand how the optimization methods used in the LSGRG, LSSQP and Knitro Solvers behave on a non-convex model, and what each of the possible Solver Result Messages means for these Solver engines. At best, the LSGRG, LSSQP or Knitro Solver alone – like virtually all “classical” nonlinear optimization algorithms – can find a *locally optimal* solution to a reasonably *well-scaled* 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 LSGRG, LSSQP or Knitro Solver has found a *locally optimal* solution – there is no other set of values for the decision variables close to the current values which 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 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 Task Pane Engine tab, or the "Precision" parameter in VBA and Solver SDK).

When Solver has Converged to the Current Solution

When the LSGRG, LSSQP, or Knitro 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 best solution found so far has changed very little for the last few iterations or trial solutions. In the LSGRG and LSSQP Solvers, the "change in the solution" is compared to the value of the Convergence option in the Task Pane Engine tab, or the "Convergence" parameter in VBA and Solver SDK. The LSGRG Solver stops if the *relative change in the objective function value* is less than the Convergence value; the LSSQP Solver stops if the *maximum normalized complementarity gap of the variables* is less than the Convergence value. For most problems, the default Convergence value (which is different for the two Solvers) is appropriate. For more information, see the discussion of the Convergence option for each Solver in the chapter "Solver Options." The Knitro Solver uses slightly different criteria to determine when to stop due to slow progress, but the effect is much the same.

A *poorly scaled* model is more likely to trigger this stopping condition, even if the Use Automatic Scaling option is set to True. So it pays to design your model to be reasonably well scaled in the first place: The typical values of the objective and constraints should not differ from each other, or from the decision variable values, by more than three or four orders of magnitude.

If you are getting this message when you are seeking a locally optimal solution, you can change the setting of the Convergence option 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 Solver has encountered numerical accuracy or stability problems in optimizing the model, and it has tried all available methods to overcome the numerical problems, but cannot reach an optimal solution. The issues involved are beyond the level of this User Guide, as well as most of the books recommended in the Analytic Solver User Guide. One possibility worth checking is that some of your constraints are redundant, and should be removed. To analyze problems of this nature, you may need specialized consulting assistance.

Limitations on Global Optimization

Nonlinear optimization problems in general, and non-convex global optimization problems in particular, can be very challenging for any solution method, for several reasons. In such problems, there may be multiple feasible regions and many locally optimal solutions, with little analytic information about the problem functions available to guide a Solver towards feasible, better or optimal solutions. (In fact, just deciding whether or not a non-convex problem has any feasible solutions is itself a global optimization problem.) Because of these properties, the “classical” optimization methods used by the standard GRG Solver, the Large-Scale GRG Solver, the Large-Scale SQP Solver, and the Knitro Solver cannot guarantee that a globally optimal solution, or even a feasible solution, if one exists, will be found.

The multistart methods for global optimization included in Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK offer a better chance of finding a globally optimal solution, or at least a feasible solution, but with few or no guarantees. Yet these methods prove useful because they typically find *better* solutions than the classical methods alone, even if they cannot prove global optimality. Some of the capabilities and limitations of these solution methods are discussed below.

Problems that include non-smooth functions are even more difficult, and the available solution methods offer still fewer guarantees – as discussed in the next section, “Limitations on Non-Smooth Optimization.”

Multistart Search with the Large-Scale GRG, SQP and Knitro Solvers

The multistart methods for global optimization included in Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK can overcome some of the limitations of the LSGRG, LSSQP and Knitro Solvers alone, but they are not a panacea. The multistart methods will automatically run the LSGRG, LSSQP or Knitro Solver from a number of starting points and will return 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 more time you allow, the better the coverage.

However, the performance of the multistart methods is generally limited by the performance of the LSGRG, LSSQP or Knitro Solver on the subproblems. If the Solver stops prematurely due to slow convergence, or fails to find a feasible point on a given run, the multistart methods can improve upon this only by finding another starting point from which the Solver *can* find a feasible solution, or a better locally optimal solution, by following a different path into the same region. If the LSGRG, LSSQP, or Knitro Solver is having difficulty with the subproblems, you may wish to try the LSSQP Evolutionary or OptQuest Solver as another alternative.

If the LSGRG, LSSQP or Knitro Solver reaches the same locally optimal solution on many different runs initiated by the multistart methods, this will tend to decrease the Bayesian estimate of the number of locally optimal solutions in the problem, causing the multistart methods to stop relatively quickly. In many cases this indicates that the globally optimal solution has been found – but you should always inspect and think about the solution, and consider whether you should run the Solver manually from starting points selected from your knowledge of the problem.

Large-Scale GRG, SQP and Knitro Solvers 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 LSGRG, LSSQP and Knitro Solvers on the subproblems. If the 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 LSGRG, LSSQP or Knitro 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, you may wish to try the OptQuest Solver, or ask Frontline Systems about other alternatives.

Determining infeasibility of a non-convex problem is especially difficult for an interior point nonlinear method. If you find that the Knitro Solver is taking a very long time on your problem, without finding a feasible solution, you should set the **Solution Method** option to **Active Set** on the Task Pane Engine tab, and try solving the problem again.

Limitations on Non-Smooth Optimization

As discussed in the chapter “Mastering Conventional Optimization Concepts” in the Analytic Solver User Guide, non-smooth problems – where the objective or constraints are computed with discontinuous or non-smooth 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 used in Excel spreadsheets is the IF function, where the conditional test is dependent on the decision variables. Common non-smooth functions in Excel are ABS, MIN and MAX. Analytic Solver Comprehensive and Analytic Solver Optimization can often help in such cases – they can automatically transform a model containing these functions, replacing them with additional integer and continuous variables and linear constraints, to create an ‘equivalent’ model whose solution is the same as your original model. If the resulting model contains only linear constraints, then a powerful LP/MIP Solver, such as the Large-Scale LP/QP Solver, Gurobi Solver or XPRESS Solver, can be used to solve the problem. This is further described in the Analytic Solver User Guide.

Other common discontinuous functions in Excel are CHOOSE, the LOOKUP functions, COUNT, INT, ROUND, 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 includes non-smooth functions, your simplest course of action is to use the OptQuest Solver engine or the LSSQP Solver engine with its integrated Evolutionary Solver to find a “good” solution. You can try using the LSGRG Solver, the LSSQP Solver alone, or the Knitro Solver on problems of this type, but you should be aware of the effects of non-smooth functions on these Solver engines, as summarized below.

You *can* use discontinuous functions such as IF and CHOOSE in calculations 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, are non-smooth. For more information on this subject, read the section “Non-Smooth Functions” in the chapter “Mastering Conventional Optimization Concepts” in the Analytic Solver User Guide.

Effect on the Large-Scale GRG, SQP, and Knitro Solvers

A smooth nonlinear solver, such as the LSGRG, LSSQP or Knitro 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 LSGRG, LSSQP and Knitro Solvers can sometimes deal with discontinuous or non-smooth functions which are “incidental” to the problem, but as a general statement, these Solver engines require that all problem functions are smooth nonlinear.

If you are using Analytic Solver Comprehensive and Analytic Solver Optimization with default settings, the Polymorphic Spreadsheet Interpreter will compute derivatives of the problem functions using *automatic differentiation*. By default, the Interpreter will in fact compute ‘point values’ for derivatives for a special set of commonly used non-smooth functions – ABS, IF, MAX, MIN and SIGN – since derivatives are defined at *most* points for these functions. This enables the LSGRG, LSSQP or Knitro Solver to complete the solution process in many cases.

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 LSGRG, LSSQP or Knitro Solver, you’ll likely receive the message “Solver encountered an error computing derivatives.” You can still solve your model with the Optimization Model group Interpreter option set to Excel Interpreter – but this means that the Solver will compute estimates of the derivatives via finite differencing, and all of the caveats noted above on the Solver’s ability to find a solution will still apply.

The LSSQP Solver with its integrated Evolutionary Solver offers a better way to attack large problems with some non-smooth functions. This Solver uses genetic algorithm methods, focusing on the non-smooth variables and functions, to find good starting points for local searches performed using powerful SQP methods; it seeks to minimize the effects of non-smooth functions on the SQP-based local searches. This combination is often effective, but it can offer few guarantees of optimal solutions.

If you’re using Solver SDK, and you’ve written an Evaluator that the Solver can call to obtain derivatives of the objective and constraints, your situation is similar to that of the Interpreter in Analytic Solver Comprehensive and Analytic Solver Optimization: Since the derivative of a non-smooth function is undefined at certain points, your Evaluator must either return a nonzero value to signal an error – thereby terminating the solution process – or else it must “come up with” approximate values for the derivatives. If you haven’t supplied an Evaluator for derivatives, but you *have* written an Evaluator for function values, the Solver will call your Evaluator many times with perturbed values for the decision variables, and compute estimates of the derivatives via finite differencing. Again, all of the caveats above on the Solver’s ability to find a solution will apply.

If you’re using the Knitro Solver with its Second Derivatives option set to “Analytic 2nd Derivatives” (the default – see “Knitro Solver Options” in the chapter “Solver Engine Options”), you may receive the Solver Result message “Unable to compute analytic 2nd derivatives.” If all of your problem functions are smooth, this message implies that some of your functions have non-smooth first derivatives, i.e. points where their second derivatives are not defined. You should either modify your model, or choose “Analytic 1st Derivatives” or “Finite Differences” for the Second Derivatives option.

OptQuest Solver Solutions and Stopping Conditions

The OptQuest Solver is designed to deal with problems involving non-smooth or discontinuous functions. However, it is important to understand what the OptQuest Solver can and cannot do, and what each of the possible Solver Result Messages means for this Solver engine. Because the OptQuest Solver does not rely on derivative or gradient information, *it cannot determine whether a given solution is optimal* – except in rare cases where all variables are integer and bounded, and it can prove optimality by enumerating all possible solutions. Hence, the OptQuest Solver is designed to seek continuously improved solutions – it stops and returns a solution only if you press ESC or (in the SDK) your Evaluator returns the “user abort” code, or when it exceeds one of the limits you’ve set via its Solver Engine options: Max Time, Iterations, or Stop When Objective Hasn’t Improved.

“Good” Versus Optimal Solutions

The OptQuest 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 cannot perform mathematical tests for optimality; it knows only that a solution is “better” in comparison to other solutions found earlier. It may sometimes find the true optimal solution, but in most cases, it will not be able to *tell* you that this solution is truly optimal.

Evaluating a Solution Found by the OptQuest Solver

Once you have a solution from the OptQuest Solver, what can you do with it? Here are some ideas:

1. Allow a different sequence of random numbers to be used (via the Task Pane Engine tab, or via the “RandomSeed” parameter in VBA and Solver SDK), and run the OptQuest Solver again, to see if it finds an even better solution in a reasonable length of time.
2. If you are watching the progress of the OptQuest Solver and the chart of the objective in the Task Pane Output tab, you can press ESC and click the Restart button, at any time. In Solver SDK, your Evaluator for iterations can examine the current Trial Solution and return the “restart” code to accomplish the same thing.
3. Set the “Number of Solutions to Report” in the Task Pane Engine tab to a value greater than 1, solve the problem again, and examine the Solutions Report to see a set of alternative solutions. In VBA and Solver SDK, you can achieve the same effect by setting OptQuest-specific parameters.
4. Adjust the Precision (Obj Fun) and Precision (Dec Var) settings in the Task Pane Engine tab, and solve the problem again. In VBA and Solver SDK, you can do this by setting OptQuest-specific parameters. These and other settings are discussed in the next chapter, “Solver Engine Options.”
5. Keep the resulting solution, switch to the GRG Nonlinear, LSGRG, LSSQP, or Knitro Solver and start it from that solution, and see if it finds the same or a better solution. If the nonlinear Solver displays the message “Solver found a solution” (or returns result code 0) you may have found at least a *locally optimal* point – but remember that this test assumes smoothness of problem functions.

6. Select and examine the Population Report (in Solver SDK, call the equivalent API functions). If the Best Values are similar from run to run of the OptQuest 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.

Solver Engine Options

This chapter describes the options available for the Large-Scale LP/QP Solver, Large-Scale GRG Solver, Large-Scale SQP Solver, Knitro Solver, MOSEK Solver, Gurobi Solver, XPRESS Solver, and OptQuest Solver. In Analytic Solver Comprehensive and Analytic Solver Optimization options may be examined or set interactively via the Task Pane Engine tab, or programmatically using either the **new object-oriented API** described below, or the **traditional VBA functions** described in the later chapter “Programming the Solver Engines.” In Solver SDK, options may be examined or set via the **new object-oriented API** or the **SDK procedural API**; both are described below.

Bear in mind that the options that control numerical tolerances and solution strategies are pre-set to the choices that are applicable to the great majority of problems; you should only change these settings in exceptional circumstances. 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 Analytic Solver Comprehensive and Analytic Solver Optimization, you can examine or set Solver Engine options using the object-oriented API described in this section. In Solver SDK, you can examine or set these options using either the object-oriented API or the SDK procedural API. In both APIs, 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 Max Time 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. (Since Java currently lacks properties, the syntax used by Solver SDK is slightly different.) For example, you can set the Max Time limit for the currently selected Solver Engine to 1000 seconds by writing:

VBA / 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);`

To get the current Max Time 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
```

SDK Procedural API

In the SDK procedural API, Solver Engine options and parameters are handled in a manner similar to the object-oriented API, but you use procedural function calls to get and set parameter values. You can set the Max Time limit for the currently selected Solver Engine to 1000 seconds by writing:

C/C++: `SolverEngParamSet(myProb, "MaxTime", 1000);`

Matlab: `SolverSDK('EngineParamSet',myProb,'MaxTime',1000);`

To access all of the options and parameters supported by a Solver Engine, you would write in C/C++:

```
LPCWSTR myEngine[100], myParam[100];
double myValue; int i, myCount;
SolverProbEngineGet (myProb, &myEngine);
SolverEngParamCount (myProb, myEngine, &myCount);
for (i = 0; i < myCount; i++)
{
   SolverEngParamName (myProb, myEngine, i, &myParam);
   SolverEngParamGet (myProb, myParam, &myValue);
   wprintf (L"%s = %g\n", myParam, myValue);
}
```

In C++ and Matlab, you can use either the object-oriented API or the procedural API, but most users find the object-oriented API more convenient.

Common Solver Options

The options described in this section are common to the LP/Quadratic Solver, nonlinear GRG Solver, Evolutionary Solver, and SOCP Barrier Solver in Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK, and to the Large-Scale LP/QP Solver, Large-Scale GRG Solver, Large-Scale SQP Solver, Knitro Solver, MOSEK Solver, and Gurobi Solver Engines. Some of these options are also used by the OptQuest Solver, as described in a later section.

Max Time

VBA / SDK: Engine.Params("MaxTime"), integer value > 0

The value of the Max Time option 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. By default, this option is blank, which means there is no time limit.

Iterations

VBA / SDK: Engine.Params(“Iterations"), integer value > 0

The value of the Iterations option 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. When this option is empty/blank (the default), there is no limit on the number of iterations.

Whether or not you set a Max Time limit or Iteration limit, you can press the ESC key or click the Pause/Stop button at the top left of the Task Pane while the Solver is running. After a momentary delay, the Show Trial Solution dialog box will appear, and you will have the option to stop at that point or continue or restart the solution process.

In Solver SDK, you can achieve the same effect by writing an Evaluator method that is called on each iteration or Trial Solution. In this method, you can check for a press of the ESC key (or some other condition) and then display a dialog, asking the user whether the solution process should continue or stop.

Precision

VBA / SDK: Engine.Params("Precision")

Gurobi, Knitro, Large-Scale GRG, Large-Scale LP, Large-Scale SQP, OptQuest and Xpress Engines: 0.00000001<= Precision <= 0.0001

Mosek Engine: 0 <= Precision <= 1

Use this option to determine 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. 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 >= 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 setting the Use Automatic Scaling option to True instead of altering the Precision setting. Generally, this setting should be from 1.0E-6 (0.000001) to 1.0E-4 (0.0001); for the Large-Scale SQP Solver and the Knitro Solver, a value of 1.0E-6 is highly recommended.

Precision and Integer Constraints

Another use of the Precision value 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, the variable value is treated as an integer. (The Gurobi Solver uses a different option named “IntegralityTol” for this purpose.)

Precision and Automatic Scaling

When the Use Automatic Scaling option is True, the Precision value is compared to the internally scaled values of the variables and constraints. (In the Large-Scale SQP Solver, the Precision value is *always* compared to the normalized constraint violations.) This makes the comparison relative rather than absolute, which is a further reason not to make the Precision value much tighter than 1.0E-6 (0.000001).

Assume Non-Negative

VBA: Engine.Params("AssumeNonneg"), value 1/True or 0/False
SDK: Use Variable object NonNegative method or SolverVarNonNegative function

Set this option to True to cause any decision variables that are not given explicit lower bounds (via >=, binary, or alldifferent constraints in the Task Pane Model tab) to be given a lower bound of zero when the problem is solved. This option has no effect for decision variables that do have explicit >= constraints, even if those constraints allow the variables to assume negative values.

In Analytic Solver Comprehensive and Analytic Solver Optimization, an alternative to this option is to use the Platform tab Decision Vars Lower option to set any lower bound you wish (not just zero) on any decision variables that are not given explicit lower bounds.

Use Automatic Scaling

VBA / SDK: Engine.Params("Scaling"), value 0 or 1/True or -1/False

Set this option to True to cause the Solver to re-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 “Solver Result Messages.”

If your model is nonlinear and you set this option to True, *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 Use Automatic Scaling option depends on how well these starting values reflect the values encountered during the solution process.

Show Iteration Results

VBA: Engine.Params("StepThru"), value 1/True or 0/False
SDK: Define an Evaluator for Eval_Type_Iteration

Set this option to True if you want to show the results of every major iteration or Trial Solution on the spreadsheet. When you solve, you'll see the same Show Trial Solution dialog that appears when you press ESC or click the Pause/Stop button at any time during the solution process; but when this option is True, the dialog 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.

Bypass Solver Reports

VBA: Engine.Params("BypassReports"), value 1/True or 0/False
SDK: Not Applicable

Set this option to True to save time during the solution process if you do not need the reports for the current solution run. When this option is set to False (the default), the Solver always performs extra computations to prepare for the possibility that you will select one or more reports from the Ribbon. When this option is set to True, the extra computations are skipped; the gallery of Optimization Reports will be greyed out, and you won't be able to select any reports for this run.

Even though report generation in Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK 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. When you have 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.

Max Subproblems

VBA / SDK: Parameter Name "MaxSubProblems", integer value > 0

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. Use this option to place 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 value of this option is blank by default, meaning there is no limit on the number of subproblems.

In a problem with integer constraints, the Max Subproblems limit should be used in preference to the Iterations limit; the Iterations limit should be left blank (unlimited), or set high enough for each of the individual subproblems solved during the Branch & Bound process.

Max Integer Solutions

VBA / SDK: Parameter Name "MaxIntegerSols", integer value > 0

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. Use this option to place a limit on the number of feasible integer solutions that are found by the Branch & Bound algorithm before the Solver **pauses** and asks you whether to continue or stop the solution process. The value of this option is blank by default, meaning there is no limit on the number of feasible integer solutions.

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.

The Solver retains the feasible integer solution with the best objective value so far, called the "incumbent." The objective value of this solution is shown in the Progress area of the Task Pane Output tab.

Integer Tolerance

VBA / SDK: Parameter Name "IntTolerance", 0 <= value <= 1

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. It is sometimes called the "MIP gap."

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 its proceeds, the B&B method also tightens the "best bound" on how good the integer solution can be.

Each time the Solver finds a new incumbent – an improved all-integer solution – it computes the maximum percentage difference between the objective of this solution and the current best bound on the objective:

(Objective of incumbent **-** Objective of best bound) / 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. 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 < 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 set the Integer Cutoff option to this value. 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

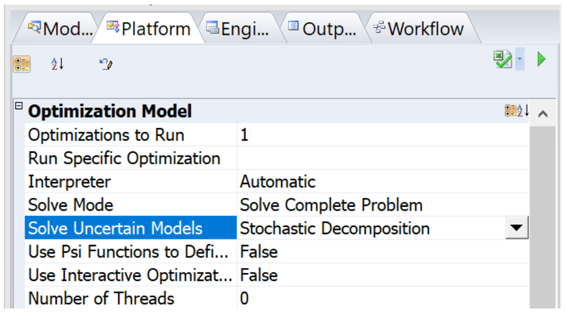
VBA / SDK: Engine.Params("SolveWithout"), value 1-Solve Without Integer Constraints or 0-Solve *With* Integer Constraints

This option is supported by the Solver Engines, but it's preferable to use the Platform 'Solve Mode' option set to Solve Without Integer Constraints (in VBA and Solver SDK use Model.Params("SolveMode") = 2) instead.

If you solve a model (by clicking the Optimize button on the Ribbon or the green arrow in the Task Pane, or by calling Problem.Solver.Optimize in VBA or Solver SDK) with this option setting, 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's much more convenient to set this option than to delete the integer constraints and add them back again later.

Stochastic Decomposition Options

The Stochastic Decomposition section of this dialog contains all of the options which are specific to the Stochastic Decomposition method bundled within Analytic Solver Comprehensive and Analytic Solver Optimization (Analytic Solver Desktop only) and utilized in the Large Scale LP Solver Engine, the Gurobi Solver Engine, and the Xpress Solver Engine. Stochastic Decomposition can be used to solve linear models with recourse variables and uncertainty in the constraints only. (Model must contain at least one constraint that does not include uncertainty.) To run Stochastic Decomposition, set Solve Uncertain Models to Stochastic Decomposition on the Task Pane Platform tab as shown in the screenshot below. Typically, Stochastic Decomposition should only be used if the internal model created when Deterministic Equivalent is selected for Solve Uncertain Models is so large that the time and memory required to solve the model is impractical.



Optimization Model

Optimizations to Run	1
Run Specific Optimization	
Interpreter	Automatic
Solve Mode	Solve Complete Problem
Solve Uncertain Models	Stochastic Decomposition
Use Psi Functions to Defi...	False
Use Interactive Optimizat...	False
Number of Threads	0

Tau

VBA / SDK: Engine.Params("StochTau"), 1 <= integer value

As interations proceed, the cut which was formed based on the incumbent vector is periodically re-evaluated. If tau iterations have passed since the last update or if the value of the incumbent is less than the objective value at a specific iteration, the incumbent is reformed. This function will solve an additional subproblem (based upon the incumbent and the most recent observation of omega) add the dual solution to the data structures, and re-form the incumbent cut (thus replacing it in the array of cuts).

Tolerance

VBA / SDK: Engine.Params("StochTol"), 0 <= value <= 0.001

The tolerance setting is used to determine the stopping condition for the Stochastic Decomposition method. A larger value will result in the Stochastic Decomposition methods stopping more quickly, while a smaller value will force the Stochastic Decomposition methods to run longer.

Compute Confidence Interval

VBA / SDK: Engine.Params("StochCI"), 0/False 1/True

If this option is set to true, the stochastic decomposition methods will compute a 95% confidence interval for the objective function.

Objective Error

VBA / SDK: Engine.Params("StochErr"), 0 <= value <= 1

If Compute Confidence Interval is set to true, then the confidence interval for the objective function will be accurate to within this value, with 95% confidence.

Objective Improvement

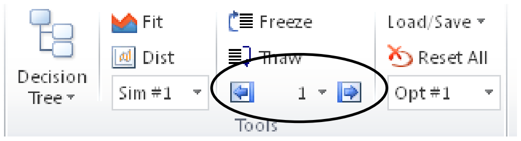
VBA / SDK: Engine.Params("StochR"), 0 <= value <= 1

This parameter sets the minimum amount of improvement which must be observed in order to update the incumbent.

Compute Recourse Statistics

VBA / SDK: Engine.Params("StochComp"), 0 <= value <= 1

If false, recourse variables for each trial will not be computed. Only "normal" decision variable values will be available after Solver has found a solution. If true, recourse variables will be computed for each trial which can be viewed by clicking through each trial on the Trial # Controls in the Tools section on the Analytic Solver Ribbon.



After Solver has found a solution, a histogram may be obtained over all trials by double clicking a cell containing a recourse variable or constraint.

Gurobi Solver Options

SDK: Engine name "Gurobi Solver", file name Gurobieng.dll

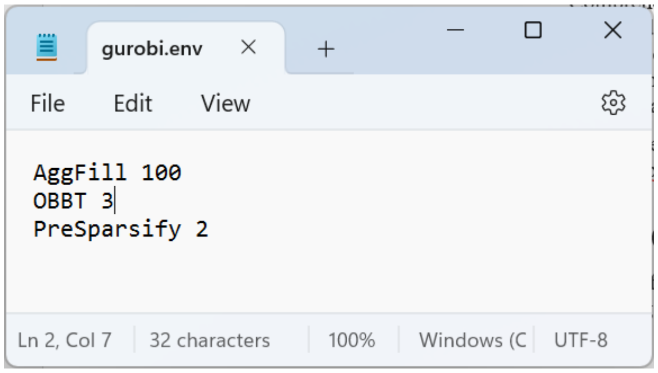
Gurobi is a powerful optimization solver widely used for solving linear programming (LP), mixed-integer programming (MIP), and quadratic programming (QP) problems. It offers a range of options that allow users to customize performance and behavior, including parameters for tuning the algorithm's speed, managing memory usage, and setting optimality tolerances. With features such as parallel processing and extensive logging, Gurobi enables users to efficiently tackle complex optimization challenges across various industries. The settings are categorized into several sections, including:

- General: These options control the overall behavior of the solver, such as the algorithm to use, whether the model is quadratic, scaling and time limits.
- Tolerances: Gurobi's tolerance options play a crucial role in determining the precision of optimization results. Key parameters include the optimality tolerance, which sets the allowable gap between the best-known objective value and the optimal solution, and feasibility tolerances, which dictate how close a solution must be to satisfy constraints. Adjusting these tolerances can significantly impact solver performance and solution quality, allowing users to balance between computational speed and solution accuracy based on their specific problem requirements. By fine-tuning these options, users can enhance the solver's effectiveness in various applications.
- LP: The Gurobi Engine offers a range of options specifically designed for linear programming (LP) problems, allowing users to tailor the solver's performance to their needs.
- Barrier: In this section, users can determine whether the Newton Barrier method will Cross Over to the Simplex method near the solution, so that sensitivity analysis information can be obtained.
- MIP: Here, users can set absolute and relative Integer Tolerances and related parameters which determine how close the Solver must come to an integer optimal solution before stopping.
- Stochastic Decomposition: This section contains options specific to the Stochastic Decomposition method bundled within Analytic Solver Comprehensive and Analytic Solver Optimization (Analytic Solver Desktop only) and utilized in the Gurobi Solver Engine. Stochastic Decomposition can be used to solve linear models with recourse variables and uncertainty in the constraints only. See the earlier Stochastic Decomposition section that appears earlier in this guide for more information.

Each option is listed using the name from the Desktop add-in and the Cloud app in the form: Desktop_Name/Cloud_Name.

Additional Options Available in VBA / SDK

If you want more fine-grained control over the behavior of the Gurobi Solver on linear mixed-integer problems and you are using Analytic Solver Desktop or Solver SDK, you can access additional options that aren’t available in the Task Pane Engine tab or documented here. To configure options not included in the Solver Task Pane, create a new text file named “gurobi.env” and list the option names and their settings, one per line. Save this workbook to the current directory (the same directory where the workbook is located). The Gurobi engine will apply the settings from this file and the Solver taskpane, when solving the model. Note that option settings in the Solver task pane will override any settings specified in the gurobi.env file.



```
AggFill 100
OBBT 3
PreSparsify 2
```

Contact Frontline Systems at 775-831-0300 or info@solver.com for more information. To use these options, you should have a good deal of experience solving linear mixed-integer problems. Access the Gurobi Optimizer Reference Manual here. This reference manual lists all available Gurobi Solver engine options.

Gurobi General Solver Options

Note that this Solver Engine supports the following Common Solver Options discussed earlier: Max Time, Iterations, Show Iterations, Assume Non-Negative, Bypass Solver Reports, Maximum Subproblems, Maximum Feasible Solutions and Random Seed. See the previous section, Common Solver Options, for more information.

Assume Quadratic Objective/AssumeQP

VBA / SDK: Engine.Params("AssumeQP"), value True/False

This option is used only if you are solving a quadratic programming problem, and you’ve either set the Task Pane Platform tab Optimization Model Interpreter to Excel Interpreter, or set the Advanced group Supply Engine With option to Gradients. When you use the PSI Interpreter and supply the Gurobi Solver with Structure or Convexity information, this option is ignored, and the type of objective – linear or quadratic – will be determined automatically.

Otherwise, if the Excel Interpreter is used you must use the QUADPRODUCT function and check this option. Failure to do so will result in the error message “Linearity conditions are not satisfied.”

Max Subproblems/ MaxSubProblems

VBA / SDK: Parameter Name "MaxSubProblems", integer value > 0

The value for the Max Subproblems option 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; 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 Subproblems/ MaxSubProblems

VBA / SDK: Parameter Name "MaxSubProblems", integer value > 0

The value for the Max Subproblems option 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; 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.

Number of Threads

VBA / SDK: Parameter Name "numberThreads", Default value = number of available system processors; 0 – Turns parallel processing off

The number of parallel threads of execution to be used when solving mixed-integer problems. Enter an integer from 1 to the Number of available system processors. The default is the number of system processors. A value of 0 turns parallel processing off.

Scaling

VBA / SDK: Engine.Params("Scaling"), -1 – Automatic, 0 – Disabled, 1-Normal, 2 – Geometric and 3 – Aggressive

These options determine how the Xpress^{MP} Optimizer will re-scale your model internally. Bear in mind that it is always a good idea to design your model so that all quantities are within a few orders of magnitude of each other, to minimize possible scaling problems.

By default, the rows and columns of the model are scaled to improve the numerical properties of the constraint matrix. Scaling is removed before returning the final solution. Scaling typically helps reduce solution times but may result in larger constraint violations in the original, unscaled model. Disabling scaling can sometimes reduce constraint violations. Experimenting with different scaling options can improve performance for numerically challenging models. For example, Geometric mean scaling is particularly effective for models with a wide range of coefficients across the constraint matrix. Normal and Aggressive scaling are less directly tied to specific model characteristics, so some experimentation may be required to determine their impact on performance.

Automatic – Xpress will automatically select the best scaling algorithm to apply to the model.

Disabled – Select this option to prohibit scaling of any type.

Normal – This option will only be selected if it appears to be clearly superior.

Geometric Mean – This option setting is well suited for models with a wide range of coefficients in the constraint matrix rows or columns.

Aggressive – This option will only be selected if it appears to be clearly superior.

Gurobi Tolerances, LP & Barrier Solver Options

Feasibility Tolerance/FeasibilityTol

VBA / SDK: Engine.Params("FeasibilityTol"), double value between 1E-9 and 1E-2; default value 1E-6.

This is the Gurobi Solver's primal feasibility tolerance. All constraints must be satisfied to this tolerance.

Optimality Tolerance/OptimalityTol

VBA / SDK: Engine.Params("OptimalityTol"), double value between 1E-9 and 1E-2; default value 1E-6.

This is the Gurobi Solver's optimality tolerance, also known as the dual feasibility tolerance. Reduced costs must all be smaller than this tolerance in the improving direction in order for a model to be declared optimal.

Integer Feasibility Tolerance/IntegralityTol

VBA / SDK: Engine.Params("IntegralityTol"), double value between 1E-9 and 1E-1; default value 1E-5.

This is the Gurobi Solver's integer feasibility tolerance. An integrality restriction on a decision variable is considered satisfied when the variable's value is less than this tolerance from the nearest integer value.

PSD Tolerance/PSDTol

VBA/*SDK*: Engine.Params("PSDTol"), value > 0, default = 1e-6

This parameter affects QP and MIQP models. The Gurobi engine can only solve convex QP models. If a QP model is nonconvex, Gurobi tries to perturb the diagonal values of the Q matrix so as to make the problem convex. This parameter controls the maximum diagonal perturbation allowed on the Q matrix to make it positive semi-definite. If a larger perturbation is required, the model is considered nonconvex and cannot be solved using the Gurobi engine.

QCP Tolerance/QCPTol

VBA/*SDK*: Engine.Params("QCPTol"), 0 <= value <= 1, default = 1e-6

This parameter affects QCP (quadratically constrained) models which are solved with the barrier method. This solver stops when the relative difference between the primal and dual objective values is less than the tolerance setting. Passing a smaller value for this option may result in a more accurate solution, if one exists, but could also result in the solver failing to converge.

LP Method

VBA / SDK: Engine.Params("Method"), -1 – Automatic, 0 - Primal Simplex, 1 - Dual Simplex, 2 - Barrier Method, 3 – Concurrent, 4 – Deterministic Concurrent

This option selects the algorithm used to solve the root node of a mixed-integer programming problem. Primal Simplex, Dual Simplex, and Barrier Method can be used for continuous QP models. Primal Simplex or Dual Simplex should be used when solving the root of an MIQP model. The Barrier Method should be used when solving continuous QCP models. If memory is an issue when solving an LP, then it is recommended to use the Dual Simplex method.

LP Presolve

VBA / SDK: Engine.Params("Presolve"), -1- Automatic, 0 - Off, 1 - Conservative, 2 - Aggressive

This option controls the Presolve process in the Gurobi Solver. More aggressive application of presolve takes more time, but can sometimes lead to a significantly tighter formulation of the model, saving time in the overall solution process.

LP Pricing

VBA / SDK: Engine.Params("Pricing"), -1 - Automatic, 0 - Partial Pricing, 1 - Steepest Edge, 2 - Devex, or 3 - Quick-Start Steepest Edge

This option determines the variable ‘pricing’ strategy used in the Simplex method. The default setting (Automatic) is usually the best choice; you’ll need deep knowledge of how the Simplex method performs to select a better setting.

Barrier Iteration Limit/BarIterLimit

VBA / SDK: Engine.Params("BarIterLimit"), integer value > 0

This option is effective only when the LP Method option is set to 2 (Barrier). It limits the number of Barrier iterations that will be performed before the Solver stops. It is a separate limit because the number of Barrier iterations is typically much smaller than the number of Simplex iterations (but each iteration is computationally much more expensive). The Barrier method frequently finishes within 50 iterations, and rarely goes beyond 100 iterations.

Barrier Convergence Tolerance/BarConvTol

VBA / SDK: Engine.Params("BarConvTol"), double value between 1E-10 and 1E-0; default value 1E-8.

This option is effective only when the LP Method option is set to 2 (Barrier). It sets the Barrier convergence tolerance. The Barrier Solver terminates when the relative difference between the primal and dual objective values is less than this tolerance.

Crossover Strategy/CrossOver

VBA / SDK: Engine.Params("CrossOver"), -1 - Automatic, 0 - No crossover, 1 - Primal Simplex (dual variables first), 2 – Dual Simplex (dual variables first), 3 - Primal Simplex (primal variables first), 4 - Dual Simplex (primal variables first)

This option determines the crossover strategy used to transform the Barrier solution into a basic solution (as produced by the Simplex method). Use value 0 to disable crossover – the Solver will return an interior solution. Options 1 and 2 push dual variables first, then primal variables. Option 1 finishes with primal, while option 2 finishes with dual. Options 3 and 4 push primal variables first, then dual variables. Option 3 finishes with primal, while option 4 finishes with dual. The default value of -1 chooses automatically.

Ordering

VBA / SDK: Engine.Params("Ordering"), -1 - Automatic, 0 - Approximate Minimum Degree, 1 - Nested Dissection.

This option selects the Barrier sparse matrix fill-reducing algorithm. A value of 0 chooses Approximate Minimum Degree ordering; a value of 1 chooses Nested Dissection ordering. The default value of -1 chooses automatically.

Gurobi MIP Solver Options

Note that this Solver Engine supports the Common Solver Options, Integer Tolerance and Integer Cutoff. See the Common Solver section that appeard previously, for more information.

Node File Start/NodeFileStart

VBA / SDK: Engine.Params("NodeFileStart"), double value > 0

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. The Node File Start option determines how much memory can be used for the Branch & Bound tree before the Gurobi Solver starts writing the tree node information to a disk file. It is an integer or fractional value *measured in gigabytes* (GB). The default value is effectively infinity; if you find that the Gurobi Solver returns the Solver Result message ‘Out of memory’, you should set this option to a value less than the amount of main (RAM) memory you have available for the Solver to use (e.g. 1 or 2).

Heuristics

VBA / SDK: Engine.Params("Heuristics"), 0 < value < 1, default 0.05

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. The Heuristics option controls the amount of time the Solver will spend on mixed-integer heuristics. Larger values produce more and better feasible solutions, at a cost of slower progress in the optimal value of the ‘incumbent’ or best solution so far.

Max SubMip Nodes/SubMips

VBA / SDK: Engine.Params("SubMips"), integer value >= 0

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. The Max SubMipt Nodes option limits the number of nodes explored by the RINS (Relaxation Induced Neighborhood Search) heuristic. Exploring more nodes can produce better solutions, but it generally takes longer. You can increase this if you are having trouble finding good feasible solutions.

Variable Branching/VarBranching

VBA / SDK: Engine.Params("VarBranching"), -1 - Automatic, 0 - Pseudo Reduced Cost Branching, 1 - Pseudo Shadow Price Branching, 2 - Maximum Infeasibility Branching, 3 - Strong Branching.

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. The Varaible Branching option controls the branch variable selection strategy. Variable selection can have a significant impact on overall time to solution, but the default (automatically chosen) strategy is usually the best choice.

Cut Generation/CutGeneration

VBA / SDK: Engine.Params("CutGeneration"), -1 – Automatic, 0 - No cut generation, 1 = Conservative cut generation, 2 - Aggressive cut generation, 3 – Very aggressive cut generation.

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. The Cut Generation option determines the overall strategy for cut generation. 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.

Symmetry

VBA / SDK: Engine.Params("Symmetry"), -1- Automatic, 0 - Off, 1 - Conservative, 2 - Aggressive

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. The Symmetry option controls mixed-integer symmetry detection. The default setting (Automatic) works well for most problems.

MIP Focus/MIPFocus

VBA / SDK: Engine.Params("MIPFocus"), 0 - Balanced approach, 1 - Focus on finding feasible solutions, 2 - Focus on proving optimality, 3 - Focus on moving the best objective bound.

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. The MIP Focus option controls the focus of the mixed-integer solver. The default value tries to strike a balance between finding good feasible solutions and proving optimality. If you are more interested in good quality feasible solutions, use a value of 1. If you believe the Solver is having no trouble finding the optimal solution, and wish to focus more attention on proving optimality, use a value of 2. If the best objective bound is moving very slowly (or not at all), you may want to try a value of 3 to focus on the bound.

Solution Improvement

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. Gurobi has the ability to change parameter settings in the middle of a solve in order to adopt a new strategy that attempts to find a better feasible solution rather than trying to move the Objective of best bound.

The Gurobi engine 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, Gurobi finds "candidate" integer solutions, and keeps the best solution so far as the "incumbent." By eliminating alternatives as its proceeds, Gurobi also tightens the "best bound" on how good the integer solution can be.

Each time Gurobi finds a new incumbent – an improved all-integer solution – it computes the maximum percentage difference (or optimality gap) between the objective of this solution and the current best bound on the objective:

(Objective of incumbent - Objective of best bound) / Objective of best bound

Solution Improvement/ImproveStartNodes

VBA / SDK : Engine.Params("ImproveStartNodes") 0 < value

This parameter allows you to specify the node count that, when reached, will trigger Gurobi to switch to a solution improvement strategy. For example, setting this parameter to 10 will cause Gurobi to switch strategies once the node count is larger than 10.

Cut Passes/CutPasses

VBA / SDK: Engine.Params("CutPasses") -1 < value

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. Enter a non-negative value here to specify the maximum number of cutting plane passes to be performed during the root cut generation. The default value of -1 will select the number of cut passes automatically.

Pre Passes/PrePasses

VBA / SDK: Engine.Params("PrePasses") -1 < value

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. Enter a non-negative value here to specify the maximum number of passes performed by Presolve. The default value of -1 will select the number of cut passes automatically.

Improve Start Gap/ImproveStartGap

VBA / SDK: Engine.Params("ImproveStartGap") 0 < value

This parameter allows you to specify an optimality gap that, when reached, will trigger Gurobi to switch to a solution improvement strategy. For example, setting this parameter to 0.1 will cause Gurobi to switch strategies once the relative optimality gap is smaller than 0.1.

Zero-objective heuristic

Number of nodes to explore in the zero-objective heuristic. Note that this heuristic is only applied at the end of the MIP root, and only when no other root heuristic finds feasible solution. Maximum value is 2,000,000,000.

Knitro Solver Options

SDK: Engine name "Knitro Solver", file name Knitroeng.dll

This Solver Engine supports the Common Solver Options discussed earlier, plus the Global Optimization options, and the Convergence, Population Size, Random Seed, Treat Constraints as Linear, Treat Objective as Linear, and Solution method options, which are specific to the Knitro Solver. The Global Optimization, Population Size and Random Seed options are discussed below in the section “Options for Global Optimization.” The other options specific to the Knitro Solver are discussed here.

Convergence

VBA / SDK: Engine.Params("Convergence"), 0 <= value <= 1

This option is included for compatibility with the GRG, Large-Scale GRG, and Large-Scale SQP Solvers, but its value is not currently used by the Knitro Solver. As discussed in the chapter “Solver Result Messages,” each of the nonlinear Solvers (including the Knitro Solver) will stop and display the message “Solver has converged to the current solution” when the solution is changing very slowly for the last few iterations or trial solutions. The GRG, Large-Scale GRG, and Large-Scale SQP Solvers use the Convergence value in testing for this condition. The Knitro Solver uses slightly different criteria to determine when to stop due to slow progress, but the effect is much the same.

If you are getting this message when you’re seeking a locally optimal solution, you should 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.

Treat Constraints as Linear/LinearConstraints

VBA / SDK: Engine.Params("LinearConstraints"), value 1-Treat as Linear or 0-Don’t Treat as Linear

The Treat Constraints as Linear and Treat Objective as Linear options are used only if (i) you have a problem with all linear constraints and possibly a linear objective but (ii) you *aren’t* using the PSI Interpreter in Analytic Solver Desktop, or you *aren’t* setting the Model object AllGradDepend property, or calling the SolverModAll-GradDependSet procedural API function, in Solver SDK. (Analytic Solver Cloud always using the Psi Interpreter.) These options tell the Knitro Solver to treat the constraints and/or the objective as linear, and exploit this treatment to solve the problem more efficiently. If you are solving a problem with all linear constraints, such as a linear programming (LP) or quadratic programming (QP) problem, you can use this option to speed up the solution process.

In Analytic Solver Desktop, use of the PSI Interpreter is the *default* setting, the Knitro Solver obtains information about linearity of the constraints and objective from the Polymorphic Spreadsheet Interpreter, and these options are ignored.

When you solve a problem Interpreter = Excel Interpreter in Analytic Solver Desktop, or without setting the Model object AllGradDepend property in Solver SDK, the Knitro Solver normally treats the objective and all constraints as smooth nonlinear functions – even if they are actually linear. Hence, the Solver will compute gradients of these functions at each major iteration. If this option is set to 1, the Knitro Solver will *assume* that the constraints are linear (hence their gradients are constant) and will compute their gradients only once, at the beginning of the solution process. This option is especially effective when used in combination with the "Treat Objective as Linear" option, for a linear programming problem, as explained below.

Treat Objective as Linear/LinearObjective

VBA / SDK: Engine.Params("LinearObjective"), value 1-Treat as Linear or 0-Don't Treat as Linear

If you are solving a linear programming problem, you can use this option to greatly speed up the solution process. As described under "Treat Constraints as Linear" above, when you solve a problem with Interpreter = Excel Interpreter in Analytic Solver Desktop, or without setting the Model object AllGradDepend property in Solver SDK, the Knitro Solver normally treats the objective and all constraints as smooth nonlinear functions, and computes gradients of these functions via finite differencing at each major iteration. When both options "Treat Constraints as Linear" and "Treat Objective as Linear" are True, the Knitro Solver computes the function gradients only once, at the beginning of the solution process – taking about the same amount of time as "Problem Setup" in a large-scale LP Solver. The Knitro Solver then solves the problem using active-set methods that are highly effective, and competitive with the Simplex method for linear programming.

If you set only "Treat Constraints as Linear," but not "Treat Objective as Linear" to True, the improvement in solution time depends on whether or not you are using *automatic differentiation* in Analytic Solver Desktop, or you are supplying an Evaluator for Eval_Type_Gradient in Solver SDK. If automatic differentiation or an Evaluator for derivatives are being used, you should see a significant speedup, since only derivatives for the *objective* need be computed on each major iteration. But if finite differencing is being used (Interpreter = Excel Interpreter, or you've defined only an Evaluator for Eval_Type_Function), you may not see much speed improvement, especially in Excel. This is because it takes almost as much time to compute gradients of *some* of the functions (or even just one, the objective) as it does to compute gradients of *all* of the functions, by recalculating the Excel spreadsheet or by calling your function Evaluator with perturbed values for each of the decision variables.

Relax Bounds on Variables

VBA / SDK: Engine.Params("RelaxBounds"), value 1-Relax or 0-Don't Relax

By default, the Knitro Solver seeks to ensure that any trial points evaluated during the solution process will not have values that violate the bounds on the variables you specify. If your problem functions cannot be evaluated for values outside the variable bounds, this default behavior will ensure that the solution process can continue. However, at times the Knitro Solver can make more rapid progress along a given search direction by testing trial points with values *outside* the bounds on the variables. If you want to permit this to happen, set this option to 1. If you receive the Solver Result Message "Solver encountered an error value in a target or constraint cell," as a first step you should ensure that this option is set to 0.

Solution Method

VBA / SDK: Engine.Params("SolutionMethod"), value 1-Select Automatically, 2-Interior Point Direct, 3-Interior Point CG, or 4-Active Set

This option selects the overall optimization algorithm, or solution method, used by the Knitro Solver. The Knitro Solver now includes *both* interior point and active set (SLQP) methods, which enables this Solver to outperform most other Solvers on all kinds of nonlinear problems.

Select Automatically

By default, the Knitro Solver will select the solution method automatically. If the problem is linear, or if a mixed-integer (Branch & Bound) subproblem is being solved, Knitro will use its Active Set method. If the problem is nonlinear, Knitro may choose any of the solution methods, but most often the Interior Point Direct method will be used.

Interior Point Direct

This method computes a step by directly factoring the matrix of normal equations. (The Interior Point CG method uses an iterative approach to solving this system.) The Direct method can fall back automatically to the CG method if the computed step appears to be of low quality. This option may perform substantially better than the CG method if the problem is ill-conditioned.

Interior Point CG

This method uses an iterative Conjugate Gradient approach to computing the step at each iteration. It typically offers the best performance if the Hessians of the problem functions are large and dense. For challenging nonlinear problems, it is a good idea to try both the Direct and the CG option, since it is difficult to predict in advance which method will perform best on a given problem.

Active Set

This method uses a new active-set Sequential Linear-Quadratic Programming (SLQP) optimization algorithm. It usually offers the best performance on linear problems, highly constrained problems, or problems where a good starting point is known (and supplied on the Excel spreadsheet).

Derivatives Options

Derivatives

VBA / SDK: Engine.Params("Derivatives"), value 1-Forward or 2-Central

The Derivatives option is relevant only if, in Analytic Solver Comprehensive and Analytic Solver Optimization you've set Interpreter = Excel Interpreter in the Task Pane Platform tab, or you *haven't* supplied an Evaluator for Eval_Type_Gradient in Solver SDK – which means that *first* derivatives of the problem functions must be approximated via finite differences. It determines whether "forward differencing" or "central differencing" is performed.

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.

Second Derivatives

VBA / SDK: Engine.Params("SecondDerivatives"), value 1-Analytic 2nd Derivatives, 2- Analytic 1st Derivatives, or 3-Finite Differences

On each major iteration, the Knitro Solver computes an approximation of the Hessian of the Lagrangian function for the problem. The Lagrangian is a function of the objective and all of the constraints, so computing its Hessian requires that the Hessian of the objective and the Hessian of each constraint be obtained. This option determines how the Knitro Solver will obtain or approximate the Hessian values.

Analytic 2nd Derivatives indicates that Hessian values will be obtained via *second order automatic differentiation* from the Interpreter in Analytic Solver, or by calling your Evaluator for Eval_Type_Hessian in the Solver SDK. This is the most accurate method, and it is often the fastest method as well, but it may require a *great deal of memory* and computation.

Analytic 1st Derivatives indicates that the Knitro Solver should obtain gradients via *first order automatic differentiation* from the Interpreter in Analytic Solver, or by calling your Evaluator for Eval_Type_Gradient in Solver SDK, and then construct an approximation of the Hessian of each function via a quasi-Newton method (BFGS or limited-memory BFGS).

Finite Differences indicates that the Knitro Solver should obtain gradients via *first order automatic differentiation* from the Interpreter in Analytic Solver or by calling your Evaluator for Eval_Type_Gradient in Solver SDK, and then estimate the Hessian values by taking finite differences of the first derivatives.

If you've set Interpreter = Excel Interpreter in Analytic Solver Desktop (only), or if you *haven't* supplied an Evaluator for Eval_Type_Gradient in Solver SDK, then "analytic" gradients will not be available when the Knitro Solver runs. (You should do this only if you have no alternative, perhaps because your model depends on functions for which the PSI Interpreter cannot compute gradients, or functions for which it's too difficult to write your own Evaluator for function gradients.) In this case, the Knitro Solver will compute approximate *first* derivatives via finite differencing – either Forward or Central as set by the Derivatives option described above – and then use these values to approximate the Hessian or second derivative values, using either the "Analytic 1st Derivatives" or "Finite Differences" method described here. Since there are two levels of approximation involved in either case, the solution found may be less accurate, and the Knitro Solver may require more iterations to reach the solution.

Options for Global Optimization

As described in the chapter "Using the Solver Engines," you can solve global optimization problems with the OptQuest Solver, or you can use the Large-Scale GRG Solver, the Large-Scale SQP Solver, or the Knitro Solver augmented with so-called "multistart" methods for such problems. This section describes options that control the multistart methods for global optimization, which will automatically run the Knitro Solver from a number of starting points in order to seek the globally optimal solution.

As seen in earlier sections of this chapter, the Global Optimization options group and the Population Size and Random Seed options are supported by the Large-Scale GRG Solver, the Large-Scale SQP Solver, and the Knitro Solver, and they have the same effect in each of these Solver engines.

Multistart Search

VBA / SDK: Engine.Params("MultiStart"), value 1-Use Multistart Search or 0-Don't Use Multistart Search

Set this option to 1/True to activate the multistart methods for global optimization. If this option is set to 0/False, the remaining options in this group (Population Size, Number of Threads, Maximum Start/Bound Range, Stopping Rule and Random Seed) are ignored.

The multistart methods will generate candidate starting points for the Knitro Solver with randomly selected values between the bounds you specify for the variables, then attempt to find a local optimum from each starting point, using the original problem. After all candidate starting points are tried, Knitro will return the local optimum with the best objective function value, assuming at least one local optimum has been found. If no local optimum has been found, Knitro will return the best feasible solution found. If no feasible solutions have been found, Knitro will return an infeasible result.

While obtaining the global optimum to a nonlinear optimization problem is typically the desired outcome, Knitro is unable to guarantee that multistart will be able to find the global optimum. In general, the global optimum can only be found with special knowledge of the problem, i.e. either the objective or constraints or both may be required to be bounded by another piece-wise convext function(s). Knitro solves with very little information pertaining to the functional form of the problem. The probability of finding a good local solution increases as the number of starting points increases.

The latest Knitro engine offers an improved multi-start procedure. This new multi-start, by default, automatically terminates when the probability of finding new local solutions is small, while always producing deterministic results. Set the Stopping Rule option, "Stop when probability of finding a new solution is low", to control how quickly termination is triggered.

Number of Threads

VBA / SDK: Model.Params("NumThreads"), integer value > 0, default value = 0

If the Number of Threads parameter on the *Platform* tab of the Solver Task Pane is greater than 0, then KNITRO will use that value to set the number of threads for the KNITRO engine.

If the Number of Threads parameter on the *Platform* tab of the Solver Task Pane is equal to 0, then the KNITRO engine will use the value for the Number of Threads parameter on the KNITRO engine tab.

If Number of Threads on both the Platform tab and the Knitro Engine tab are equal to 0, then the KNITRO engine will set number of threads equal to the number of processors.

The KNITRO engine uses the Number of Threads parameter to determine the number of threads used in Multistart, when the branch and bound method is used to solve a mixed-integer problem and during the "normal" solving process, for example when computing multiple derivatives. If Number of Threads = N, KNITRO will not exceed N threads in total.

Population Size

VBA / SDK: Engine.Params("PopulationSize"), integer value > 0, default value = 0

This option has an effect only if the Multistart option is set to 1/True. It determines the size of the “population” of candidate starting points for multistart search.

If 0 is entered for Population Size, Knitro will automatically choose a value based on the problem size using the calculation which will be the minimum of 200 or 10 * the number of variables in the problem.

If an integer n greater than 0 is entered, Multistart will try n number of start points.

Maximum Start Range

VBA / SDK: Engine.Params("MaxStartRange"), integer value > 0; default value = 1.0e+20

This option specifies the maximum range that each variable may take on when determining candidate starting points.

If a variable has an upper and lower bounds and the difference between them (upper bound – lower bound) is less than or equal to the setting for this option, then new starting point values for the variable can be any number between its upper and lower bounds.

If the variable is unbounded one one side or both, or the difference between the bounds is greater than the setting for this option, the new starting point values are restricted by this option. If x_i is such a variable, then all initial values satisfy:

$$Max\{b_i^L, x_i^0 - \tau\} \leq x_i \leq \min\{b_i^U, x_i^0 + \tau\} \; where$$

$$\tau = \min\{Maximum\ Start\ Range/2, Maximum\ Bound\ Range/2$$

Where $x_i^0$is the initial value of x_i provided by the user, and b_i^L and $b_i^U$are the variable bounds (possibly infinite) on x_i. This option has no effect unless Multistart is set to True.

Maximum Bound Range

VBA / SDK: Engine.Params("MaxBoundRange"), integer value > 0; default value = 1,000.0

This option specifies the maximum range that an unbounded variable can take on when determing candidate starting points.

If a variable is unbounded in one or both directions, then new starting point values will be restricted by this option. If x_i is such a variable, then all initial values satisfy:

$$\begin{aligned} Max\{b_i^L, x_i^0 - Maximum\ Bound\ Range/2\} &\leq x_i \\ &\leq \min\{b_i^U, x_i^0 + Maximum\ Bound\ Range\} \end{aligned}$$

where $x_i^0$is the initial value of x_i provided by the user, and b_i^L and $b_i^U$are the variable bounds (possibly infinite) on x_i. This option has no effect unless Multistart is set to True.

A Note on Maximum Start and Bound Range

The Maximum Bound Range option applies to unbounded variables in at least one direction, for example the variable upper and/or lower bounds are infinite, and keeps new starting points within a total range equal to the value Maximum Bound Range.

The Maximum Start Range option applies to all variables and keeps new starting points within a total range equal to the value for the Maximum Start Range option, This option overrules the Maximum Bound Range option if the Maximum Start Range creates a tighter bound.

In general, use Maximum Start Range to limit the multstart search only if the initial starting point is known to be the center of the desired search area. Use Maximum Bound Range as an alternate bound to limit multistart when a variable is unbounded.

Stopping Rule

VBA / SDK: Engine.Params("StoppingRule"), integer value = 0, 1, 2, 3 , default value = 0

This option specifies the condition for terminating multistart. This option has no effect unless the Multistart option is set to true.

The options are:

0. Stop after the trial points in the Population Size have been run. – Use this option to stop Solver only after all trial points have been solved.
1. Stop after the first local optimal solution is found. – Use this option to stop Solver after the first locally optimal solution has been found.
2. Stop after the first feasible solution is found. **-** Use this option to stop Solver after the first feasible solution has been found.
3. Stop after the first solution estimate is found. **-** Use this option to stop Solver after the first solution estimate has been found.
4. Stop when the probability of finding a new solution is low. – Use this option to control how quickly termination is triggered.

Random Seed

VBA / SDK: Engine.Params("RandomSeed"), integer value > 0

This option has an effect only if the Multistart option is set to 1/True. It sets a seed for the random number generator used by the multistart methods.

The multistart methods use a process of random sampling to generate candidate starting points for the Knitro 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 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 Knitro Solver options on each search for a locally optimal solution. To do this, enter an integer value for this option; this value will then be used to “seed” the random number generator each time you solve.

Integer Options

See above for descriptions on Max Subproblems, Max Feasible Solutions and Integer Cutoff.

Branching Variable Selection Rule

VBA / SDK: Engine.Params(""), integer value > 0

This option specifies which braning rule to use for the branch and bound method.

Automatic (Default)

Most Fractional Branching

Pseudo Cost Branching

Strong Branching

Integer Feasibility Tolerance

VBA / SDK: Engine.Params("IntegralityTol"), double value between 1E-10 and 1E-1; default value 1E-8.

The value of this option is the tolerance within which a decision variable's value (at the optimal solution to a subproblem) is considered to be integer, for purposes of the Branch & Bound method. If the absolute value of the difference between the variable's value and the nearest integer is less than this tolerance, the variable is treated as having that exact integer value. The default setting is 1.0e-8.

LP Algorithm

Specifies which algorithm to use for any linear programming subproblems that may occur in the MIP branch and bound procedure.

Automatic (Default): Let Knitro automatically select the best LP algorithm, based on the characteristics of the problem being solved.

Interior Direct: Use the Interior/Direct algorithm.

Interior CG: Use the Interior CG algorithm.

Active Set: Use the Active Set Algorithm.

Node Algorithm

Specifies which algorithm to use for any linear programming subproblems that may occur in the MIP branch and bound procedure.

Automatic (Default): Let Knitro automatically select the best algorithm, based on the characteristics of the model being solved.

Interior Direct: Use the Interior/Direct algorithm.

Interior CG: Use the Interior CG algorithm.

Active Set: Use the Active Set Algorithm.

SQP: Use the SQP algorithm.

All: Run all algorithms in parallel.

Node Selection Rule

Specifies the MIP select rule for choosing the next node in the branch and bound tree.

Automatic (Default): Let Knitro automatically choose the best node selection rule for the model being solved.

Depth First: Search the tree using a depth first procedure.

Best Bound: Select the node with the best relaxation bound.

Combination: Use depth first unless pruned, then use best bound.

Absolute Integer Tolerance

VBA / SDK: Engine.Params("AbsIntTol"), any numeric value

The value of this option is the absolute tolerance used to determine whether the Branch & Bound method should continue or stop.

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 objective value of an integer solution can be. If the absolute difference between the incumbent’s objective value and the best bound is less than this tolerance, the Solver will stop with the message “Solver found an integer solution within tolerance” (result code 14).

Relative Integer Tolerance

VBA / SDK: Engine.Params("RelIntTol"), 0 < value < 1

The value of this option is the relative tolerance used to determine whether the Branch & Bound method should continue or stop.

As described above for the Absolute Integer Tolerance, the Branch & Bound method keeps track of the objective value of the “incumbent” and the “best bound” on the objective found so far. If the absolute difference between the incumbent’s objective value and the best bound, *divided by the best bound,* is less than the Relative Integer Tolerance, the Solver will stop with the message “Solver found an integer solution within tolerance” (result code 14).

MIP Method

Knitro offers two algorithms for solving mixed-integer optimization models (MINLP): NLBB and MISQP. Both methods are designed for convex mixed-integer problems and serve only as heuristics for non-convex problems.

- NLPBB (nonlinear branch-and-bound algorithm
- MISQP (mixed-integer sequential quadratic programming algorithm

The NLPBB algorithm is a standard branch-and-bound algorithm for nonlinear optimization. This method involves solving a relaxed, continuous nonlinear optimization subproblem at every node of the branch-and-bound tree. This method is designed to be applied to solve convex models but can also be applied to non-convex models where it can typically find a local solution. Note: If solving a non-convex model, the optimality gap measure may not be accurate since this measure assumes that the nonlinear optimization subproblems are solved to global optimality.

The MISQP algorithm is a heuristic extension of the SQP method, tailored for continuous, nonlinear optimization problems that include integer variables. It is particularly suited for small-scale problems, typically fewer than 100 variables, where function evaluations may involve compute intensive black-box calculations where derivatives may not be available. This method handles models where the integer variables cannot be relaxed, meaning that function evaluations can only occur when integer variables are at integer values. This method typically converges in far fewer function evaluations compared with other MINLP methods in Knitro and can be selected when solving either convex or non-convex models. When applied to non-convex models, this method may converge to a local feasible point.

Automatic: Allow Knitro to select the best MIP method automatically. (Default)

Standard Branch and Bound: Use the standard branch and bound method.

Hybrid Quesada: Grossman – Use the hybrid Quesada-Grossman method. This option is only relevant to convex, nonlinear problems only.

Mixed – Integer SQP: (MISQP) Use the mixed -integer SQP method. This method allows for non-relaxable integer variables.

Large-Scale GRG Solver Options

SDK: Engine name "Large-Scale GRG Solver", file name LSGRGeng.dll

This Solver Engine supports the Common Solver Options discussed earlier, plus the Global Optimization options, and the Convergence, Population Size, Random Seed, Recognize Linear Variables, Relax Bounds on Variables options, Estimates, Derivatives and Search options, which are specific to the Large-Scale GRG Solver.

The Global Optimization options and the Population Size and Random Seed options are discussed below in the section "Options for Global Optimization." The other options specific to the Large-Scale GRG (LSGRG) Solver are discussed here.

Convergence

VBA / SDK: Engine.Params("Convergence"), 0 <= value <= 1

As discussed in the chapter "Solver Result Messages," the LSGRG 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 LSGRG Solver stops if the absolute value of the *relative* change in the objective function is less than the value of the Convergence option for the last few iterations. While the default value of 1.0E-4 (0.0001) is suitable for most problems, it may be too large for some models, causing the LSGRG Solver to stop prematurely when this test is satisfied, instead of continuing for more iterations 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 Convergence option to a smaller value such as 1.0E-5 or 1.0E-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: Engine.Params("RecognizeLinear"), value 1-Recognize or 0-Don't Recognize

This option has only a limited effect in the LSGRG engine. It is intended to take advantage of the fact that in many nonlinear problems, some of the variables occur linearly in the objective and all of the constraints. Hence the partial derivatives of the problem functions with respect to these variables are constant, and need not be re-computed on each iteration. When *finite differencing* is used to estimate partial derivatives, this means that the steps involved in perturbing such a variable and computing values for all of the problem functions can be skipped.

In Analytic Solver Desktop when the PSI Interpreter is used (which is the default) and always in Analytic Solver Cloud, using this option will not save any time, because partial derivatives are computed via *automatic differentiation* rather than *finite differencing*. Similarly, in Solver SDK, when you supply an Evaluator to compute derivatives, this option won't save any time. Further, the PSI Interpreter can supply "dependents analysis" information, which the LSGRG Solver will use to recognize variables that occur linearly in only *some* (as well as all) of the problem functions.

Relax Bounds on Variables

VBA / SDK: Engine.Params("RelaxBounds"), value 1-Relax or 0-Don't Relax

By default (and *unlike* the nonlinear GRG Solver bundled within Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK), the LSGRG Solver ensures that any trial points evaluated during the solution process will not have values that violate the bounds on the variables you specify, even by a small amount. If your problem functions cannot be evaluated for values outside the variable bounds, this default behavior will ensure that the solution process can continue. However, at times the LSGRG Solver can make more rapid progress along a given search direction by testing trial points with values slightly outside the bounds on the variables. If you want to permit this to happen, set this option to 1. If you receive the Solver Result Message "Solver encountered an error value in a target or constraint cell," as a first step you should set this option to 0.

Other Nonlinear Options

The default values for the Estimates, Derivatives and Search options can be used for most problems. If you'd like to change these options to improve performance on your model, this section will provide some general background on how they are used by the LSGRG Solver.

On each major iteration, the LSGRG Solver requires values for the partial derivatives 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: Engine.Params("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:** This choice has no bearing on quadratic programming problems.

Derivatives

VBA / SDK: Engine.Params("Derivatives"), value 1-Forward or 2-Central

On each major iteration, the LSGRG Solver requires values for the partial derivatives of the objective and constraints (i.e. the Jacobian matrix). In Analytic Solver Desktop, these derivatives may be computed via *automatic differentiation* (when Interpreter = Psi Interpreter) or via *finite differencing* (when Interpreter = Excel Interpreter). In Solver SDK, if you supply an Evaluator for Eval_Type_Gradient, it is used to compute derivatives in a manner analogous to automatic differentiation; if you don't supply such an Evaluator, the LSGRG Solver computes derivatives via finite differencing – calling your function Evaluator (Eval_Type_Function) once or twice for each decision variable in order to do so. In Analytic Solver Cloud automatic differentiation is always used.

The Derivatives option in the LSGRG Solver Options dialog is relevant only if finite differencing is being used (which is never the case in Analytic Solver Cloud) to compute derivative values. It determines whether "forward differencing" or "central differencing" is performed.

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.

Search Option

VBA / SDK: Engine.Params("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. Instead, a direction is chosen through an estimation method. The default choice Newton uses a quasi-Newton (or BFGS) method, which maintains an *approximation* to the Hessian matrix; this requires more storage (an amount proportional to the square of the number of currently binding constraints) but performs very well in practice. The alternative choice Conjugate uses a conjugate gradient method, which does not require storage for the Hessian matrix and still performs well in most cases. The choice you make here is not crucial, since the LSGRG solver is capable of switching *automatically* between the quasi-Newton and conjugate gradient methods depending on the available storage.

Large-Scale LP/QP Solver Options

SDK: Engine name "Large-Scale LP Solver", file name LSLPeng.dll

This Solver Engine supports the Common Solver Options discussed earlier except the Precision option, plus the Primal Tolerance, Dual Tolerance, and Do Presolve options, which are specific to the Large-Scale LP/QP Solver. The Bypass Solver Reports option is worth noting here *if you are using Analytic Solver Desktop*, since it can have a large impact on solution time when the Large-Scale LP/QP Solver is used.

Note that the default values for Primal Tolerance and Dual Tolerance have been chosen very carefully, and the Large-Scale LP/QP Solver is designed to solve the vast majority of LP problems "out of the box" with these default tolerances. To change these tolerances, you probably should have some background in LP solution methods, or consult with someone who has such a background.

Primal Tolerance

VBA / SDK: Engine.Params("PrimalTolerance"), 0 < value < 1

The Primal Tolerance is the maximum amount by which the primal constraints can be violated and still be considered feasible. The default value of 1.0E-7 for this tolerance is suitable for most problems.

Dual Tolerance

VBA / SDK: Engine.Params("DualTolerance"), 0 < value < 1

The Dual Tolerance is the maximum amount by which the dual constraints can be violated and still be considered feasible. The default value of 1.0E-7 for this tolerance is suitable for most problems.

Presolve

VBA / SDK: Engine.Params("Presolve"), value 1-Do Presolve or 0-No Presolve

When this option is set to 1 (which is the default setting), the Large-Scale LP/QP 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

VBA / SDK: Engine.Params("Derivatives"), value 1-Forward or 2-Central

When a quadratic programming (QP) problem is solved with the Large-Scale LP/QP Solver, the quadratic Solver extension requires first and second partial derivatives of the objective function at various points. In Analytic Solver Cloud and when Interpreter is set to Automatic or Psi Interpreter in Analytic Solver Desktop, these derivatives may be computed via automatic differentiation or via finite differencing. For more information, see the Analytic Solver User Guide.

Derivatives In Analytic Solver Desktop

When Interpreter = PSI Interpreter in the Task Pane Platform tab of *Analytic Solver Desktop*, automatic differentiation is used, exact derivative values are computed, and the setting of the Derivatives choice is ignored. When Interpreter = 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 each iteration may take up to twice as long as with Forward differencing. Note that automatic differentiation is much faster than either Forward or Central differencing.

Use Classic Search

VBA / SDK: Parameter Name "ClassicSearch", value 1/True or 0/False

Classic Search uses the same search methods as previous releases for problems with integer variables and quadratically constrained problems (QCPs). In our universe of test models, the new (default) search methods yield faster solutions in about 90% of cases – but in the 10% of cases where the new methods yield *slower* solutions, switching to Classic Search will yield the same results you were getting before.

Large-Scale LP/QP Mixed-Integer Options

Preprocessing

VBA / SDK: Parameter Name "PreProcessing", 1 = Automatic, 2 = None, 3 = Aggressive

Use this option to determine the extent of Preprocessing and Probing strategies used by the LP/Quadratic Solver on LP/MIP (linear mixed-integer) problems. Select from **None**, **Aggressive**, and **Automatic** (the default).

These methods consider the possible settings of certain binary integer variables and their implications for fixing the values of other binary integer variables, tightening the bounds on continuous variables, and in some cases, determining that the subproblem is infeasible (so it is unnecessary to solve it at all).

They can also scan the model for constraints of the form x1 + x2 + … + xn = 1 where all of the variables xi 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 xi must be 1, and all the others must be 0; hence only n possible permutations of values for the variables (rather than 2n) need be considered.

Cuts

VBA / SDK: Parameter Name "Cuts", 1 = Automatic, 2 = None, 3 = Aggressive

Use this option to determine the extent of Cut Generation strategies used by the LP/Quadratic Solver on LP/MIP (linear mixed-integer) problems. Select from **None**, **Aggressive**, and **Automatic** (the default).

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 used by the LP/Quadratic Solver include:

- **Knapsack 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**: These cuts are found by examining the basis inverse at the optimal solution of a previously solved LP relaxation of the problem. The LP/Quadratic Solver can also generate Reduce and Split cuts, which are variants of Gomory cuts.
- **Mixed Integer Rounding Cuts**: These cuts are produced by rounding the coefficients for integer variables and the right hand sides of constraints. In contrast to most other cuts, Mixed Integer Rounding cuts are specifically designed for problems with general integer variables, not just binary variables.

- **Clique Cuts**: Cuts for both row cliques and start cliques are generated, using a method due to Hoffman and Padberg.
- **Flow Cover Cuts**: These cuts may be generated from constraints that include continuous variables with upper bounds that can be zero or positive, depending on the setting of associated binary variables. Flow Cover cuts are useful only for mixed-integer problems, with at least some continuous variables.

Heuristics

VBA / SDK: Parameter Name "Heuristics", 1 = Automatic, 2 = None, 3 = Aggressive

Use this option to determine the extent of Heuristic strategies used by the LP/Quadratic Solver on LP/MIP (linear mixed-integer) problems. Select from **None**, **Aggressive**, and **Automatic** (the default).

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. Heuristics used by the LP/Quadratic Solver include:

- **Local Tree Search**: When a new incumbent solution is bound, the Solver alters its normal node selection strategy to look for other possible integer solutions "nearby" in the Branch & Bound tree.
- **Rounding Heuristic**: This heuristic is used to seek possible integer solutions (by adjusting the values of individual integer variables) in the "vicinity" of a known integer solution.
- **Feasibility Pump**: This heuristic is designed to quickly find good incumbent solutions. These incumbents can themselves be good solutions, but they also help the Solver prune the overall search of the Branch & Bound tree. This method is applied only once, after all cuts have been added to the problem. It is a relatively expensive procedure in computational terms, but it frequently yields substantial speedup.
- **Greedy Cover Heuristic**: This heuristic seeks to quickly find an integer feasible solution by testing combinations of values for binary integer variables in linear constraints with right hand sides of 1. Problems with constraints of this form are often called set covering, set packing, or set partitioning problems; this heuristic is most effective on such problems.

Thread Mode

VBA / SDK: Parameter Name "Thread_Mode"
0 - Use branching, numberThreads at a time (default)
1 - Use deterministic, numberThreads at a time
2 - Use root cuts, numberThreads at a time
8 - Use heuristics, numberThreads at a time
9 - No Threads, use one thread at a time.

Determines how the threads are used when a mixed integer model is being solved. If N equals the "Number of Threads", the collection of nodes will be solved using the selected mode, N threads at a time.

Number of Threads

VBA / SDK: Parameter Name "numberThreads", Default value = number of available system processors; 0 – Turns parallel processing off

The number of parallel threads of execution to be used when solving mixed-integer problems. Enter an integer from 1 to the Number of available system processors. The default is the number of system processors. A value of 0 turns parallel processing off.

Additional Options Available in VBA / SDK

If you want more fine-grained control over the behavior of the Large-Scale LP/QP Solver on linear mixed-integer problems when using *either Desktop Analytic Solver or Solver SDK*, you can use the options described below. They override the general settings obtained with the Preprocessing, Cuts and Heuristics options above. To use these options, you should have a good deal of experience solving linear mixed-integer problems.

Use Strong Branching

VBA / SDK: Engine.Params("StrongBranching"), value 1-Use Strong Branching or 0-Don't Use Strong Branching

When this option is set to 1, 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.

Maximum Cut Passes

VBA / SDK: Engine.Params("MaxRootCutPasses"), integer value >= -1
Engine.Params("MaxTreeCutPasses"), integer value >= -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 options (see below) is selected. 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 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 for this option.

Knapsack Cuts

VBA / SDK: Engine.Params("KnapsackCuts"), value 1-Generate These Cuts or 0-Don't Generate These Cuts

When this option is set to 1, 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: Engine.Params("GomoryCuts"), value 1-Generate These Cuts or 0-Don't Generate These Cuts

When this option is set to 1, 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 Large-Scale LP/QP 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.

Mixed Integer Rounding Cuts

VBA / SDK: Engine.Params("MirCuts"), value 1-Generate These Cuts or 0-Don't Generate These Cuts

When this option is set to 1, Mixed Integer Rounding cuts may be generated. These cuts are produced by rounding the coefficients for integer variables and the right hand sides of constraints. In contrast to most other cuts, Mixed Integer Rounding cuts are specifically designed for problems with general integer variables, not just binary variables.

Probing Cuts

VBA / SDK: Engine.Params("ProbingCuts"), value 1-Generate These Cuts or 0-Don't Generate These Cuts

When this option is set to 1, Probing cuts may be generated. This process is similar to the Preprocessing and Probing methods used in the Large-Scale SQP 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.

Two Mixed Integer Rounding Cuts

VBA / SDK: Engine.Params("TwoMirCuts"), value 1-Generate These Cuts or 0-Don't Generate These Cuts

When this option is set to 1, Two Mixed Integer Rounding cuts may be generated. The procedure used to obtain these cuts is a two-step variation of the one-step procedure used to generate Mixed Integer Rounding cuts.

Clique Cuts

VBA / SDK: Engine.Params("CliqueCuts"), value 1-Generate These Cuts or 0-Don't Generate These Cuts

When this option is set to 1, Clique cuts may be generated. Cuts for both row cliques and start cliques are generated, using a method due to Hoffman and Padberg.

Reduce and Split Cuts

VBA / SDK: Engine.Params("RedSplitCuts"), value 1-Generate These Cuts or 0-Don't Generate These Cuts

When this option is set to 1, Reduce and Split cuts may be generated. These cuts are variants of Gomory cuts.

Flow Cover Cuts

VBA / SDK: Engine.Params("FlowCoverCuts"), value 1-Generate These Cuts or 0-Don't Generate These Cuts

When this option is set to 1, Flow Cover cuts may be generated from constraints that include continuous variables with upper bounds that can be zero or positive, depending on the setting of associated binary variables. Flow Cover cuts have no effect in a ‘pure integer’ problem with no continuous variables.

Local Tree

VBA / SDK: Engine.Params("LocalTree"), value 1-Search the Local Tree or 0-Don’t Search the Local Tree

When this option is set to 1, each time a new best integer solution (an “incumbent”) is found, a search is conducted for additional possible solutions at “nearby” nodes in the Branch & Bound tree. This heuristic can sometimes discover good integer solutions early in the solution process.

Special Ordered Sets

VBA / SDK: Engine.Params("SOSCuts"), value 1-Generate These Cuts or 0-Don’t Generate These Cuts

This strategy scans the model for constraints of the form $x_1 + x_2 + \ldots + x_n = 1$ 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.

Greedy Cover Heuristic

VBA / SDK: Engine.Params("GreedyCover"), value 1-Perform this Search or 0-Don’t Perform this Search

When this option is set to 1, a heuristic is used to search for “incumbent” solutions using the so-called Greedy Cover algorithm. This heuristic seeks to quickly find an integer feasible solution by finding variable values that satisfy constraints where a linear function of binary integer variables must be less than, equal to, or greater than 1. Problems with this type of constraint are often called set covering, set packing, or set partitioning problems; this heuristic is most effective on such problems.

Feasibility Pump Heuristic

VBA / SDK: Engine.Params("FeasibilityPump"), value 1-Perform this Search or 0-Don’t Perform this Search

When this option is set to 1, a heuristic is used that is designed to quickly find good “incumbent” solutions. These incumbents can themselves be quite good solutions, and they also speed up the Branch & Bound search by eliminating many other subproblems. This method is applied only once, after initial cuts have been added to the problem. It is a relatively expensive procedure computationally, but it can yield enormous time savings on many problem.

Local Search Heuristic

VBA / SDK: Engine.Params("LocalHeur"), value 1-Use This Heuristic or 0-Don’t Use This Heuristic

When this option is set to 1, a “local search” heuristic 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: Engine.Params("RoundingHeur"), value 1-Use This Heuristic or 0-Don’t Use This Heuristic

When this option is set to 1, 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.

Large-Scale GRG Solver Options

SDK: Engine name "Large-Scale GRG Solver", file name LSGRGeng.dll

This Solver Engine supports the Common Solver Options discussed earlier, plus the Global Optimization options, and the Convergence, Population Size, Random Seed, Recognize Linear Variables, Relax Bounds on Variables options, Estimates, Derivatives and Search options, which are specific to the Large-Scale GRG Solver.

The Global Optimization options and the Population Size and Random Seed options are discussed below in the section “Options for Global Optimization.” The other options specific to the Large-Scale GRG (LSGRG) Solver are discussed here.

Convergence

VBA / SDK: Engine.Params("Convergence"), 0 <= value <= 1

As discussed in the chapter “Solver Result Messages,” the LSGRG 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 LSGRG Solver stops if the absolute value of the *relative* change in the objective function is less than the value of the Convergence option for the last few iterations. While the default value of 1.0E-4 (0.0001) is suitable for most problems, it may be too large for some models, causing the LSGRG Solver to stop prematurely when this test is satisfied, instead of continuing for more iterations 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 Convergence option to a smaller value such as 1.0E-5 or 1.0E-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: Engine.Params("RecognizeLinear"), value 1-Recognize or 0-Don’t Recognize

This option has only a limited effect in the LSGRG engine. It is intended to take advantage of the fact that in many nonlinear problems, some of the variables occur linearly in the objective and all of the constraints. Hence the partial derivatives of the problem functions with respect to these variables are constant, and need not be re-computed on each iteration. When *finite differencing* is used to estimate partial derivatives, this means that the steps involved in perturbing such a variable and computing values for all of the problem functions can be skipped.

In Analytic Solver Desktop when the PSI Interpreter is used (which is the default) and always in Analytic Solver Cloud, using this option will not save any time, because partial derivatives are computed via *automatic differentiation* rather than *finite differencing*. Similarly, in Solver SDK, when you supply an Evaluator to compute derivatives, this option won’t save any time. Further, the PSI Interpreter can supply “dependents analysis” information, which the LSGRG Solver will use to recognize variables that occur linearly in only *some* (as well as all) of the problem functions.

Relax Bounds on Variables

VBA / SDK: Engine.Params("RelaxBounds"), value 1-Relax or 0-Don’t Relax

By default (and *unlike* the nonlinear GRG Solver bundled within Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK), the LSGRG Solver ensures that any trial points evaluated during the solution process will not have values that violate the bounds on the variables you specify, even by a small amount. If your problem functions cannot be evaluated for values outside the variable bounds, this default behavior will ensure that the solution process can continue. However, at times the LSGRG Solver can make more rapid progress along a given search direction by testing trial points with values slightly outside the bounds on the variables. If you want to permit this to happen, set this option to 1. If you receive the Solver Result Message “Solver encountered an error value in a target or constraint cell,” as a first step you should set this option to 0.

Other Nonlinear Options

The default values for the Estimates, Derivatives and Search options can be used for most problems. If you’d like to change these options to improve performance on your model, this section will provide some general background on how they are used by the LSGRG Solver.

On each major iteration, the LSGRG Solver requires values for the partial derivatives 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: Engine.Params("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:** This choice has no bearing on quadratic programming problems.

Derivatives

VBA / SDK: Engine.Params("Derivatives"), value 1-Forward or 2-Central

On each major iteration, the LSGRG Solver requires values for the partial derivatives of the objective and constraints (i.e. the Jacobian matrix). In Analytic Solver Desktop, these derivatives may be computed via *automatic differentiation* (when Interpreter = Psi Interpreter) or via *finite differencing* (when Interpreter = Excel Interpreter). In Solver SDK, if you supply an Evaluator for Eval_Type_Gradient, it is used to compute derivatives in a manner analogous to automatic differentiation; if you don't supply such an Evaluator, the LSGRG Solver computes derivatives via finite differencing – calling your function Evaluator (Eval_Type_Function) once or twice for each decision variable in order to do so. In Analytic Solver Cloud automatic differentiation is always used.

The Derivatives option in the LSGRG Solver Options dialog is relevant only if finite differencing is being used (which is never the case in Analytic Solver Cloud) to compute derivative values. It determines whether "forward differencing" or "central differencing" is performed.

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.

Search Option

VBA / SDK: Engine.Params("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. Instead, a direction is chosen through an estimation method. The default choice Newton uses a quasi-Newton (or BFGS) method, which maintains an *approximation* to the Hessian matrix; this requires more storage (an amount proportional to the square of the number of currently binding constraints) but performs very well in practice. The alternative choice Conjugate uses a conjugate gradient method, which does not require storage for the Hessian matrix and still performs well in most cases. The choice you make here is not crucial, since the LSGRG solver is capable of switching *automatically* between the quasi-Newton and conjugate gradient methods depending on the available storage.

Large-Scale SQP Solver Options

SDK: Engine name "Large-Scale SQP Solver", file name LSSQPeng.dll

This Solver Engine supports the Common Solver Options discussed earlier, plus the the Global Optimization options, and the Convergence, Population Size, Mutation Rate, Random Seed, Local Search, Treat Constraints as Linear, Treat Objective as Linear, Derivatives options, which are specific to the Large-Scale SQP Solver.

Use of Convergence, Population Size and Random Seed

If the problem is diagnosed as *non-smooth*, the Large-Scale SQP Solver uses the Evolutionary Solver as a "top-level" or global search algorithm, and uses SQP methods for local searches (when the Local Search option is set to Gradient Local or Automatic Choice). In this case, the Convergence, Population Size and Random

Seed options apply to the Evolutionary Solver, and the default Convergence value of 1.0E-6 is used for local searches by the SQP Solver. If the problem is diagnosed as *smooth*, then the Evolutionary Solver is not used. In this case, the Convergence option is used by the SQP Solver as described immediately below, and the Population Size and Random Seed options apply to the multistart methods controlled by the Global Optimization option group, if they are used.

The Global Optimization options, and use of the Population Size and Random Seed options by the multistart methods are discussed below in the section "Options for Global Optimization." Other Large-Scale SQP Solver options are discussed here.

Convergence

VBA / SDK: Engine.Params("Convergence"), 0 <= value <= 1

As discussed in the chapter "Solver Result Messages," the LSSQP Solver will stop and display the message "Solver has converged to the current solution" when the solution is changing very slowly for the last few iterations or trial solutions. More precisely, the LSSQP Solver stops if the *maximum normalized complementarity gap of the variables* is less than the value of the Convergence option for the last few iterations. Smaller Convergence values may produce more accurate results, but will require more computing time. The default value of 1.0E-6 (0.000001) is suitable for a wide range of problems.

If you are getting this message when you are seeking a locally optimal solution, you can change the Convergence option value, 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.

Treat Constraints as Linear/LinearConstraints

VBA / SDK: Engine.Params("LinearConstraints"), value 1-Treat as Linear or 0-Don't Treat as Linear

The Treat Constraints as Linear and Treat Objective as Linear options are used only if (i) you have a problem with all linear constraints and possibly a linear objective but (ii) you *aren't* using the PSI Interpreter in Analytic Solver Desktop, or you *aren't* setting the Model object AllGradDepend property, or calling the SolverModAllGradDependSet procedural API function, in Solver SDK. (This option has no relevance in Analytic Solver Cloud.) These options tell the LSSQP Solver to treat the constraints and/or the objective as linear, and exploit this treatment to solve the problem more efficiently. If you are solving a problem with all linear constraints, such as a linear programming (LP) or quadratic programming (QP) problem, you can use this option to speed up the solution process.

In Analytic Solver Desktop, use of the PSI Interpreter is the *default* setting, the LSSQP Solver obtains information about linearity of the constraints and objective from the Polymorphic Spreadsheet Interpreter, and these options are ignored.

When you solve a problem Interpreter = Excel Interpreter in Analytic Solver Desktop, or without setting the Model object AllGradDepend property in Solver SDK, the Large-Scale SQP Solver normally treats the objective and all constraints as smooth nonlinear functions – even if they are actually linear. Hence, the Solver will compute gradients of these functions at each major iteration. If this option is set to 1, the LSSQP Solver will *assume* that the constraints are linear (hence their gradients are constant) and will compute their gradients only once, at the beginning of the solution process. This option is especially effective when used in combination with the “Treat Objective as Linear” option, for a linear programming problem, as explained below.

Treat Objective as Linear/Linear Objective

VBA / SDK: Engine.Params("LinearObjective"), value 1-Treat as Linear or 0-Don’t Treat as Linear

If you are solving a linear programming problem, you can use this option to greatly speed up the solution process. As described under “Treat Constraints as Linear” above, when you solve a problem with Interpreter = Excel Interpreter in Analytic Solver Desktop or without setting the Model object AllGradDepend property in Solver SDK, the Large-Scale SQP Solver normally treats the objective and all constraints as smooth nonlinear functions, and computes gradients of these functions via finite differencing at each major iteration. (This option has no relevance in Analytic Solver Cloud.) When both options “Treat Constraints as Linear” and “Treat Objective as Linear” are True, the LSSQP Solver computes the function gradients only once, at the beginning of the solution process – taking about the same amount of time as “Problem Setup” in a large-scale LP Solver. The Large-Scale SQP Solver then solves the problem using active-set methods that are highly effective, and competitive with the Simplex method for linear programming.

If you set only “Treat Constraints as Linear,” but not “Treat Objective as Linear” to True, the improvement in solution time depends on whether or not you are using *automatic differentiation* in Analytic Solver Desktop, or you are supplying an Evaluator for Eval_Type_Gradient in Solver SDK. (*Automatic differentiation* is always used in Analytic Solver Cloud.) If automatic differentiation or an Evaluator for derivatives are being used, you should see a significant speedup, since only derivatives for the *objective* need be computed on each major iteration. But if finite differencing is being used (Interpreter = Excel Interpreter in Analytic Solver Desktop, or you’ve defined only an Evaluator for Eval_Type_Function), you may not see much speed improvement, especially in Excel. This is because it takes almost as much time to compute gradients of *some* of the functions (or even just one, the objective) as it does to compute gradients of *all* of the functions, by recalculating the Excel spreadsheet or by calling your function Evaluator with perturbed values for each of the decision variables.

Derivatives

VBA / SDK: Engine.Params("Derivatives"), value 1-Forward or 2-Central

On each major iteration, the LSSQP Solver requires values for the partial derivatives of the objective and constraints (i.e. the Jacobian matrix). In Analytic Solver Comprehensive and Analytic Solver Optimization, these derivatives may be computed via *automatic differentiation* or via *finite differencing*. In Solver SDK, if you supply an Evaluator for Eval_Type_Gradient, it is used to compute derivatives in a manner analogous to automatic differentiation; if you don’t supply such an Evaluator, the LSSQP Solver computes derivatives via finite differencing – calling your function Evaluator (Eval_Type_Function) once or twice for each decision variable in order to do so.

The Derivatives option in the LSSQP Solver Options dialog is relevant only if finite differencing is being used to compute derivative values. It determines whether “forward differencing” or “central differencing” is performed.

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.

Model Based Search

VBA / SDK: Engine.Params("ModelBasedSearch"), value 0-None, 1-CPU Based, 2 – GPU Based

This option takes effect only when the Global Search option is set to Scatter Search. When this option is set to "None", the Scatter Search algorithm is used without any Model Based Local Search. When this option is set to either "CPU Based" or "GPU Based", an internal model of the problem is created (using *radial basis functions*) which closely fits the original problem in the search region. The Evolutionary Solver then uses this internal model to evaluate many points in parallel (either on the CPU or GPU - depending on the option setting) rather than using Excel or the PSI Interpreter to evaluate each of these points sequentially. Only the most promising of these points are evaluated against the actual spreadsheet model. The most promising points from this evaluation are added to the population of best solutions. This new search method typically results in better solutions in less time when compared to using only the Scatter Search algorithm.

Most new computers now contain GPUs or Graphics Processing Units, either on custom chips (in most laptops) or plug-in cards. These devices are typically used to quickly build and manipulate display images for video games and other graphics. However, their ability to perform floating point arithmetic in parallel can be harnessed for computational purposes. Some GPU cards or chips contain 256, 512 or more special-purpose processors; in comparison, the computer's main CPU usually contains just 2 to 4 general-purpose processors If your computer includes an Nvidia or AMD Radeon graphics card or chip, it would be worth your time to compare the performance of both the CPU and GPU based approaches. You very well could find that the GPU based search results in a solution in significantly less time than the CPU based search.

Large-Scale SQP Evolutionary Solver Options

If the problem is diagnosed as non-smooth (NSP), the options described in this section control the operation of the Evolutionary Solver "top-level" algorithm.

Convergence

VBA / SDK: Engine.Params("Convergence"), 0 <= value <= 1

The LSSQP/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. The stopping condition is satisfied if 99% of the population members all have fitness values that are within the Convergence tolerance of each other.

Population Size

VBA / SDK: Engine.Params("PopulationSize"), integer value >= 0

This option sets the number of candidate solutions in the population. 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).

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: Engine.Params("MutationRate"), 0 < value < 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. 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.

Random Seed

VBA / SDK: Engine.Params("RandomSeed"), integer value > 0

The LSSQP/Evolutionary Solver makes extensive use of random sampling. Because of these random choices, the LSSQP/Evolutionary Solver will normally find 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, set the Random Seed option to a positive integer value; this value will then be used to "seed" the random number generator each time you click Solve.

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 LSSQP/Evolutionary Solver. As noted under the Mutation rate option, a "generation" or subproblem in the LSSQP/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. Choosing Automatic Choice (the default), will select 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 LSSQP/Evolutionary Solver using 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.

Filtered Local Search

In the LSSQP/Evolutionary Engine, 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 LSSQP/Evolutionary Solver 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 LSSQP/Evolutionary Solver’s mutation and crossover operations.

In addition to the Local Search options, the LSSQP/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 LSSQP/Evolutionary Solver is unusually strong in its ability to deal with a combination of constraints and non-smooth functions.

If the LSSQP/Evolutionary Solver stops with the message "Solver encountered an error computing derivatives," you should set Use Sparse Variables to True in the Solver Model dialog Options tab, and Solve again.

Fix Nonsmooth Variables

VBA / SDK: Parameter Name "FixNonSmooth", value 1/True or 0/False

In the LSSQP/Evolutionary Solver, this option determines how non-smooth variable occurrences in the problem will be handled during the local search step. 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 model.

Global Search

VBA / SDK: Parameter Name "GlobalSearch", value - 1/Genetic Algorithm or 0/Scatter Search

If this option is set to Genetic Algorithm, then the Evolutionary Solver will use methods from the literature on genetic algorithms (its traditional methods) to solve the model. Otherwise, the Evolutionary Solver will use methods from the literature on scatter search. On some models, the scatter search algorithm will result in better answers in less time when compared to the genetic algorithm. However, for other models, the genetic algorithm may be more successful. Since the scatter search algorithm tends perform best, by a modest margin, on the majority of models, it is the default choice. But we suggest you try both algorithms with your model to see which works better for you.

Large-Scale SQP Mixed-Integer Options

Use of the options in this section 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: Engine.Params("ProbingFeasibility"), value 1-Use Probing/Feasibility or 0-Don't Use Probing Feasibility

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:

$x_1 + x_2 + x_3 + x_4 + x_5 \leq 1$

where x_1 through x_5 are all binary integer variables. Whenever one of these variables is fixed at 1, all of the others are forced to be 0; Probing allows the Solver to determine this *before* solving the problem. In some cases, the Feasibility tests performed as part of Probing will determine that the subproblem is infeasible, so it is unnecessary to solve it at all. This is a special case of a “clique” or “Special Ordered Set” (SOS) constraint.

Bounds Improvement

VBA / SDK: Engine.Params("BoundsImprovement"), value 1-Use Bounds Improvement or 0-Don’t Use Bounds Improvement

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 Solver 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: Engine.Params("OptimalityFixing"), value 1-Use Optimality Fixing or 0-Don’t Use Optimality Fixing

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 >= 10 and A1:A5 <= 10, which create “implied” equalities**, instead of explicit equalities such as A1:A5 = 10. Watch out for situations such as A1:A5 >= 10 and A3:D3 <= 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.

Variable Ordering and Pseudocost Branching

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.

The Solver's Branch & Bound method computes *pseudocosts,* which 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 the Task Pane Model tab, or the order in which they appear in the array arguments that you pass to API functions in Solver SDK. You may be able to improve performance by manually ordering the decision variables, based on your knowledge of the problem.

Cut Generation

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 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 for these options to find the best settings for your problem.

Max Knapsack Cuts

VBA / SDK: Engine.Params("MaxKnapsackCuts"), integer value >= 0

This option sets 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 an active set (Simplex) method. Knapsack cuts, also known as *lifted cover inequalities,* can be generated only for groups of binary 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: Engine.Params("KnapsackPasses"), integer value >= 0

This option sets the number of "passes" the Solver should make over a given subproblem, looking for Knapsack 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.

Max Gomory Cuts

VBA / SDK: Parameter Name "MaxGomoryCuts", integer value > 0

This option sets 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 set the Use Automatic Scaling option to True.** If you see the Scaling Report listed as an option in the Reports – Optimization gallery, select it and examine the report contents to help find scaling problems in your model. If you have trouble finding the integer optimal solution with the default settings for Gomory cuts, you may want to enter 0 for this option, to eliminate Gomory cuts as a possible source of problems due to rounding.

Max Gomory Passes

VBA / SDK: Parameter Name "GomoryPasses", integer value > 0

This option sets 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’s possible to solve an LP/MIP problem to optimality by generating Gomory cuts in multiple passes, without any branching via Branch & Bound; however, experience has shown that this is usually less efficient than using Branch & Bound. The default value of 1 pass is best for many models, but you may find that increasing this value improves solution time for your model.

MOSEK Solver Options

SDK: Engine name "MOSEK Solver Engine", file name MOSEKeng.dll

This Solver Engine supports the Common Solver Options discussed earlier, plus the Pivot Tolerance, Ordering Strategy, Scaling, LP/QP/QCP Tolerances, Conic Tolerances, and Nonlinear Tolerances, which are specific to the MOSEK Solver. The Precision option has a somewhat specialized meaning for this Solver engine, as discussed below.

Precision

VBA / SDK: Engine.Params("Precision"), 0 < value < 1

This parameter determines both the maximum absolute primal bound violation and the maximum absolute dual bound violation in an optimal basic solution. The default value of 1E-6 is appropriate for most problems.

Pivot Tolerance

VBA / SDK: Engine.Params("PivotTolerance"), 0 < value < 1

This is the relative pivot tolerance used in the LU factorization, in the basis identification procedure. Any positive number up to 0.999999 can be used.

Ordering Strategy/Ordering

VBA / SDK: Engine.Params("Ordering"), value 1-Automatic Choice, 2-Local Fill-in 1, 3-Local Fill-in 2, 4-Graph Partitioning, 5-Alt. Graph Partitioning, 6-No Ordering

This option group determines the column ordering strategy used by the interior-point optimizer when factorizing the Newton equation system. The options are:

Automatic Choice

The ordering strategy is chosen automatically based on matrix characteristics.

Local Fill-in 1

An approximate minimum local-fill-in ordering is used.

Local Fill-in 2

A variant of minimum local-fill-in ordering is used.

Graph Partitioning

The ordering is determined by a graph partitioning algorithm.

Alt. Graph Partitioning

The ordering is determined by an alternative graph partitioning algorithm.

No Ordering

The columns of the matrix are not reordered.

Scaling

VBA / SDK: Engine.Params("Scaling"), value 1-Automatic Choice, 2-Aggressive Scaling, 3-No Scaling, 4-Conservative Scaling

This option group determines how the problem is scaled before the interior-point optimizer is used. The options are:

Automatic Choice

The degree of scaling is chosen automatically.

Aggressive Scaling

Automatic scaling is "aggressive" – the magnitudes of problem values (e.g. coefficients and right hand sides of constraints) may be adjusted by large amounts to make the Newton equation matrix well-conditioned.

No Scaling

The problem values are used as-is, with no automatic rescaling.

Conservative Scaling

Automatic scaling is "conservative" – the magnitudes of problem values (e.g. coefficients and right hand sides of constraints) may be adjusted by small amounts to make the Newton equation matrix better-conditioned.

Maximum Barrier Iterations

VBA / SDK: Engine.Params("MaxBarrierIterations"), 1 <= value

The value of the Maximum Barrier Iterations option determines the maximum number of iteations that will be performed by the interior point optimizer. For the vast majority of models, the default setting of 100 iterations is appropriate.

LP/QP/QCP Tolerances

Dual Feasibility Tolerance/DualFeasibility

VBA / SDK: Engine.Params("DualFeasibility"), 0 < value < 1

This is the dual feasibility tolerance used for linear and quadratic optimization problems. The default value is appropriate for most problems.

Primal Feasibility Tolerance/PrimalFeasibility

VBA / SDK: Engine.Params("PrimalFeasibility"), 0 < value < 1

This is the primal feasibility tolerance used for linear and quadratic optimization problems. The default value is appropriate for most problems.

Model Feasibility Tolerance/ModelFeasibility

VBA / SDK: Engine.Params("ModelFeasibility"), 0 < value < 1

This tolerance determines when the optimizer declares the model to be primal or dual infeasible. Smaller values cause the optimizer to take more time and be more "conservative" in declaring the model infeasible.

Complementarity Gap Tolerance/CompGapTol

VBA / SDK: Engine.Params("CompGapTol"), 0 < value < 1

This is a relative tolerance for the complementarity gap in the interior point method, for linear and quadratic optimization problems.

Central Path Tolerance/CentralPathTol

VBA / SDK: Engine.Params("CentralPathTol"), 0 < value < 0.5

This tolerance determines how closely the interior-point optimizer follows the central path. A larger value of this parameter causes the optimizer to follow the central path more closely (the value must be less than 0.5). On numerically unstable problems, consider increasing this tolerance from its default value.

Gap Termination Tolerance/GapTerminationTol

VBA / SDK: Engine.Params("GapTerminationTol"), 0 < value < 0.5

This is a relative tolerance for the primal-dual optimality gap, for linear and quadratic optimization problems.

Relative Step Size

VBA / SDK: Engine.Params("StepSize"), 0 < value < 1

This is the relative step size to the constraint boundary used by the interior point optimizer for linear and quadratic optimization problems.

Conic Tolerances

Dual Feasibility Tolerance/DualFeasibilityCone

VBA / SDK: Engine.Params("DualFeasibilityCone"), 0 < value < 1

This is the dual feasibility tolerance used for conic optimization problems. The default value is appropriate for most problems.

Primal Feasibility Tolerance/PrimalFeasibilityCone

VBA / SDK: Engine.Params("PrimalFeasibilityCone"), 0 < value < 1

This is the primal feasibility tolerance used for conic optimization problems. The default value is appropriate for most problems.

Model Feasibility Tolerance/ModelFeasibilityCone

VBA / SDK: Engine.Params("ModelFeasibilityCone"), 0 < value < 1

This tolerance determines when the optimizer declares the model to be primal or dual infeasible. Smaller values cause the optimizer to take more time and be more “conservative” in declaring the model infeasible.

Complementarity Gap Tolerance/CompGapTolCone

VBA / SDK: Engine.Params("CompGapTolCone"), 0 < value < 1

This is a relative tolerance for the complementarity gap in the interior point method, for conic optimization problems.

Gap Termination Tolerance/GapTerminationTolCone

VBA / SDK: Engine.Params("GapTerminationTolCone"), 0 < value < 1

This is a relative tolerance for the primal-dual optimality gap, for conic optimization problems.

Note on Solving Nonlinear Models

The Mosek Engine V2023 *Q3 and later* does not support the solving of nonlinear models.

OptQuest Solver Options

SDK: Engine name "OptQuest Solver", file name OPTQeng.dll

This Solver Engine supports the Max Time, Iterations, Show Iteration Results, Assume Non-Negative, and Bypass Solver Reports options in common with the other Solver Engines, and other options specific to the OptQuest Solver that are described in this section.

Precision (Obj Fun)/ObjPrecision

VBA / SDK: Engine.Params("ObjPrecision"), 0 < value < 1

This option determines what the OptQuest Solver considers a significant improvement in the objective function; in other words, it helps the search determine what progress means.

If the solution process seems to be spending too much time on tiny improvements that don’t matter to you, then you should make the Objective Function Precision a larger number. On the other hand, if the solution process is not finding any improvements, then perhaps it needs to take smaller steps to make progress – in which case you should make the Objective Function Precision a smaller number.

The Objective Function Precision can be set to a number between 0 and 1. If the number that you consider suitable for a significant improvement does not fall into this range, then the model should be rescaled. For example, if you consider an improvement of 5 or more in the objective function value to be worth keeping, then you can take the existing objective function and divide it by 10 to create a rescaled objective function; in the rescaled version, then, an improvement of 0.5 would be considered worthwhile.

Be cautious about using very small values for this precision setting: Such an approach could cause a reduction in the diversity in the population.

The default value for the Objective Function Precision is 0.0001.

Precision (Variable)/VarPrecision

VBA / SDK: Engine.Params("VarPrecision"), 0 < value < 1

This option helps the OptQuest Solver determine if a new solution it has just generated is essentially the same as a solution it has previously evaluated. The purpose of this check is to avoid spending time evaluating a new solution that is basically equivalent to a previous solution.

The OptQuest Solver makes this determination using the following steps: First, it multiplies each decision variable's range (its upper bound minus its lower bound) by the Decision Variable Precision setting value, thus computing a number that it considers a significant difference in the value of each decision variable. Then it compares the decision variable values in the new solution to the decision variable values in each previous solution. If it finds a previous solution in which none of the decision variables have a significant difference from the values in the new solution, then it considers these solutions essentially the same, and discards the new solution without wasting time evaluating it.

The Decision Variable Precision setting is only active when the Check for Duplicated Solutions option is set to 1/True. If it is set to 0/False, then the OptQuest Solver does not attempt to determine if the new solution is essentially the same as a solution previously evaluated, and the Decision Variable Precision edit box is greyed out.

The default value for the Decision Variable Precision is 0.0001.

Number of Solutions to Report/NumSolutions

VBA / SDK: Engine.Params("NumSolutions"), 1 <= integer value <= 10,000

This option determines the number of solutions (sets of values for the decision variables) that will appear in the Solutions Report, if you select this report at the end of an optimization. The solutions in the report are the *N* best solutions from the final population, where *N* is the value of this option; they are ordered first by feasibility (with all feasible solutions preceding any infeasible solutions), then by objective value.

Use a Fixed Seed/FixedSeed

VBA / SDK: Engine.Params("FixedSeed"), value 1-Use a Fixed Seed or 0-Don't Use a Fixed Seed

The OptQuest Solver uses an element of randomness when generating new solutions to ensure diversity. If you need to duplicate the results of the optimization search at a later date, the OptQuest Solver must be able to duplicate the sequence of random numbers it used originally. This can be accomplished via this option.

When this option is set to 1/True (the default setting), you specify a fixed seed value for the OptQuest Solver's random number generator (via the Random Seed option below). Then, in the future, you can specify the same fixed seed value in order to reproduce the optimization run.

When this option is set to 0/False, the OptQuest Solver will use a different sequence of random numbers each time it is run, which means that you may get different results on each run even if the model has not changed.

Random Seed

VBA / SDK: Engine.Params("RandomSeed"), integer value > 0

When the Use a Fixed Seed option is set to 1/True, the integer value of this option is used to seed the OptQuest Solver's random number generator.

Check for Duplicated Solutions

VBA / SDK: Engine.Params("CheckDup"), value 1/True or 0/False

The Check for Duplicated Solutions option is designed to speed up the optimization process by eliminating from consideration new solutions that are essentially the same as previous solutions that were already evaluated. The OptQuest Solver uses the Precision (Dec Var) setting when comparing each new solution to the previous solutions, as more fully described under the "Precision (Dec Var)" option above. The default setting is 1/True.

When the number of trial solutions already evaluated is not large, the OptQuest Solver can check for duplicated solutions quickly, so the benefit outweighs the cost. However, when the number of evaluated solutions is very large, then the time it takes to compare the new solution to all the previous ones outweighs the benefit. As a general guideline, use this feature only when the setting for Iterations is less than 100,000.

Auto Stop

VBA / SDK: Engine.Params("AutoStop"), value 1-Stop When Objective Hasn't Improved or 0-Don't Stop

With this option, you can cause the OptQuest Solver to stop automatically when it reaches a point where it is making slow or no progress. When this option is set to 1/True, the Solver will stop and return the best solution found so far, if no solution with a better objective value (determined by the Auto Stop Percentage) has been found for the last *N* iterations, where *N* is the value of the Auto Stop Iterations option below. The default setting is 0/False. This is the *only* way to cause the OptQuest Solver to stop automatically based on solution values; otherwise it stops only when ESC is pressed or its Max Time or Iterations limit is reached.

Auto Stop Solutions/AutoStopSolutions

VBA / SDK: Engine.Params("AutoStopSolutions"), integer value > 0

When the Auto Stop option is set to 1/True, this option sets the maximum number of iterations the Solver should run without finding an improved objective function value (determined by the setting for Auto Stop Percentage) before it stops and returns the best solution found so far. The default value is 500.

Auto Stop Percentage

VBA / SDK: Engine.Params("AutoStopPercentage"), value > 0

When the Auto Stop option is set to 1/True, OptQuest will stop when the Auto Stop Solutions are explored and found to have a cumulative improvement smaller than this percentage.

Use PSI Linearization

VBA / SDK: Engine.Params("UsePsi"), value > 0

When the Use Psi Linearization option is set to 1/True, OptQuest will use the Psi Interpreter to generate 1st derivatives in order to detect any linear constraints in the model. If False, all constraints are assumed to be non-smooth.

Check Nonlinear Constraints

VBA / SDK: Engine.Params("CheckNonlinear"), value > 0

When the Check Nonlinear Constraints option is set to 1/True, OptQuest will utilize nonlinear constraints to generate solutions. This may cause an increase in the amount of time to generate solutions. If you are finding that OptQuest is taking an unusually long time to finish, set this option to False and resolve.

OptQuest Solver Options

SDK: Engine name "OptQuest Solver", file name OPTQeng.dll

This Solver Engine supports the Max Time, Iterations, Show Iteration Results, Assume Non-Negative, and Bypass Solver Reports options in common with the other Solver Engines, and other options specific to the OptQuest Solver that are described in this section.

Precision (Obj Fun)/ObjPrecision

VBA / SDK: Engine.Params("ObjPrecision"), 0 < value < 1

This option determines what the OptQuest Solver considers a significant improvement in the objective function; in other words, it helps the search determine what progress means.

If the solution process seems to be spending too much time on tiny improvements that don’t matter to you, then you should make the Objective Function Precision a larger number. On the other hand, if the solution process is not finding any improvements, then perhaps it needs to take smaller steps to make progress – in which case you should make the Objective Function Precision a smaller number.

The Objective Function Precision can be set to a number between 0 and 1. If the number that you consider suitable for a significant improvement does not fall into this range, then the model should be rescaled. For example, if you consider an improvement of 5 or more in the objective function value to be worth keeping, then you can take the existing objective function and divide it by 10 to create a rescaled objective function; in the rescaled version, then, an improvement of 0.5 would be considered worthwhile.

Be cautious about using very small values for this precision setting: Such an approach could cause a reduction in the diversity in the population.

The default value for the Objective Function Precision is 0.0001.

Precision (Variable)/VarPrecision

VBA / SDK: Engine.Params("VarPrecision"), 0 < value < 1

This option helps the OptQuest Solver determine if a new solution it has just generated is essentially the same as a solution it has previously evaluated. The purpose of this check is to avoid spending time evaluating a new solution that is basically equivalent to a previous solution.

The OptQuest Solver makes this determination using the following steps: First, it multiplies each decision variable's range (its upper bound minus its lower bound) by the Decision Variable Precision setting value, thus computing a number that it considers a significant difference in the value of each decision variable. Then it compares the decision variable values in the new solution to the decision variable values in each previous solution. If it finds a previous solution in which none of the decision variables have a significant difference from the values in the new solution, then it considers these solutions essentially the same, and discards the new solution without wasting time evaluating it.

The Decision Variable Precision setting is only active when the Check for Duplicated Solutions option is set to 1/True. If it is set to 0/False, then the OptQuest Solver does not attempt to determine if the new solution is essentially the same as a solution previously evaluated, and the Decision Variable Precision edit box is greyed out.

The default value for the Decision Variable Precision is 0.0001.

Number of Solutions to Report/NumSolutions

VBA / SDK: Engine.Params("NumSolutions"), 1 <= integer value <= 10,000

This option determines the number of solutions (sets of values for the decision variables) that will appear in the Solutions Report, if you select this report at the end of an optimization. The solutions in the report are the *N* best solutions from the final population, where *N* is the value of this option; they are ordered first by feasibility (with all feasible solutions preceding any infeasible solutions), then by objective value.

Use a Fixed Seed/FixedSeed

VBA / SDK: Engine.Params("FixedSeed"), value 1-Use a Fixed Seed or 0-Don't Use a Fixed Seed

The OptQuest Solver uses an element of randomness when generating new solutions to ensure diversity. If you need to duplicate the results of the optimization search at a later date, the OptQuest Solver must be able to duplicate the sequence of random numbers it used originally. This can be accomplished via this option.

When this option is set to 1/True (the default setting), you specify a fixed seed value for the OptQuest Solver's random number generator (via the Random Seed option below). Then, in the future, you can specify the same fixed seed value in order to reproduce the optimization run.

When this option is set to 0/False, the OptQuest Solver will use a different sequence of random numbers each time it is run, which means that you may get different results on each run even if the model has not changed.

Random Seed

VBA / SDK: Engine.Params("RandomSeed"), integer value > 0

When the Use a Fixed Seed option is set to 1/True, the integer value of this option is used to seed the OptQuest Solver's random number generator.

Check for Duplicated Solutions

VBA / SDK: Engine.Params("CheckDup"), value 1/True or 0/False

The Check for Duplicated Solutions option is designed to speed up the optimization process by eliminating from consideration new solutions that are essentially the same as previous solutions that were already evaluated. The OptQuest Solver uses the Precision (Dec Var) setting when comparing each new solution to the previous solutions, as more fully described under the "Precision (Dec Var)" option above. The default setting is 1/True.

When the number of trial solutions already evaluated is not large, the OptQuest Solver can check for duplicated solutions quickly, so the benefit outweighs the cost. However, when the number of evaluated solutions is very large, then the time it takes to compare the new solution to all the previous ones outweighs the benefit. As a general guideline, use this feature only when the setting for Iterations is less than 100,000.

Auto Stop

VBA / SDK: Engine.Params("AutoStop"), value 1-Stop When Objective Hasn't Improved or 0-Don't Stop

With this option, you can cause the OptQuest Solver to stop automatically when it reaches a point where it is making slow or no progress. When this option is set to 1/True, the Solver will stop and return the best solution found so far, if no solution with a better objective value (determined by the Auto Stop Percentage) has been found for the last *N* iterations, where *N* is the value of the Auto Stop Iterations option below. The default setting is 0/False. This is the *only* way to cause the OptQuest Solver to stop automatically based on solution values; otherwise it stops only when ESC is pressed or its Max Time or Iterations limit is reached.

Auto Stop Solutions/AutoStopSolutions

VBA / SDK: Engine.Params("AutoStopSolutions"), integer value > 0

When the Auto Stop option is set to 1/True, this option sets the maximum number of iterations the Solver should run without finding an improved objective function value (determined by the setting for Auto Stop Percentage) before it stops and returns the best solution found so far. The default value is 500.

Auto Stop Percentage

VBA / SDK: Engine.Params("AutoStopPercentage"), value > 0

When the Auto Stop option is set to 1/True, OptQuest will stop when the Auto Stop Solutions are explored and found to have a cumulative improvement smaller than this percentage.

Use PSI Linearization

VBA / SDK: Engine.Params("UsePsi"), value > 0

When the Use Psi Linearization option is set to 1/True, OptQuest will use the Psi Interpreter to generate 1st derivatives in order to detect any linear constraints in the model. If False, all constraints are assumed to be non-smooth.

Check Nonlinear Constraints

VBA / SDK: Engine.Params("CheckNonlinear"), value > 0

When the Check Nonlinear Constraints option is set to 1/True, OptQuest will utilize nonlinear constraints to generate solutions. This may cause an increase in the amount of time to generate solutions. If you are finding that OptQuest is taking an unusually long time to finish, set this option to False and resolve.

XPRESS Solver Options

SDK: Engine name "XPRESS Solver", file name XPRESSeng.dll

The Xpress Solver is a powerful tool for mathematical optimization, widely used for solving linear, integer, and quadratic programming problems. The default settings for this engine are typically adequate for most users, but engine options are available for those who wish to customize and optimize their solving experience. The settings are categorized into several sections, including:

- General: These options control the overall behavior of the solver, such as the algorithm to use, whether the model is quadratic, scaling and time limits.
- LP: The Xpress Engine offers a range of options specifically designed for linear programming (LP) problems, allowing users to tailor the solver's performance to their needs. Key LP options include settings that control numerical stability and solution accuracy.
- Presolve: Settings for presolve techniques can simplify the problem before optimization.
- Newton Barrier: In this section, users can determine whether the Newton Barrier method will Cross Over to the Simplex method near the solution, so that sensitivity analysis information can be obtained or set the Relative Gap Duality Tolerance, the most common stopping tolerance for the Barrier method.
- Integer: Here, users can select a conservative, aggressive, or automatically determined Cut Strategy, or you can disable generation of cuts. You can set absolute and relative Integer Tolerances and related parameters which determine how close the Solver must come to an integer optimal solution before stopping.

- Node Selection: This section allows you to select the basic strategy for choosing nodes (subproblems to be explored) by the Branch and Cut method.
- Heuristics: This section allows you to control the frequency and nature of heuristic methods used to find an integer solution early in the Branch & Cut search.
- Stochastic Decomposition: This section contains options specific to the Stochastic Decomposition method bundled within Analytic Solver Comprehensive and Analytic Solver Optimization (Analytic Solver Desktop only) and utilized in the Gurobi Solver Engine. Stochastic Decomposition can be used to solve linear models with recourse variables and uncertainty in the constraints only. See the earlier Stochastic Decomposition section that appears earlier in this guide for more information.

Note that this Solver Engine supports the following Common Solver Options discussed earlier: Assume Non-Negative, Bypass Reports, Max Time and Integer Cutoff. See the previous section for more information.

Each option is listed using the name from the Desktop add-in and the Cloud app in the form: Desktop_Name/Cloud_Name.

Additional Options Available in VBA / SDK

If you want more fine-grained control over the behavior of the Xpress Solver on linear mixed-integer problems and you are using Analytic Solver Desktop or Solver SDK, you can use additional options that aren't available in the Task Pane Engine tab or documented here. Contact Frontline Systems at 775-831-0300 or info@solver.com for more information. To use these options, you should have a good deal of experience solving linear mixed-integer problems. Access the Xpress Optimization Help here. This reference guide documents all available Xpress Solver options.

Xpress General Options

For information related to Assume Non-Negative, Bypass Solver Reports and Max Time, see the Common Options section that appears earlier in this guide.

Algorithm To Use/Algorithm

VBA / SDK: Engine.Params("Algorithm"), value 1-Default, 2-Dual Simplex, 3-Primal Simplex, 4-Newton-Barrier

This option allows you to select the main algorithm or solution method to be used to solve the problem (or each subproblem, in the case of mixed-integer programming problems). In the latest release of Xpress^{MP}, the default solution method is the Dual Simplex method, so the Default and Dual Simplex choices are equivalent. The other possible choices are the Primal Simplex method or the Newton Barrier ("interior point") method.

The Dual Simplex method is usually the best choice for mixed-integer programming problems. The Newton Barrier method may be best if the problem involves a very large number of variables, and the constraints are not very tight.

Assume Quadratic Objective/AssumeQP

VBA / SDK: Engine.Params("AssumeQP"), value True/False

This option is used only if you are solving a quadratic programming problem, and you've either set the Task Pane Platform tab Optimization Model Interpreter to Excel Interpreter, or set the Advanced group Supply Engine With option to Gradients. When you use the PSI Interpreter and supply the XPRESS Solver with Structure or Convexity information, this option is ignored, and the type of objective – linear or quadratic – will be determined automatically.

Otherwise, if the Excel Interpreter is used you must use the QUADPRODUCT function and check this option. Failure to do so will result in the error message "Linearity conditions are not satisfied."

RHS Tolerance/RHSTol

VBA / SDK: Engine.Params("RHSTol"), 0 < value < 1

The value of this option is the "zero tolerance" for constraint right hand sides. If the absolute value of the right hand side of a constraint is less than or equal to this tolerance, it is treated as zero.

Markowitz Tolerance/MarkowitzTol

VBA / SDK: Engine.Params("MarkowitzTol"), 0 < value < 1

The value of this option is the Markowitz tolerance for the elimination phase of the presolve step. It is not used if the Presolve box is not checked.

Matrix Elements Zero Tolerance/MatrixTol

VBA / SDK: Engine.Params("MatrixTol"), 0 < value < 1

The value of this option is the "zero tolerance" for matrix elements. If the absolute value of a matrix element is less than or equal to this tolerance, it is treated as zero.

Scaling Options

VBA / SDK: Engine.Params("Scaling"), -1 – Automatic (Default), 0 – Disabled, 1- Cautious strategy, 2 – Moderate strategy, 3 – Aggressive strategy.

These options determine how the Xpress^{MP} Optimizer will re-scale your model internally. Bear in mind that it is always a good idea to design your model so that all quantities are within a few orders of magnitude of each other, to minimize possible scaling problems.

This option determines whether the Xpress Solver should automatically select between different scaling algorithms.

Automatic – Xpress will automatically select the best scaling algorithm to apply to the model.

Disabled – Select this option to prohibit scaling of any type.

Cautious – Non-standard scaling will only be selected if it appears to be clearly superior.

Moderate – Select this option to apply a moderate scaling strategy to the model.

Aggressive – Standard scaling will only be selected if it appears to be clearly superior.

XPRESS Solver LP Options

Crashing

VBA / SDK: Engine.Params("Crashing"), value 0-Turn Off All Crash Procedures, 1-Singletons Only (One Pass), 2-Singletons Only (Multi Pass), 3-Multiple Passes Using Slacks, 4-Multiple Passes, Slacks at End; also see Slack Passes option below

This option determines the "crashing" method to use when the (Primal) Simplex method begins. (When the Dual Simplex method is used, this option is not applicable.) The "crashing" method attempts to create an initial basis that is as close to feasibility and triangularity as possible, with a goal of reducing the number of Simplex iterations required to find an optimal solution. The possible choices for the "crashing" method are listed in order from least to most aggressive (and time-consuming).

For many problems, Singletons only (one pass or multi pass) may yield an overall improvement in solution time; multiple passes can be relatively expensive compared to the time taken for regular Simplex iterations.

Slack Passes/SlackPasses

VBA / SDK: Engine.Params("Crashing"), set to 11 or higher-Slacks At End, Maximum *N* Passes where *N* = value – 10.

In Excel, this appears as a separate option, but it is used only when the Crashing option is set to Multiple Passes, Slacks at End. It allows you to control the number of passes (hence the time taken) used in the crashing procedure.

Pricing

VBA / SDK: Engine.Params("Pricing"), value 1=Devex Pricing, -1=Partial Pricing, 0-Decide Automatically

This option determines the "pricing" method to use on each (Primal) Simplex iteration. (When the Dual Simplex method is used, this option is not applicable.) The "pricing" method selects a variable to enter the basis. In general, Devex pricing requires more time on each iteration but may reduce the total number of iterations, whereas partial pricing saves time on each iteration, possibly at the expense of more iterations. The Default choice is to determine the pricing method automatically.

Use 'Big M' Method/UseBigM

VBA / SDK: Engine.Params("UseBigM"), value 1=Use 'Big M' Method, 0-Don't Use 'Big M' Method

If this option is set to 1, the "Big M" method is used; if it is set to 0, the traditional Simplex Phase I (to achieve feasibility) and Phase II (to achieve optimality) is used. In the "Big M" method, the objective coefficients of the real variables are taken into account during the "feasibility" phase, possibly leading to an initial feasible basis that is closer to optimal. The tradeoff is some possible roundoff error due to the presence of the "Big M" factor in the problem.

Use Automatic Perturbation/UsePerturb

VBA / SDK: Engine.Params("UsePerturb"), value 1=Use Automatic Perturbation, 0-Don't Use Automatic Perturbation

If this option is set to 1, the problem is perturbed (with slight alterations in the variable bounds) if the Simplex method encounters an excessive number of degenerate pivot steps. This will often enable the Simplex method to escape from degeneracy and make further progress towards an optimal solution.

Iterations

VBA / SDK: Engine.Params("Iterations"), integer value > 0

The value of this option determines the maximum number of iterations (or "pivots") that will be performed by the Simplex method before it stops. For mixed-integer programming problems, this is the maximum number of iterations on each subproblem explored by the Branch & Bound / Branch & Cut method. It plays the same role as the Common Solver Option "Iterations", but it appears separately for the XPRESS Solver. When this option is empty/blank (the default), there is no limit on the number of iterations.

Infeasibility Penalty/BigMPenalty

VBA / SDK: Engine.Params("BigMPenalty"), any numeric value

The value of this option is the infeasibility penalty or "Big M" factor, used if the "Big M" Method is selected. If this option is left empty/blank, the "Big M" factor is determined automatically (which is recommended in most cases).

Perturbation Value/PertubTol

VBA / SDK: Engine.Params("PerturbTol"), any numeric value

The value of this option is the perturbation factor used when the "Use Automatic Perturbation" option is selected. If this option is 0 or blank, the perturbation factor is determined automatically (which is recommended in most cases).

Markowitz Tolerance for Factorization/FactorizationTol

VBA / SDK: Engine.Params("FactorizationTol"), 0 < value < 1

The value of this option is the Markowitz tolerance used in (re)factorization of the basis matrix. The default value is appropriate for most problems.

Invert Frequency/InvertFrequency

VBA / SDK: Engine.Params("InvertFrequency"), integer value >= -1

This option determines the frequency with which the Simplex method basis will be inverted. The basis is maintained in factorized form, and on most Simplex iterations, it is incrementally updated to reflect the pivot step just taken. This is much faster than re-computing the full inverted matrix on each iteration. However, after a number of incremental updates, the basis becomes less well-conditioned or numerically stable, so it becomes desirable or necessary to re-compute the full inverted matrix.

An integer value entered for this option determines the maximum number of Simplex iterations between full inversions of the basis matrix. A value of -1 (the default) indicates that the frequency of inversions should be determined automatically, based on the conditioning of the basis matrix – subject to the minimum number of iterations specified in the edit box below.

Minimum Number of Iterations Between Inverts/InvertMinimum

VBA / SDK: Engine.Params("InvertMinimum"), integer value > 0

If the Invert Frequency is -1, the value of this option determines the minimum number of iterations between full inversions of the basis matrix. The default value of 3 is suitable for most problems; in practice, the number of iterations between full inversions will usually be much larger than this value.

Reduced Cost Tolerance/ReducedTol

VBA / SDK: Engine.Params("ReducedTol"), 0 < value < 1

The value of this option is the zero tolerance for reduced costs. On each iteration, the Simplex method "prices" nonbasic variables, looking for a variable to enter the basis that has a non-zero reduced cost. The candidates are only those variables that have reduced costs with absolute values greater than this tolerance.

Eta Elements Zero Tolerance/ETATol

VBA / SDK: Engine.Params("EtaTol"), 0 < value < 1

The value of this option is the zero tolerance for elements of "eta vectors." On each iteration, the basis inverse is updated via (pre)multiplication by an elementary matrix – which is an identity matrix except for one column, the eta vector. Elements of eta vectors whose absolute values are smaller than this tolerance are treated as zero in this updating step.

Pivot Tolerance/PivotTol

VBA / SDK: Engine.Params("PivotTol"), 0 < value < 1

The value of this option is the zero tolerance for matrix elements chosen for pivoting. On each iteration, the Simplex method looks for a non-zero matrix element in a given column to "pivot" upon. Any matrix element with absolute value less than this tolerance is treated as zero for this purpose, and hence will not be chosen as the pivot element.

Relative Pivot Tolerance/RelPivotTol

VBA / SDK: Engine.Params("RelPivotTol"), 0 < value < 1

The value of this option is the relative zero tolerance for matrix elements chosen for pivoting. On each iteration, the Simplex method looks for a non-zero matrix element in a given column to "pivot" upon. Any matrix element whose absolute value, divided by the absolute value of the largest element in the column, is less than this tolerance is treated as zero for this purpose.

Pricing Candidate List Sizing\PricingCand

VBA / SDK: Engine.Params("PricingCand"), value > 0

The value of this option is the partial pricing candidate list sizing parameter. On each iteration, the Simplex method "prices" some of the nonbasic variables, looking for a variable to enter the basis. The number of variables to be "priced" is multiplied by this parameter. The default value of 1 means that a default-length candidate list is maintained. A value of 2 would mean that a list twice as long as the default would be used; a value of 0.5 would mean that a list half as long would be used; and so on.

XPRESS Solver Presolve Options

Presolve

VBA / SDK: Engine.Params("Presolve"), value -1-Do Not Apply Presolve, 0-Apply Presolve, Do Not Declare Infeasibility, 1-Apply Presolve, 2-Apply Presolve, Keep Redundant Bounds

This option determines the overall use of Presolve methods. The Presolve step attempts to simplify the problem by making simple tests to detect and remove redundant constraints, tighten bounds on variables, and the like. In some cases, the Presolve step may determine that the problem is infeasible, or even determine the optimal solution.

Apply Presolve, Do Not Declare Infeasibility - Selecting this option applies the Presolve step, but proceeds with the Simplex or Barrier method even if Presolve methods determined that the problem is infeasible. In rare cases, due to small numerical differences and the ability of the main Solver algorithms to 'perturb' bounds when seeking a feasible solution, selecting this option may enable you to find a feasible solution.

Do Not Apply Presolve - Selecting this option skips the Presolve step. In this case, none of the Presolve Options settings are used.

Apply Presolve - Selecting this option applies the Presolve step. If the Presolve methods determine that the problem is infeasible, the Solver will return immediately with message "Solver could not find a feasible solution" in the Solver Results dialog.

Apply Presolve, Keep Redundant Bounds - Selecting this option applies the Presolve step, but does not remove redundant bounds (bounds on constraints that are always satisfied, due to bounds on variables or other constraints). In rare case, this may save time in cut generation.

Apply Presolve, Remove Redundant Bounds - Selecting this option applies the Presolve step, removing redundant bounds (bounds on constraints that are always satisfied, due to bounds on variables or other constraints).

Presolve Options

VBA / SDK: Engine.Params("PresolveOptions"), integer value = sum of chosen option values below

This group of options selects individual Presolve methods. In Excel, each method appears as a True/False option (a check box in the legacy Solver Options dialog). 'Columns' refer to decision variables – for example, a singleton column is a variable that appears in only one constraint – and 'Rows' refer to constraints – for example, a singleton row is a constraint involving only one variable. The last four options turn *off* certain methods.

Singleton Column Removal ***[1]***
Singleton Row Removal ***[2]***
Forcing Row Removal ***[4]***
Dual Reductions ***[8]***
Redundant Row Removal ***[16]***
Duplicate Column Removal ***[32]***
Duplicate Row Removal ***[64]***
Strong Dual Reductions ***[128]***
Variable Eliminations ***[256]***
No IP Reductions ***[512]***
No Semi-Continuous Variable Detection ***[1024]***
No Advanced IP Reductions ***[2048]***
Linearly Dependent Row Removal ***[4096]***
No Integer Variable and SOS Detection ***[8192]***

XPRESS Solver Newton-Barrier Options

Cross-Over Control/CrossOver

VBA / SDK: Engine.Params("CrossOver"), value 1=Cross-Over to a Basic Solution, 0-No Cross-Over if Presolve Turned Off

This option determines whether the Newton Barrier method will “cross over” to the Simplex method at the optimal solution, in order to provide an ending basis and sensitivity analysis information. The default option is to perform the cross-over step.

Relative Duality Gap Tolerance/BarrierDualityGapTol

VBA / SDK: Engine.Params("BarrierDualityGapTol"), 0 < value < 1

The value of this option is the “relative duality gap” or convergence tolerance for the Newton Barrier method. When the difference between the primal and dual objective values is less than this tolerance, the Solver determines that the optimal solution has been found. A value of 1.0E-8 to 1.0E-9 is appropriate for most problems.

Maximum Iterations/BarrierIterations

VBA / SDK: Engine.Params("BarrierIterations"), integer value > 0

The value of this option determines the maximum number of iterations performed by the Newton Barrier method. Whereas the Simplex method usually performs a number of iterations proportional to the number of constraints in the problem, the Newton Barrier method tends to find the optimal solution, to a given accuracy, after a number of iterations that is independent of the size of the problem. (However, the time required for *each iteration* of the Newton Barrier method increases with the size of the problem.) The default value is usually more than sufficient, even for very large problems.

Minimal Step Size/BarrierStepSize

VBA / SDK: Engine.Params("BarrierStepSize"), 0 < value < 1

The value of this option is the tolerance for the "step size" termination criterion in the Newton Barrier method. On each iteration of the Newton Barrier method, a step is taken along a computed search direction. If the step size is less than the Minimal Step Size specified here, the Solver will stop and return the current solution. If the Newton Barrier method is making small improvements in the Relative Duality Gap Tolerance on later iterations (see the basic Newton-Barrier tab), it may be better to set this value higher to terminate after a close approximation to the solution has been found.

Cholesky Decomposition Tolerance/CholeskyTol

VBA / SDK: Engine.Params("CholeskyTol"), 0 < value < 1

The value of this option is the zero tolerance for pivot elements in the Cholesky decomposition or factorization of the normal equations coefficient matrix, computed at each iteration of the Newton Barrier method. If the absolute value of the pivot element is less or equal to than this tolerance, it is handled specially in the Cholesky decomposition process.

Primal Infeasibility Tolerance/PrimalFeasTol

VBA / SDK: Engine.Params("PrimalFeasTol"), 0 < value < 1

The value of this option is the tolerance for the "primal infeasibility" termination criterion in the Newton Barrier method. If the relative norm of the difference between the constraints and their bounds in the primal problem falls below this tolerance, the Solver will stop and return the current solution. Note that the Relative Duality Gap Tolerance on the basic Newton-Barrier tab controls the primary termination criterion for the Newton Barrier method.

Dual Infeasibility Tolerance/DualFeasTol

VBA / SDK: Engine.Params("DualFeasTol"), 0 < value < 1

The value of this option is the tolerance for the "dual infeasibility" termination criterion in the Newton Barrier method. If the relative norm of the difference between the constraints and their bounds in the dual problem falls below this tolerance, the Solver will stop and return the current solution. Note that the Relative Duality Gap Tolerance on the basic Newton-Barrier tab controls the primary termination criterion for the Newton Barrier method.

Column Density Factor/ColumnDensity

VBA / SDK: Engine.Params("ColumnDensity"), integer value >= 0

The integer value of this option determines which columns in the normal equations coefficient matrix are considered to be dense. Columns with a number of nonzero elements greater than this value as treated as dense, and are handled specially in the Cholesky factorization of this matrix. A value of 0 (the default) means that the column density factor should be determined automatically.

Max Number Indefinite Iterations/BarrierIndefLimit

VBA / SDK: Engine.Params("BarrierIndefLimit"), integer value > 0

The integer value of this option determines the maximum number of iterations allowed while the Hessian of the Lagrangian is an indefinite matrix. If this limit is exceeded, the Solver will stop with a Solver Result message reporting the problem.

Ordering Algorithm/CholeskyOrder

VBA / SDK: Engine.Params("CholeskyOrder"), value 0-Determine Automatically, 1-Minimum Degree, 2-Minimum Local Fill, 3-Nested Dissection

This option determines the ordering algorithm for the Cholesky factorization of the normal equations coefficient matrix, which is used to preserve the sparsity of the factorized matrix. The default choice is to select the ordering algorithm automatically. The Minimum Degree option selects diagonal elements with the fewest nonzeroes in their rows/columns. The Minimum Local Fill option considers the adjacency graph of nonzeroes in the matrix and seeks to eliminate nodes in a way that minimizes the creation of new edges (i.e. nonzeroes). The Nested Dissection option considers the adjacency graph and recursively seeks to separate it into non-adjacent pieces.

Check Convexity/CheckConvexity

VBA / SDK: Engine.Params("CheckConvexity"), value 0-False or 1-True

If True, quadratic constraints and the objective are first checked for convexity before the solution process begins.

XPRESS Solver Mixed-Integer Options

For information related to Integer Cutoff, see the Common Options section that appears earlier in this guide.

Cut Strategy/CutStrategy

VBA / SDK: Engine.Params("CutStrategy"), value -1=Determine Automatically, 0-No Cuts, 1-Conservative Cut Strategy, 2-Moderate Cut Strategy, 3-Aggressive Cut Strategy

This option determines the strategy used in generating "cuts," which are additional constraints added to subproblems explored by the Branch & Bound / Branch & Cut method that reduce the size of the feasible region without eliminating any potential integer solutions.

The default choice is to determine the cut strategy automatically. You can also specify that no cuts should be generated, or you can select a "conservative cut strategy," a "moderate cut strategy," or an "aggressive cut strategy." If no cuts are generated, some time will be saved in the Branch & Cut process, but more total subproblems will probably have to be explored. A conservative cut strategy specifies fewer opportunities for generating cuts; this may result in a better chance of finding a "good" – though not necessarily optimal – integer solution. An aggressive cut strategy specifies more opportunities for generating cuts; this will often lead to improvement of the best bound in the Branch & Bound process, and will usually require fewer subproblems to be explored to prove optimality; however, generation of the additional cuts and solution of the subproblems will take somewhat more time per subproblem. A “moderate” cut strategy lies between these two alternatives.

Absolute Integer Tolerance/AbsIntTol

VBA / SDK: Engine.Params("AbsIntTol"), any numeric value

The value of this option is the absolute tolerance used to determine whether the Branch & Bound method should continue or stop.

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 objective value of an integer solution can be. If the absolute difference between the incumbent’s objective value and the best bound is less than this tolerance, the Solver will stop with the message “Solver found an integer solution within tolerance” (result code 14).

Relative Integer Tolerance/RelIntTol

VBA / SDK: Engine.Params("RelIntTol"), 0 < value < 1

The value of this option is the relative tolerance used to determine whether the Branch & Bound method should continue or stop.

As described above for the Absolute Integer Tolerance, the Branch & Bound method keeps track of the objective value of the “incumbent” and the “best bound” on the objective found so far. If the absolute difference between the incumbent’s objective value and the best bound, *divided by the best bound,* is less than the Relative Integer Tolerance, the Solver will stop with the message “Solver found an integer solution within tolerance” (result code 14).

Maximum Number of Nodes/MaxSubProblems

VBA / SDK: Parameter Name "MaxSubProblems", integer value > 0

This option has an effect only if you have integer (or binary or alldifferent) constraints in your model. Use this option to place 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 value of this option is blank by default, meaning there is no limit on the number of subproblems.

Maximum Number of Solutions/MaxFeasibileSols

VBA / SDK: Parameter Name "MaxFeasibleSols", integer value > 0

The value in the Max Number of Solutions edit box places a limit on the number of feasible solutions found by the Xpress engine 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.

Amount to Add to Solution to Obtain New Cutoff/AbsAddCut

VBA / SDK: Engine.Params("AbsAddCut"), any nonzero numeric value

You can use the “Amount to Add” and “Percentage to Add” options to further speed up the solution of mixed-integer problems. When the Branch & Bound method finds a feasible integer solution with a given objective, you may not be interested in searching for additional integer solutions unless their objectives are “substantially” better than the objective of this known solution. If you specify a nonzero value for this option, it will be added to the objective of the known integer solution to form a new “cutoff” value, similar to the Integer Cutoff option described earlier. This process will be repeated each time an improved integer solution is found. The effect is to cut off the search in portions of the Branch & Bound tree whose best possible objective would not be “substantially” better than the current solution – thereby saving time in the search.

Percent to Add to Solution to Obtain New Cutoff/RelAddCut

VBA / SDK: Engine.Params("RelAddCut"), 0 < value < 1

This option works in conjunction with the Amount to Add to Solution to Obtain New Cutoff (above) to further speed up the search for integer solutions. If you specify a value (between zero and 100) for this option, then whenever the Branch & Bound method finds an improved integer solution, the objective of this solution will be multiplied by .01 times this value, to form a new candidate “Amount to Add” value. The new Amount to Add to Solution to Obtain New Cutoff is then set to the maximum of its current value and this computed value. The effect is to cut off the search in portions of the Branch & Bound tree whose best possible objective would not be “x% better” than the current solution – saving time in the search.

Number of Threads/MipThreads

VBA / SDK: Engine.Params("MipThreads"), integer value > 0

This option allows you to specify the number of parallel threads of execution to be used in solving mixed-integer problems. Each thread normally runs on a separate processor and solves subproblems in the Branch & Bound tree, under the control of a ‘master’ program.

The default value of 0 means that all of your computer’s processors will be used. If you have other applications running on the same computer, and you wish to *reserve* one or more processors for these other applications, you can use this option to limit the number of threads used by the XPRESS Solver. Note that Windows multi-tasking will automatically share processors among applications; use this option only if you must ensure that your other applications run more quickly while the XPRESS Solver is running.

Integer Feasibility Tolerance/IntegerFeasTol

VBA / SDK: Engine.Params("IntegerFeasTol"), 0 < value < 1

The value of this option is the tolerance within which a decision variable’s value (at the optimal solution to a subproblem) is considered to be integer, for purposes of the Branch & Bound method. If the absolute value of the difference between the variable's value and the nearest integer is less than this tolerance, the variable is treated as having that exact integer value.

Cut Frequency/CutFrequency

VBA / SDK: Engine.Params("CutFrequency"), integer value >= -1

The value of this option determines the frequency with which cuts will be generated in the Branch & Bound tree. "Cuts" are additional constraints added to subproblems explored by the Branch & Bound / Branch & Cut method that reduce the size of the feasible region without eliminating any potential integer solutions.

The Cut Frequency value is applied to the depth of the subproblem or node in the Branch & Bound tree. For example, if the value is 5, new cuts are generated at nodes that occur at every 5th "level" in the tree. The default value is –1, which means that the Cut Frequency should be determined automatically.

Default Pseudo Cost/PseudoCost

VBA / SDK: Engine.Params("PseudoCost"), value > 0

The value of this option is the default pseudo cost used in computing the estimated degradation associated with an unexplored subproblem or node in the Branch & Bound tree. A "pseudo cost" is associated with each integer decision variable and is an estimate of the amount by which the objective will worsen if that variable is forced to an integer value. This default value is used initially, but pseudo costs for each variable are updated as new subproblems are explored.

Maximum Depth Cut Generation/CutDepth

VBA / SDK: Engine.Params("CutDepth"), integer value >= -1

The value of this option determines the maximum depth in the Branch & Bound tree at which cuts will be generated. "Cuts" are additional constraints added to subproblems explored by the Branch & Bound / Branch & Cut method that reduce the size of the feasible region without eliminating any potential integer solutions.

Generating cuts can take significant time, and cuts are usually less important at deeper levels in the Branch & Bound tree, because earlier cuts or tighter bounds imposed on the variables have already reduced the feasible region. A value of 0 for this option means that cuts will be generated only at the root node – they will not be generated in the tree at all. The default value is –1, which means that the Maximum Depth for Cut Generation should be determined automatically.

Integer Preprocessing/IntPreProcessing

VBA / SDK: Engine.Params("IntPreProcessing"), value -1-Determined by Matrix Characteristics, or sum of the values for each option below

This option controls the types of "preprocessing" that will be performed at each node or subproblem, before solving the subproblem with the Simplex method. The various preprocessing methods each take some time, but they can simplify the subproblem before it is solved and may even determine optimality or infeasibility of the subproblem without the need to use the Simplex method. The default choice is to select the Preprocessing Options automatically based on the characteristics of the LP coefficient matrix at the node.

No Integer Preprocessing (0) - If this option is chosen, the method for reduced cost fixing is determined automatically, based on the characteristics of the LP coefficient matrix.

Reduced Cost Fixing at Each Node (1) - If this option is chosen, reduced cost fixing will be performed at each node in the Branch & Bound tree. Reduced cost fixing can tighten bounds on the variables at the current node, based on the reduced costs computed for the variables at a "parent" node in the tree, and the objective of the current incumbent.

Logical Preprocessing at Each Node (2) - If this option is chosen, logical preprocessing will be performed at each node in the Branch & Bound tree. Logical preprocessing is performed on 0-1 or binary integer variables, and often results in fixing the values of many 0-1 variables based on the fixed values of other 0-1 variables occurring in the same constraints. This simplifies the problem before it is solved, and may even determine optimality or infeasibility of the subproblem before the Simplex method is started.

Probing at the Top Node (4) - If this option is chosen, probing of 0-1 or binary integer variables is performed at the top or root node. Probing sets certain 0-1 variables to either 0 or 1 and then deduces implications for other 0-1 variables occurring in the same constraints. Alone or in combination with other strategies, it can greatly reduce the number of nodes that need to be explored.

Reduced Cost Fixing and Logical Preprocessing – This option combines Reduced Cost Fixing and Logical Preprocessing at each node.

Reduced cost Fixing and Probing at the Top Node - This option combines Reduced Cost Fixing at each node and Probing at the top node.

Logical Preprocessing and Probing – This option combines Logical Preprocessing at each node and Probing at the top node.

Reduced Cost Fixing, Logical Preprocessing, and Probing – This option combined Reduced Cost Fixing and Logical Preproessing at each node and Probing at the top node.

Strong Branching Global Entities/StrongBranchGlobal

VBA / SDK: Engine.Params("StrongBranchGlobal"), integer value >= -1

This option determines the number of infeasible global entities on which the Solver will perform strong branching. The default value of –1 means that the number should be determined automatically.

"Strong branching" is a method where several iterations of the dual Simplex method are carried out at a node, without solving the subproblem to optimality, to determine more accurate pseudo costs for the variables. The pseudo costs are later used to guide the selection of nodes to be explored and variables to be branched upon.

Strong Branching Dual Iterations/StrongBranchDual

VBA / SDK: Engine.Params("StrongBranchDual"), integer value >= -1

This option determines the number of dual Simplex iterations to carry out when performing strong branching. The default value of –1 means that the number should be determined automatically.

Number of Lifted Cover Inequalities at the Top/CoverCutsTop

VBA / SDK: Engine.Params("CoverCutsTop"), integer value >= -1

The value of this option determines the number of times that lifted cover inequalities are generated at the top or root node of the Branch & Bound tree. The default value of –1 means that the number of times should be determined automatically.

A lifted cover inequality is a type of cut (additional constraint) that can be particularly effective at reducing the size of the feasible region without eliminating any potential integer solutions. Multiple "passes" can be made, generating new lifted cover inequalities on each pass, and further reducing the feasible region; however, this can take a significant amount of time (compared to solving the subproblem).

Separate options for Lifted Cover Inequalities and Gomory Cuts are provided to control the number of passes at the top or root node, and at nodes deeper in the tree. There is usually a greater payoff to cut generation at the top node, since these cuts can be applied to every subproblem explored deeper in the Branch & Bound tree. If these options are set manually, a typical value might be 20 for the number of passes at the top node, and 1 or 2 for the number of passes at nodes deeper in the tree.

Number of Gomory Cut Passes at the Top/GomoryCutsTop

VBA / SDK: Engine.Params("GomoryCutsTop"), integer value >= -1

The value of this option determines the number of times that Gomory cuts are generated at the top or root node of the Branch & Bound tree. The default value of –1 means that the number of times should be determined automatically.

Gomory cuts can always be generated if the current subproblem does not yield an all-integer solution; however Gomory cuts are usually not as effective as lifted cover inequalities in reducing the size of the feasible region.

Number of Lifted Cover Inequalities in the Tree/CoverCutsTree

VBA / SDK: Engine.Params("CoverCutsTree"), integer value >= 0

The value of this option determines the number of times that lifted cover inequalities are generated at nodes other than the top or root node of the Branch & Bound tree. As noted above for the option Lifted Cover Inequalities at the Top Node, the payoff from generating these cuts is greater at the top node, since they apply to all nodes in the Branch & Bound tree, whereas cuts generated at an arbitrary node apply only to that subproblem and its descendants (if any). This is reflected in the default value 1.

Number of Gomory Cut Passes in the Tree/GomoryCutsTree

VBA / SDK: Engine.Params("GomoryCutsTree"), integer value >= 0

The value of this option determines the number of times that Gomory cuts are generated at nodes other than the top or root node of the Branch & Bound tree. As discussed above, the payoff from generating these cuts is greater at the top node, since they apply to all nodes in the Branch & Bound tree, whereas cuts generated at an arbitrary node apply only to that subproblem and its descendants (if any). This is reflected in the default value 1.

XPRESS Solver Node Selection Options

Control

VBA / SDK: Engine.Params("Control"), value 0-Choice Dependent on the Matrix Characteristics, 1-Local First, 2-Best First, 3-Local Depth First, 4-Best First for the first *N* nodes, then Local First, 5-Pure Depth First; for *N* see “Number of Nodes for Best First” below

This option determines the set of possible subproblems, or “nodes” in the Branch & Bound tree, that will next be considered for solution each time a given subproblem has been solved. (Additional options below determine which specific subproblem is selected from the set of possible subproblems chosen here.)

The default choice is to determine the set of possible nodes automatically, based on the characteristics of the LP coefficient matrix.

The Local First choice first considers only the immediate descendants and the siblings of the subproblem just solved. If these nodes have already been evaluated, then all other unexplored nodes in the tree are considered.

The Best First choice treats all unexplored nodes equally. This option results in a classical “best-first” or “breadth-first” search.

The Local Depth First choice first considers only the immediate descendants and the siblings of the subproblem just solved. If these nodes have already been evaluated, then the deepest nodes in the Branch & Bound tree are considered.

The Best First, then Local First choice acts like the Best First choice for the first *N* subproblems explored, then it acts like the Local First choice. This is often an effective compromise, because it provides some breadth of coverage of the possible integer solutions, but explores early candidate solutions more deeply than a pure breadth-first search. The Number of Nodes for Best First option determines the number of subproblems *N*.

The Pure Depth First choice considers the deepest nodes in the Branch & Bound tree. This option results in a classical “depth-first” search.

Number of Nodes for Best First/BreadthFirst

VBA / SDK: Engine.Params("BreadthFirst"), integer value > 0

The value of this option is the number of nodes *N* for which the Best First strategy will be used, before switching to the Local First strategy. The default value is 10.

Node Selection Criterion/NodeCriterion

VBA / SDK: Engine.Params("NodeCriterion"), value 1-Choose Based on Forrest-Hirst-Tomlin Criterion, 2-Always Choose Node with Best Estimated Objective, 3-Always Choose Node with Best Bound on Objective

This option determines how the next subproblem or node in the Branch & Bound tree is selected for further processing (which may include steps such as integer preprocessing, cut generation, and solution of the LP relaxation). The *group* of subproblems to be evaluated by the Node Selection Criterion is determined by the Control option described earlier.

The default (third) choice is to select the node with the best *bound* on the objective value (of an integer solution that may be found by exploring that node). The second choice is to select the node with the best *estimated* objective value, where the estimate is computed from pseudo costs associated with the integer variables. The first choice is to select the node based on the Forrest-Hirst-Tomlin criterion, which takes into account the best known integer solution and seeks a node with a large potential improvement over that solution, relative to the estimated degradation in the objective expected in an all-integer solution that may be found by exploring this node.

Integer Variable Estimates/VarSelection

VBA / SDK: Engine.Params("VarSelection"), value -1-Determine Automatically, 1-Minimum of the 'Up' or 'Down' Pseudo Costs, 2-'Up' Pseudo Cost Plus the 'Down' Pseudo Cost, 3-Maximum of the 'Up' or 'Down' Pseudo Costs, plus Twice the Minimum of the 'Up' or 'Down' Pseudo Costs, 4- Maximum of the 'Up' or 'Down' Pseudo Costs, 5-'Down' Pseudo Cost, 6-'Up' Pseudo Cost

This option selects one of six different ways to combine the pseudo costs associated with the integer variables into an overall estimated degradation in the objective expected in an all-integer solution that may be found by exploring this node. It is relevant only if the Forrest-Hirst-Tomlin criterion is chosen for the Node Selection Criterion. By default, one of the six ways to combine the pseudo costs is chosen automatically. The choices are as follows:

1. Sum the minimum of the "up" and "down" pseudocosts
2. Sum all of the "up" and "down" pseudocosts
3. Sum the maximum, plus twice the minimum of the "up" and "down" pseudocosts. (This has been found to be effective in empirical studies.)
4. Sum the maximum of the "up" and "down" pseudocosts
5. Sum the "down" pseudocosts
6. Sum the "up" pseudocosts

XPRESS Solver Heuristics Options

Strategy

64 bit: VBA / SDK: Engine.Params("HeurStrategy"), value -1-Automatic Selection (Default), 0-No Heuristics, 1-Focus on reducing primal-dual gap, 2-Aggressive Heuristics

32 bit: VBA / SDK: Engine.Params("HeurStrategy"), value -1-Automatic Selection (Default), 0-No Heuristics, 1-Basic Heuristics, 2-Enhanced Heuristics, 3-Extensive Heuristics

This option determines how extensively the 32 bit or 64 bit XPRESS Solver will employ *heuristic methods* in an effort to quickly find an integer solution. If one is found, this integer solution becomes an "incumbent" that allows the XPRESS Solver to bound the search at other tree nodes.

In 64 bit, by default, the XPRESS Solver determines automatically when to apply heuristic methods. On some problems, you may get better results by setting the Strategy option to Focus on reducing primal-dual gap or Aggressive Heuristics. Each of these options takes progressively more time, which may pay off if good integer solutions are found; however, on a given problem, each level of heuristics may or may not succeed in identifying new integer solutions. Setting this option to No Heuristics will save a small amount of time.

In 32 bit, by default, the XPRESS Solver determines automatically when to apply heuristic methods. On some problems, you may get better results by setting the Strategy option to Basic Heuristics, Enhanced Heuristics, or Extensive Heuristics. Each of these options takes progressively more time, which may pay off (sometimes handsomely) if good integer solutions are found; however, on a given problem, each level of heuristics may or may not succeed in identifying new integer solutions. Setting this option to No Heuristics will save a small amount of time.

Frequency

VBA / SDK: Engine.Params("HeurFreq"), integer value >= -1

This option determines the frequency with which heuristics are used. Set this option to a smaller value to increase the frequency of heuristics use, or to a larger value to decrease the frequency. -1 means that the frequency will be determined automatically.

Programming the Solver Engines

Introduction

Whether you are working in Desktop Excel, or outside Excel with Solver SDK, you can write a program that will define or load an optimization problem, select a Solver engine, set options and parameters, solve the problem, and retrieve solution and report information. In Analytic Solver *Desktop* (Comprehensive or Optimization) you can use VBA (Visual Basic Applications Edition) in Excel to do this, or you can use VB.NET or C# with Microsoft's Visual Studio Tools for Office (VSTO). In Solver SDK, you can use Visual Basic, VB.NET, C#, C/C++, Java, MATLAB, or another programming language to perform the same steps.

If you have an optimization model in Desktop Excel, controlling the Solver can be as simple as *adding one or two lines* 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, all you need to do in your code is activate the worksheet and call the traditional VBA function SolverSolve, or create a problem (Dim prob As New Problem) and call the method prob.Solver.Optimize.

Note: This functionality is not supported in Analytic Solver Cloud.

Traditional VBA Functions and Object-Oriented API

In Microsoft Excel, you can use either of two APIs (Application Programming Interfaces) to control the Solver:

- The traditional VBA functions, such as **SolverOK** and **SolverSolve**, correspond to operations you can perform interactively in the Solver dialogs. You can perform a series of interactive steps and automatically record a macro that uses these functions. But more work is required if you want to use the solution values or report information in your application program.
- The new object-oriented API provides objects that correspond to the **Problem**, **Model**, **Solver**, **Engine**, **Variables**, and **Functions**. IntelliSense prompts make it much easier to write code using them, and it's much easier to use solution values and report information in your application program.

The object-oriented API is also highly compatible with the object-oriented API offered by Solver SDK, Frontline's powerful tool for building applications based on optimization and simulation in a programming language. If you're planning to move your application outside of Excel and into a "standalone program," the object-oriented API is highly recommended.

You have several other sources for information on programming Analytic Solver Comprehensive, Analytic Solver Optimization and Solver SDK:

- For examples of how to use the object-oriented API in Excel, please consult the chapter "Automating Optimization in VBA" in the Analytic Solver User Guide.
- For more information on the traditional VBA functions and the object-oriented API properties and methods available to control the Solver, please consult the chapter "VBA Object Model Reference" in the Analytic Solver Reference Guide.
- For information on the object-oriented API and procedural API offered by Solver SDK, please consult the Solver SDK User Guide.
- For details on how to set and get option and parameter values for the Solver Engines described in this Guide, please see the earlier chapter "Solver Engine Options."

This chapter describes only the additional "traditional" VBA functions available in Microsoft Excel to set and get options and parameters for the Large-Scale LP/QP Solver, Large-Scale GRG Solver, Large-Scale SQP Solver, Knitro Solver, MOSEK Solver, XPRESS Solver, and OptQuest Solver. In almost all cases, you'll use exactly the same *names* for options and parameters (such as "MaxTime" and "Iterations") in the traditional VBA functions, the new object-oriented API, and in Solver SDK.

Using the Traditional VBA Functions in Excel

In Microsoft Excel, you normally use the Solver interactively via the Ribbon, Task Pane, and dialog boxes. But you can control every aspect of the Solver's operation by calling the traditional VBA functions such as SolverSolve. You can display or hide the Solver dialogs, create or modify choices of objective, variables and constraints, check whether an optimal solution was found, and produce reports.

The traditional VBA function descriptions below refer to tabs in the traditional Solver Options dialogs, which you can display by selecting the Add-Ins tab, clicking Premium Solver, selecting your Solver Engine in the Solver Parameters dialog, and clicking the Options button.

Referencing the Traditional VBA Functions

To use the traditional VBA functions, your Visual Basic module must include a reference to the Solver add-in. In addition, to use the VBA functions described in this Guide, your Visual Basic module must include a reference to the appropriate Solver engine add-in (LSLPeng, LSGRGeng, LSSQPeng, Knitroeng, MOSEKeng, XPRESSeng or OPTQeng).

In Microsoft Excel, press Alt-F11 to open the Visual Basic Editor, choose Tools References... and make sure that the boxes next to **Solver** *and* the appropriate Solver engine add-in are checked. If you don't see these choices in the list of available references, click on the Browse... button, select Files of Type "Microsoft Excel Files" if necessary, navigate to the appropriate subdirectory (typically C:\Program Files\Frontline Systems\Engines), then select Solver.xla and LSLPeng.xla, LSGRGeng.xla, LSSQPeng.xla, Knitroeng.xla, MOSEKeng.xla, XPRESSeng.xla or OPTQeng.xla, and click OK.

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 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 standard return values range from -1 to 29; in addition, the Interval Global Solver (included in Analytic Solver Comprehensive and Analytic Solver Optimization) and the Large-Scale GRG Solver, Large-Scale SQP Solver, Knitro Solver, MOSEK Solver and OptQuest Solver can each return Solver Engine-specific values. These integer values are summarized in the description of the SolverSolve function below, but their meanings are described more fully in the chapter "Solver Result Messages." These return values are the same as the OptimizeStatus values returned when you solve a problem using the object-oriented API in VBA or Solver SDK.

One group of functions can return a variety of numeric, logical, string or array values, depending on the arguments you supply. These functions (SolverLSLPGet, SolverLSGRGGet, SolverLSSQPGet, SolverKnitroGet, SolverMOSEKGet, SolverXPRESSGet, or SolverOPTQGet), which follow the style of the SolverGet function found in the standard Excel Solver, may be used to get or "read" the option or parameter values for the current Solver model, on the active sheet or any other worksheet you choose.

Using the Object-Oriented API in Excel

You can also control every aspect of the Solver's operation by using the object-oriented API exposed by Analytic Solver Comprehensive and Analytic Solver Optimization. To use this API, you simply set properties and call methods on objects that represent the optimization Problem, Model, Solver, Engine, Variables, and Functions. The object-oriented API is both easier to use and more productive if you want to access solution values, dual values, and other report information in your program code. This API is also very similar to the object-oriented API offered by Solver SDK

Referencing the Object-Oriented API

To use the object-oriented API described in the Analytic Solver Reference Guide, your Visual Basic module must include a reference to the Analytic Solver COM server. Once you've done this, you can use any of the plug-in Solver engines, without requiring a separate reference to their Excel add-ins.

In Microsoft Excel, press Alt-F11 to open the Visual Basic Editor, choose Tools References... and make sure that the box next to **Analytic Solver 2018 Type Library** is checked. Note that this is a **different** reference from **Solver**, which is the reference you add in order to use the "traditional" VBA functions. The Analytic Solver's COM server should always appear in the list of choices.

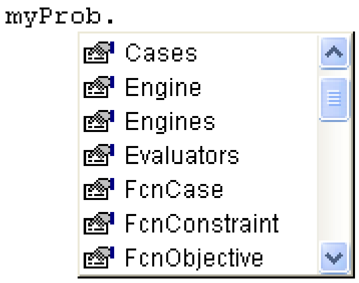
Using IntelliSense Prompting

Since the VBA Editor recognizes the object model exposed by Analytic Solver Comprehensive and Analytic Solver Optimization– 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 of code such as:

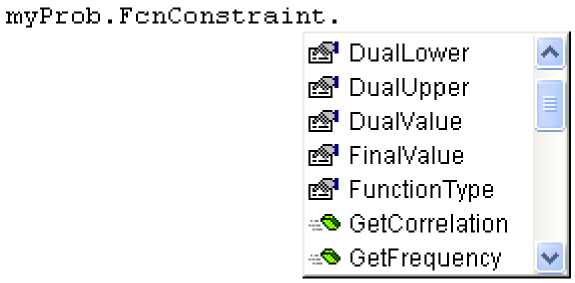
```
Dim myProb as New Problem
```

then start a new line with `myProb.`, you'll be prompted with the properties and methods available for Problems:



```
myProb.
```

If you select FcnConstraint and then type a period, you'll be prompted with the properties and methods available for Functions:



```
myProb.FcnConstraint.
```

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 Solver from new languages such as **VB.NET** and **C#**, working in Microsoft Visual Studio, and receive IntelliSense prompts for these languages.

Accessing Object Properties

With the object-oriented API, a single line of VBA code such as:

```
Dim myProb as New Problem
```

creates a Problem object and initializes it with the Solver problem defined on the active worksheet. The Problem object has associated Model, Solver, Engine, Variable and Function objects that are created automatically. You can use these objects by getting or setting **property** values, using the same syntax as assignment statements in VBA and other languages. See the chapter "Solver Engine Options" for the string names of options and parameters for all of the Solver engines.

For example, you can set the Max Time limit for the currently selected Solver Engine to 1000 seconds by writing:

```
myProb.Engine.Params("MaxTime").Value = 1000
```

and you can get the current Max Time limit in a variable myTime (declared with Dim myTime As Double) by writing:

```
myTime = myProb.Engine.Params("MaxTime").Value
```

In Solver SDK – discussed below – you can use the same property assignment statements (adapted slightly for programming language syntax) by writing statements like these:

VBA / 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);`

Handling Exceptions

Where the traditional VBA functions require that you check the return value of each function call for possible errors, the object-oriented API for Analytic Solver Comprehensive and Analytic Solver Optimization raises **exceptions** for all error conditions. You can write one block of code to handle error conditions, and this code will be executed automatically if an error occurs. The simplest way to handle errors in VBA is to write a line such as:

```
On Error Goto myHandler
```

In the code following myHandler, you can access the VBA Err object for specific information about the exception that was raised. For example:

```
myHandler: MsgBox Err.Description
```

Other languages such as VB.NET and C# have even more convenient and powerful exception handling features: You'll typically use a "try-catch" block that encloses the lines of code where errors may arise, and the lines of code to handle errors.

Using the Object-Oriented API in Custom Applications

In a custom application program that you write using Solver SDK, you always control the Solver's operation programmatically. You can either build your model as a Microsoft Excel workbook, and tell Solver SDK to load that workbook at runtime, or you can build your model as a program in your chosen language. In the latter case, you define a program function that will compute values for your objective function and constraints, given values for the decision variables, and you tell the SDK that this function is your Evaluator of Eval_Type_Function. Optionally – to improve performance – you can define a function that will compute *derivatives* or *gradients* for your objective function and constraints, at the point represented by the current values of the decision variables, and you tell the SDK that this function is your Evaluator of Eval_Type_Gradient.

Linear and Quadratic Problems

If you are solving a linear programming (LP) or LP/MIP problem, you don't have to write an Evaluator function. If you know the LP coefficients (i.e. the constant gradient elements for the objective and constraints), you can pass them directly to the SDK by setting the Model object AllLinear property. With this information, the SDK can compute values for your objective and constraint functions itself.

Similarly, if you are solving a quadratic programming (QP) or QP/MIP problem and you know the QP coefficients (i.e. the constant elements of the Hessian matrix for the quadratic objective), you can pass them directly to the SDK by setting the Model object FcnQuadratic property. This allows the SDK to compute values for your quadratic objective itself. It's your choice: You can supply an Evaluator function and let the SDK determine the linear and/or quadratic coefficients, or you can supply the coefficients and let the SDK determine the Evaluator function.

Selecting Solver Engines

You select a Solver Engine in Solver SDK just as easily as you do in Analytic Solver Comprehensive and Analytic Solver Optimization. For example, in C++ you'd write:

```
myProb.Engine = myProb.Engines["GRG Nonlinear"];
```

to select the GRG Nonlinear Solver. To use a plug-in Solver Engine with the Solver SDK, you simply add it to the collection of available Solver Engines with:

```
CProblem myProb;
CEngine myEngine ("Knitro Solver", "Knitroeng.dll");
myProb.Engines.Add (myEngine);
```

and then select this Solver and use it to solve a problem with:

```
myProb.Engine = myProb.Engines["Knitro Solver"];
myProb.Solver.Optimize();
```

You don't have to change any other part of your program, unless you wish to take advantage of Solver engine-specific options or parameters. And you can use the *same* plug-in Solver Engine with Analytic Solver Comprehensive and Analytic Solver Optimization in Excel, and with Solver SDK outside Excel.

Large-Scale GRG Solver Functions

The Large-Scale GRG Solver provides the following traditional VBA functions to either set (with SolverLSGRGOptions) or get (with SolverLSGRGGet) the current option and parameter settings that affect the performance of this Solver engine.

All of these options and parameters may also be set via the object-oriented API, as described above under "Accessing Object Properties." Engine.Param names are listed for each option in the chapter "Solver Options;" these names normally match the named arguments listed below for SolverLSGRGOptions.

In addition, you can use the SolverIntOptions and SolverIntGet functions, described below (giving only the supported arguments), to set or get the options and parameters that appear on the Engine task pane Integer tab of the dialog box for the Large-Scale GRG Solver.

SolverLSGRGGet

Returns LSGRG Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the LSGRG Solver is selected in the Solver Engines dropdown list.

VBA Syntax

SolverLSGRGGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

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 (as a decimal number)
6	The Random Seed (as a decimal number)
7	TRUE if Show Iteration Result check box is selected; FALSE otherwise
8	TRUE if Use Automatic Scaling check box is selected; FALSE otherwise
9	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
10	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise.
11	TRUE if the Recognize Linear Variables check box is selected; FALSE otherwise.
12	TRUE if the Relax Bounds on Variables check box is selected; FALSE otherwise.
13	TRUE if Multistart Search is selected; FALSE otherwise
14	TRUE if Topographic Search is selected; FALSE otherwise
15	TRUE if Require Bounds on Vars 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

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.

SolverLSGRGOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the Large-Scale GRG Solver is selected in the Solver Engines dropdown list. Specifies options for the Large-Scale GRG Solver.

VBA Syntax

SolverLSGRGOptions (MaxTime:=, Iterations:=, Precision:=, Convergence:=, PopulationSize:=, RandomSeed:=, StepThru:=, Scaling:=, AssumeNonneg:=, BypassReports:=, RecognizeLinear:=, RelaxBounds:=, 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 argument, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

Scaling (or **ScalingOption**, for backward compatibility) 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, Solver will skip preparing the information needed to create Solver Reports. If FALSE, 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.

RelaxBounds is a logical value corresponding to the Relax Bounds on Variables check box. If TRUE, the Solver may recalculate the model with values for the variables that are slightly outside their bounds, in order to speed the solution process. If FALSE, the Solver will use values for the variables that are always within the lower and upper bounds you specify.

MultiStart is a logical value corresponding to the Multistart Search check box. If TRUE, the Solver will use Multistart Search, in conjunction with the LSGRG Solver, to seek a globally optimal solution. If FALSE, the LSGRG 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.

Large-Scale SQP Solver Functions

The Large-Scale SQP Solver provides the following traditional VBA functions to either set (with SolverLSSQPOptions) or get (with SolverLSSQPGet) the current option and parameter settings that affect the performance of this Solver engine.

All of these options and parameters may also be set via the object-oriented API, as described above under "Accessing Object Properties." Engine.Param names are listed for each option in the chapter "Solver Options;" these names normally match the named arguments listed below for SolverLSSQPOptions.

In addition, you can use the SolverIntOptions and SolverIntGet functions, described below (giving only the supported arguments), to set or get the options and parameters that appear in the Integer tab of the dialog box for the Large-Scale SQP Solver.

SolverLSSQPGet

Returns LSSQP Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the LSSQP Solver is selected in the Solver Engines dropdown list.

VBA Syntax

SolverLSSQPGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

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 (as a decimal number)
6	The Random Seed (as a decimal number)
7	TRUE if Show Iteration Result check box is selected; FALSE otherwise

8	TRUE if Use Automatic Scaling check box is selected; FALSE otherwise
9	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
10	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise.
11	TRUE if the Treat Constraints as Linear check box is selected; FALSE otherwise.
12	TRUE if the Treat Objective as Linear check box is selected; FALSE otherwise.
13	TRUE if Multistart Search is selected; FALSE otherwise
14	TRUE if Topographic Search is selected; FALSE otherwise
15	TRUE if Require Bounds on Vars is selected; FALSE otherwise
16	A number corresponding to the type of Derivatives: 1 = Forward 2 = Central
17	The Mutation Rate (as a decimal number)
18	A number corresponding to the Local Search option: 1 for Randomized Local Search, 2 for Deterministic Pattern, 3 for Gradient Local, or 4 for Automatic Choice
19	TRUE if Fix Nonsmooth Variables 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.

SolverLSSQPOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the Large-Scale SQP Solver is selected in the Solver Engines dropdown list. Specifies options for the Large-Scale SQP Solver.

VBA Syntax

SolverLSSQPOptions (MaxTime:=, Iterations:=, Precision:=, Convergence:=, PopulationSize:=, RandomSeed:=, StepThru:=, Scaling:=, AssumeNonneg:=, BypassReports:=, LinearConstraints:=, LinearObjective:=, MultiStart:=, TopoSearch:=, RequireBounds:=, Derivatives:=, MutationRate:=, 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.

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 argument, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

Scaling (or **ScalingOption**, for backward compatibility) 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, Solver will skip preparing the information needed to create Solver Reports. If FALSE, Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

LinearConstraints is a logical value corresponding to the Treat Constraints as Linear check box. If TRUE, the Solver will assume that all constraints are linear functions, and will compute their gradients only once at the beginning of the solution process. If FALSE, the Solver will treat the constraints as nonlinear, and will compute their gradients at each major iteration. See the chapter “Solver Options” for a further discussion of this option.

LinearObjective is a logical value corresponding to the Treat Objective as Linear check box. If TRUE, the Solver will assume that the objective is a linear function, and will compute its gradient only once at the beginning of the solution process. If FALSE, the Solver will treat the objective as nonlinear, and will compute its gradient at each major iteration. 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 LSSQP Solver, to seek a globally optimal solution. If FALSE, the LSSQP 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, and if Multistart Search is selected, 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.

Derivatives is the number 1 or 2 and corresponds to the Derivatives option: 1 for Forward and 2 for Central.

MutationRate must be a number between zero and one, but not equal to zero or one. It corresponds to the Mutation Rate edit box.

LocalSearch is a number corresponding to the option button selected in the Local Search option group:

LocalSearch	**Local Search Strategy**
1	Randomized Local Search
2	Deterministic Pattern
3	Gradient Local
4	Automatic Choice

FixNonSmooth is a logical value corresponding to the Fix Nonsmooth Variables check box. If TRUE, the Solver will fix the nonsmooth 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.

Knitro Solver Functions

The Knitro Solver provides the following traditional VBA functions to either set (with SolverKnitroOptions) or get (with SolverKnitroGet) the current option and parameter settings that affect the performance of this Solver engine.

All of these options and parameters may also be set via the object-oriented API, as described above under “Accessing Object Properties.” Engine.Param names are listed for each option in the chapter “Solver Options;” these names normally match the named arguments listed below for SolverKnitroOptions.

In addition, you can use the SolverIntOptions and SolverIntGet functions, described below (giving only the supported arguments), to set or get the options and parameters that appear in the Integer tab of the dialog box for the Knitro Solver.

SolverKnitroGet

Returns Knitro Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the Knitro Solver is selected in the Solver Engines dropdown list.

VBA Syntax

SolverKnitroGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

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 (as a decimal number)
6	The Random Seed (as a decimal number)
7	TRUE if Show Iteration Result check box is selected; FALSE otherwise
8	TRUE if Use Automatic Scaling check box is selected; FALSE otherwise

9	`TRUE if the Assume Non-Negative check box is selected; FALSE otherwise`
10	`TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise.`
11	`TRUE if the Treat Constraints as Linear check box is selected; FALSE otherwise.`
12	`TRUE if the Treat Objective as Linear check box is selected; FALSE otherwise.`
13	`TRUE if the Relax Bounds on Variables check box is selected; FALSE otherwise.`

14	`A number corresponding to the type of Solution Method:`
	`1 = Select Automatically`
	`2 = Interior Point Direct`
	`3 = Interior Point CG`
	`4 = Active Set`

15	`TRUE if Multistart Search is selected; FALSE otherwise`
16	`TRUE if Topographic Search is selected; FALSE otherwise`
17	`TRUE if Require Bounds on Vars is selected; FALSE otherwise`

18	`A number corresponding to the type of Derivatives:`
	`1 = Forward`
	`2 = Central`

19	`A number corresponding to the type of Second Derivatives:`
	`1 = Analytic 2nd Derivatives`
	`2 = Analytic 1st Derivatives`
	`3 = Finite Differences`

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.

SolverKnitroOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the Knitro Solver is selected in the Solver Engines dropdown list. Specifies options for the Knitro Solver.

VBA Syntax

SolverKnitroOptions (MaxTime:=, Iterations:=, Precision:=, Convergence:=, PopulationSize:=, RandomSeed:=, StepThru:=, Scaling:=, AssumeNonneg:=, BypassReports:=, LinearConstraints:=, LinearObjective:=, RelaxBounds:=, SolutionMethod:=, MultiStart:=, TopoSearch:=, RequireBounds:=, Derivatives:=, SecondDerivatives:=)

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 argument, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

Scaling (or **ScalingOption**, for backward compatibility) 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, Solver will skip preparing the information needed to create Solver Reports. If FALSE, Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

LinearConstraints is a logical value corresponding to the Treat Constraints as Linear check box. If TRUE, the Solver will assume that all constraints are linear functions, and will compute their gradients only once at the beginning of the solution process. If FALSE, the Solver will treat the constraints as nonlinear, and will compute their gradients at each major iteration. See the chapter "Solver Options" for a further discussion of this option.

LinearObjective is a logical value corresponding to the Treat Objective as Linear check box. If TRUE, the Solver will assume that the objective is a linear function, and will compute its gradient only once at the beginning of the solution process. If FALSE, the Solver will treat the objective as nonlinear, and will compute its gradient at each major iteration. See the chapter "Solver Options" for a further discussion of this option.

RelaxBounds is a logical value corresponding to the Relax Bounds on Variables check box. If TRUE, the Solver may recalculate the model with values for the variables that are slightly outside their bounds, in order to speed the solution process. If FALSE, the Solver will use values for the variables that are always within the lower and upper bounds you specify.

SolutionMethod is the number 1, 2, 3 or 4 and corresponds to the Solution Method option: 1 for Select Automatically, 2 for Interior Point Direct, 3 for Interior Point CG, and 4 for Active Set.

MultiStart is a logical value corresponding to the Multistart Search check box. If TRUE, the Solver will use Multistart Search, in conjunction with the LSSQP Solver, to seek a globally optimal solution. If FALSE, the LSSQP 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, and if Multistart Search is selected, 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.

Derivatives is the number 1 or 2 and corresponds to the Derivatives option: 1 for Forward and 2 for Central.

SecondDerivatives is the number 1, 2 or 3 and corresponds to the Second Derivatives option: 1 for Analytic 2nd Derivatives, 2 for Analytic 1st Derivatives, and 3 for Finite Differences.

OptQuest Solver Functions

The OptQuest Solver provides the following traditional VBA functions to either set (with SolverOPTQOptions) or get (with SolverOPTQGet) the current option and parameter settings that affect the performance of this Solver engine.

All of these options and parameters may also be set via the object-oriented API, as described above under “Accessing Object Properties.” Engine.Param names are listed for each option in the chapter “Solver Options;” these names normally match the named arguments listed below for SolverOPTQOptions.

SolverOPTQGet

Returns OptQuest Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the OptQuest Solver is selected in the Solver Engines dropdown list.

VBA Syntax

SolverOPTQGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

TypeNum	Returns
1	The Max Time value (as a number in seconds)
2	The Iterations value (max number of trial solutions)
3	The Objective Function Precision (as a decimal number)
4	The Decision Variable Precision (as a decimal number)
5	The Population Size (as an integer number – no longer used)
6	The Boundary Frequency parameter (as a decimal number – no longer used)
7	TRUE if the Use Same Sequence of Random Numbers check box is selected; FALSE otherwise
8	The Random Number Seed value (as a decimal number)
9	TRUE if the Solve Without Integer Constraints check box is selected; FALSE otherwise

10	**TRUE if the Check for Duplicated Solutions check box is selected; FALSE otherwise**
11	**TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise.**
12	**TRUE if the Assume Non-Negative check box is selected; FALSE otherwise**
13	**TRUE if the Show Iteration Results check box is selected; FALSE otherwise**
14	**The Number of Solutions to Report (as a decimal number)**
15	**TRUE if the Stop when Objective Hasn't Improved check box is selected; FALSE otherwise**
16	**The Iterations limit for Stop when Objective Hasn't Improved (as a decimal number)**
17	**The percentage limit for Auto Stop Percentage**
18	**TRUE if Use Psi is selected; FALSE otherwise**
19	**TRUE if Check NOnlinear 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.

SolverOPTQOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the OptQuest Solver is selected in the Solver Engines dropdown list. Specifies options for the OptQuest Solver.

VBA Syntax

SolverOPTQOptions (MaxTime:=, MaxTrialSol:=, ObjPrecision:=, VarPrecision:=, PopulationSize:=, BoundFreq:=, FixedSeed:=, RandomSeed:=, SolveWithout:=, CheckDup:=, BypassReports:=, AssumeNonneg:=, StepThru:=, NumSolutions:=, AutoStop:=, AutoStopIter:=, AutoStopPerc:=, UsePSI:=, CheckNonlinear:=)

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 is an integer greater than zero corresponding to the Max Time edit box.

MaxTrialSol is an integer greater than zero corresponding to the Iterations edit box.

ObjPrecision is a number between zero and one corresponding to the Precision (Obj Fun) edit box.

VarPrecision is a number between zero and one corresponding to the Precision (Dec Var) edit box.

PopulationSize is an integer corresponding to the Population Size edit box (no longer used).

BoundFreq is a number between zero and one corresponding to the Boundary Freq edit box (no longer used).

FixedSeed is a logical value corresponding to the Use Same Sequence of Random Numbers check box. If TRUE, the Solver uses a fixed random number seed that is supplied in the RandomSeed parameter. If FALSE, the Solver does not use a fixed random number seed.

RandomSeed is an integer between zero and 999 corresponding to the random number seed edit box.

SolveWithout is a logical value corresponding to the Solve Without Integer Constraints check box. If TRUE, the Solver ignores the integer constraints for decision variables. If FALSE, no action is taken.

CheckDup is a logical value corresponding to the Check for Duplicated Solutions check box. If TRUE, the Solver checks for duplicated solutions. If FALSE, the Solver does not check for duplicated solutions.

BypassReports is a logical value corresponding to the Bypass Solver Reports check box. If TRUE, Solver will skip preparing the information needed to create Solver Reports. If FALSE, Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

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.

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.

NumSolutions is an integer greater than zero corresponding to the Number of Solutions to Report edit box.

AutoStop is a logical value corresponding to the Stop when Objective Hasn't Improved check box. If TRUE, the Solver stops when the objective value of the best solution found so far hasn't improved after *AutoStopIter* iterations. If FALSE, no action is taken.

AutoStopIter is an integer greater than zero corresponding to Stop when Objective Hasn't Improved after Iterations edit box.

AutoStopPerc is a fractional value greater than zero corresponding to Stop when Obejective Hasn't Improved after Percentage edit box.

UsePsi is a logical value corresponding to the Use Psi Interpreter eheckbox. If TRUE, Solver will use the Psi Interpreter to generated 1st derivatives.

CheckNonlinear is a logical value corresponding to the Check Nonlinear Constraints edit box.

Functions for Mixed-Integer Problems

Analytic Solver provides the following VBA functions to either set (with SolverIntOptions) or get (with SolverIntGet) the current option and parameter settings that affect the performance of the Large-Scale GRG, Large-Scale SQP, and Knitro Solver engines on mixed-integer problems.

All of these options and parameters may also be set via the object-oriented API, as described above under "Accessing Object Properties." Engine.Param names are listed for each option in the chapter "Solver Options;" these names normally match the named arguments listed below for SolverIntOptions.

SolverIntGet

Returns integer option settings for the current Solver problem on the specified sheet. These settings are entered on the Integer Options dialog, or the Integer 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)

The Large-Scale GRG and Knitro Solver engines support return values for **TypeNum** = 1 through 5 only. The Large-Scale SQP and MOSEK Solver engines support return values for **TypeNum** = 1 through 9, 13 and 14. **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 Premium Solver... from the Add-Ins tab, choosing the Options button in the Solver Parameters dialog box, then clicking the Integer tab in the Solver Options dialog. Specifies integer options for Solver engines (other than the Interval Global, Evolutionary, Large-Scale LP/QP, and OptQuest Solvers).

VBA Syntax

SolverIntOptions (MaxSubproblems:=, MaxIntegerSols:=, IntTolerance:=, IntCutoff:=, SolveWithout:=, UseDual:=, ProbingFeasibility:=, BoundsImprovement:=, OptimalityFixing:=, VariableReordering:=, UsePrimalHeuristic:=, MaxGomoryCuts:=, GomoryPasses:=, MaxKnapsackCuts:=, KnapsackPasses:=)

The arguments correspond to the options on the Integer 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.

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 and above 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 (in Analytic Solver Comprehensive and Analytic Solver Optimization's LP/Quadratic Solver Options dialog).

GomoryPasses must be an integer greater than or equal to zero. It corresponds to the Gomory Passes edit box (in Analytic Solver Comprehensive and Analytic Solver Optimization’s LP/Quadratic Solver Options dialog).

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.

The Large-Scale SQP Solver supports only the arguments ProbingFeasibility, BoundsImprovement, OptimalityFixing, UsePrimalHeuristic, MaxKnapsackCuts and KnapsackPasses.

Large-Scale LP/QP Solver Functions

The Large-Scale LP/QP Solver provides the following VBA functions to either set or get the current option and parameter settings that affect the performance of this Solver engine. The SolverLSLPOptions and SolverLSLPGet functions affect the options on the General tab, and the SolverLSLPIntOptions and SolverLSLPIntGet functions affect the options on the Integer tab of the Large-Scale LP/QP Solver Options dialog.

All of these options and parameters may also be set via the object-oriented API, as described above under “Accessing Object Properties.” Engine.Param names are listed for each option in the chapter “Solver Options;” these names normally match the named arguments listed for SolverLSLPOptions and SolverLSLPIntOptions.

SolverLSLPGet

Returns Large-Scale LP/QP Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the Large-Scale LP/QP Solver is selected in the Solver Engines dropdown list.

VBA Syntax

SolverLSLPGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

TypeNum	Returns
1	The Max Time value (as a number in seconds)
2	The Iterations value (max number of iterations)
3	The Primal Tolerance (as a decimal number)
4	The Dual Tolerance (as a decimal number)
5	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
6	TRUE if the Use Automatic Scaling check box is selected; FALSE otherwise
7	TRUE if the Show Iteration Results check box is selected; FALSE otherwise

8	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise.
9	TRUE if the Do Presolve check box is selected; FALSE otherwise
10	TRUE if Assume Quadratic Objective is selected; FALSE otherwise (no longer used)
11	A number corresponding to the type of Derivatives: 1 = Forward 2 = Central
12	True if the Classic Search 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.

SolverLSLPIntGet

Returns Large-Scale LP/QP Solver integer option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog, on the Integer tab, when the Large-Scale LP/QP Solver is selected in the Solver Engines dropdown list.

VBA Syntax

SolverLSLPIntGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

TypeNum	Returns
1	The Max Subproblems value (as a decimal number)
2	The Max Feasible 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	(Compatibility Option) The Maximum Cut Passes at Root value (as a decimal number)
7	(Compatibility Option) The Maximum Cut Passes in Tree value (as a decimal number)
8	(Compatibility Option) TRUE if the Use Strong Branching check box is selected; FALSE otherwise
9	(Compatibility option) TRUE if the Lift and Cover check box is selected; FALSE otherwise.
10	(Compatibility option) TRUE if the Rounding check box is selected; FALSE otherwise
11	(Compatibility Option) TRUE if the Knapsack check box is selected; FALSE otherwise
12	(Compatibility Option) TRUE if the Gomory check box is selected; FALSE otherwise.
13	(Compatibility Option) TRUE if the Probing check box is selected; FALSE otherwise

`14`	`(Compatibility option) TRUE if the Odd Hole check box is selected; FALSE otherwise`
`15`	`(Compatibility Option) TRUE if the Clique check box is selected;FALSE otherwise`
`16`	`(Compatibility Option) TRUE if the Rounding Heuristic check box is selected; FALSE otherwise`
`17`	`(Compatibility Option) TRUE if the Local Search Heuristic check box is selected; FALSE otherwise`
`18`	`(Compatibility Option) TRUE if the Flow Cover check box is selected; FALSE otherwise`
`19`	`(Compatibility Option) TRUE if the Mixed Integer Rounding check box is selected; FALSE otherwise`
`20`	`(Compatibility Option) TRUE if the Two Mixed Integer Rounding check box is selected; FALSE otherwise`
`21`	`(Compatibility Option) TRUE if the Reduce and Split check box is selected; FALSE otherwise`
`22`	`(Compatibility Option) TRUE if the Special Ordered Sets check box is selected; FALSE otherwise`
`23`	`TRUE if the Preprocessing check box is selected; FALSE otherwise`
`24`	`TRUE if the Feasibility Pump check box is selected; FALSE otherwise`
`25`	`TRUE if the Greedy Cover Heuristic check box is selected; FALSE otherwise`
`26`	`TRUE if the Local Tree 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.

SolverLSLPOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the Large-Scale LP/QP Solver is selected in the Solver Engines dropdown list. Specifies options for the Large-Scale LP/QP Solver.

SolverLSLPOptions MaxTime:=1, Iterations:=2, PrimalTolerance:=0.0000003, _

DualTolerance:=0.0000004, AssumeNonneg:=True, ScalingOption:=False, StepThru:= _

True, BypassReports:=False, Presolve:=True, Derivatives:=1, ClassicSearch:= _

False

VBA Syntax

SolverLSLPOptions (MaxTime:=, Iterations:=, PrimalTolerance:=, DualTolerance:=, StepThru:=, ScalingOption:=, AssumeNonneg:=, BypassReports:=, Presolve:=, Derivatives:=, AssumeQP:=, CoeffTol:=, SolutionTol:=, PivotTol:=, ReducedTol:=, Crash:=, ClassicSearch:=)

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.

PrimalTolerance must be a number between zero and one, but not equal to zero or one. It corresponds to the Primal Tolerance edit box.

DualTolerance must be a number between zero and one, but not equal to zero or one. It corresponds to the Dual Tolerance 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 argument, your function will be called each time Solver pauses; otherwise the standard Show Trial Solution dialog box will appear.

Scaling (or **ScalingOption**, for backward compatibility) is a logical value corresponding to the Use Automatic Scaling checkbox. 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, Solver will skip preparing the information needed to create Solver Reports. If FALSE, Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

Presolve is a logical value corresponding to the Do Presolve checkbox. If TRUE, the Solver will perform a Presolve step before applying the Primal or Dual Simplex method.

Derivatives is the number 1 or 2 and corresponds to the Derivatives option: 1 for Forward and 2 for Central.

ClassicSearch is a logical value corresponding to the Classic Search check box. If TRUE, then Solver will use the same search methods as previous releases for problems with integer variables and quadratically constrained problems (QCPs). If FALSE then Solver will use the new (default) search methods.

AssumeQP is included for compatibility with earlier versions, but is ignored.

CoeffTol is included for compatibility with earlier versions, but is ignored.

SolutionTol is included for compatibility with earlier versions, but is ignored

PivotTol is included for compatibility with earlier versions, but is ignored.

ReducedTol is included for compatibility with earlier versions, but is ignored.

Crash is included for compatibility with earlier versions, but is ignored.

SolverLSLPIntOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the Large-Scale LP/QP Solver is selected in the Solver Engines dropdown list, and selecting the Integer tab. Specifies integer options for the Large-Scale LP/QP Solver.

VBA Syntax

SolverLSLPIntOptions (MaxSubproblems:=, MaxIntegerSols:=, IntTolerance:=, IntCutoff:=, SolveWithout:=, MaxRootCutPasses:=, MaxTreeCutPasses:=, StrongBranching:=, PreProcess:=, KnapsackCuts:=, GomoryCuts:=, MirCuts:=, ProbingCuts:=, TwoMirCuts:=, CliqueCuts:=, RedSplitCuts:=, FlowCoverCuts:=, LocalTree:=, SOSCuts:=, GreedyCover:=, FeasibilityPump:=, LocalHeur:=, RoundingHeur:=, LiftAndCoverCuts:=, RoundingCuts:=, OddHoleCuts:=)

The arguments correspond to the options in the Integer tab of the 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. The last three arguments are included for compatibility with earlier Large-Scale LP/QP Solver versions.

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 Feasible 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.

MaxRootCutPasses must be an integer greater than or equal to -1. It corresponds to the Maximum Cut Passes at Root edit box. A value of -1 means that the Solver should choose the number of passes automatically.

MaxTreeCutPasses must be an integer greater than or equal to zero. It corresponds to the Maximum Cut Passes in Tree edit box.

StrongBranching is a logical value corresponding to the Use Strong Branching check box. If TRUE, strong branching will be performed throughout the Branch & Bound process.

PreProcess is a logical value corresponding to the Preprocessing check box. If TRUE, preprocessing will be performed.

KnapsackCuts is a logical value corresponding to the Knapsack check box. If TRUE, Knapsack cuts will be generated.

GomoryCuts is a logical value corresponding to the Gomory check box. If TRUE, Gomory cuts will be generated.

MirCuts is a logical value corresponding to the Mixed Integer Rounding check box. If TRUE, Mixed Integer Rounding cuts will be generated.

ProbingCuts is a logical value corresponding to the Probing check box. If TRUE, Probing cuts will be generated.

TwoMirCuts is a logical value corresponding to the Two Mixed Integer Rounding check box. If TRUE, Two Mixed Integer Rounding cuts will be generated.

CliqueCuts is a logical value corresponding to the Odd Hole check box. If TRUE, Clique cuts will be 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.

FlowCoverCuts is a logical value corresponding to the Flow Cover check box. If TRUE, Flow Cover cuts will be generated.

LocalTree is a logical value corresponding to the Local Tree check box. If TRUE, when new incumbents are found, a local Branch & Bound tree search will be used to seek improved integer solutions.

SOSCuts is a logical value corresponding to the Special Ordered Sets check box. If TRUE, cuts for Special Ordered Sets will be generated.

GreedyCover is a logical value corresponding to the Greedy Cover check box. If TRUE, the Greedy Cover search heuristic will be used to seek improved integer solutions.

FeasibilityPump is a logical value corresponding to the Feasibility Pump check box. If TRUE, the Feasibility Pump heuristic will be used to seek improved integer solutions.

LocalHeur is a logical value corresponding to the Local Search Heuristic check box. If TRUE, the local search heuristic will be used to seek improved integer solutions.

RoundingHeur is a logical value corresponding to the Rounding Heuristic check box. If TRUE, the rounding heuristic will be used to seek improved integer solutions.

LiftAndCoverCuts is a logical value, included for compatibility with earlier Large-Scale LP/QP Solver versions. If TRUE, Lift and Cover cuts will be generated.

RoundingCuts is a logical value, included for compatibility with earlier Large-Scale LP/QP Solver versions. If TRUE, Rounding cuts will be generated.

OddHoleCuts is a logical value, included for compatibility with earlier Large-Scale LP/QP Solver versions. If TRUE, Odd Hole cuts will be generated.

MOSEK Solver Functions

The MOSEK Solver provides the following traditional VBA functions to either set or get the current option and parameter settings that affect the performance of this Solver engine. The SolverMOSEKOptions and SolverMOSEKGet functions affect the options in the General, LP/QP/QCP and Conic sections of the Engine tab on the Solver Task Pane (when the Mosek Solver Engine is selected). The SolverMOSEKIntOptions and SolverMOSEKIntGet functions affect the options in the Integer section on the Engine tab of the Solver Task Pane, when Mosek Solver Engine is selected.

All of these options and parameters may also be set via the object-oriented API, as described above under “Accessing Object Properties.” Engine.Param names are listed for each option in the chapter “Solver Options;” these names normally match the named arguments listed below for SolverMOSEKOptions and SolverMOSEKIntOptions.

SolverMOSEKGet

Returns MOSEK Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog when the MOSEK Solver is selected in the Solver Engines dropdown list.

VBA Syntax

SolverMOSEKGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

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 Pivot Tolerance (as a decimal number)
5	TRUE if the Show Iteration Results check box is selected; FALSE otherwise
6	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
7	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise.
8	A number corresponding to the type of Ordering: 1 = Automatic Choice 2 = Local Fill-In 1 3 = Local Fill-In 2 4 = Graph Partitioning 5 = Alt. Graph Partitioning 6 = No Ordering
9	A number corresponding to the type of Scaling: 1 = Automatic Choice 2 = Aggressive Scaling 3 = No Scaling 4 = Conservative Scaling
10	The Dual Feasibility Tolerance value on the LP/QP/QCP tab (as a decimal number)
11	The Primal Feasibility Tolerance value on the LP/QP/QCP tab (as a decimal number)
12	The Model Feasibility Tolerance value on the LP/QP/QCP tab (as a decimal number)
13	The Complementarity Gap Tolerance value on the LP/QP/QCP tab (as a decimal number)
14	The Central Path Tolerance value on the LP/QP/QCP tab (as a decimal number)
15	The Gap Termination Tolerance value on the LP/QP/QCP tab (as a decimal number)
16	The Relative Step Size value on the LP/QP/QCP tab (as a decimal number)

17	The Dual Feasibility Tolerance value on the Conic tab (as a decimal number)
18	The Primal Feasibility Tolerance value on the Conic tab (as a decimal number)
19	The Model Feasibility Tolerance value on the Conic tab (as a decimal number)
20	The Complementarity Gap Tolerance value on the Conic tab (as a decimal number)
21	The Gap Termination Tolerance value on the Conic tab (as a decimal number)
22	The Dual Feasibility Tolerance value on the Nonlinear tab (as a decimal number)
23	The Primal Feasibility Tolerance value on the Nonlinear tab (as a decimal number)
24	The Complementarity Gap Tolerance value on the Nonlinear tab (as a decimal number)
25	The Gap Termination Tolerance value on the Nonlinear tab (as a decimal number)
26	The Relative Step Size value on the Nonlinear tab (as a decimal number)
27	Backward compatibility only. The current version of Mosek does not support the solving of nonlinear models. The NonConvex Feasibility Tol value on the Nonlinear tab (as a decimal number)
28	Backward compatibility only. The current version of Mosek does not support the solving of nonlinear models. The NonConvex Optimality Tol value on the Nonlinear tab as a decimal number)
29	Backward compatibility only. The current version of Mosek does not support the solving of nonlinear models. The maximum number of barrier iterations on the General tab (as an integer number)

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.

SolverMOSEKIntGet

Returns MOSEK Solver integer option settings for the current Solver problem on the specified sheet. These settings are entered in the Solver Options dialog, on the Integer tab, when the MOSEK Solver is selected in the Solver Engines dropdown list.

VBA Syntax

SolverMOSEKIntGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

TypeNum	Returns
1	The Max Subproblems value (as a decimal number)
2	The Max Feasible 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`	`The Maximum Cut Passes at Root value (as a decimal number)`
`7`	`The Maximum Cut Passes in Tree value (as a decimal number)`
`8`	`TRUE if the Preprocessing check box is selected; FALSE otherwise`
`9`	`TRUE if the Knapsack check box is selected; FALSE otherwise`
`10`	`TRUE if the Gomory check box is selected; FALSE otherwise.`
`11`	`TRUE if the Mixed Integer Rounding check box is selected; FALSE otherwise`
`12`	`TRUE if the Probing check box is selected; FALSE otherwise`
`13`	`TRUE if the Two Mixed Integer Rounding check box is selected; FALSE otherwise`
`14`	`TRUE if the Clique check box is selected; FALSE otherwise`
`17`	`TRUE if the Reduce and Split check box is selected; FALSE otherwise`
`14`	`TRUE if the Flow Cover check box is selected; FALSE otherwise`
`15`	`TRUE if the Local Tree check box is selected; FALSE otherwise`
`16`	`TRUE if the Special Ordered Sets check box is selected; FALSE otherwise`
`17`	`TRUE if the Greedy Cover Heuristic check box is selected; FALSE otherwise`
`18`	`TRUE if the Feasibility Pump check box is selected; FALSE otherwise`
`19`	`TRUE if the Local Search Heuristic check box is selected; FALSE otherwise`
`20`	`TRUE if the Rounding Heuristic 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.

SolverMOSEKOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the MOSEK Solver is selected in the Solver Engines dropdown list. Specifies options for the MOSEK Solver.

VBA Syntax

SolverMOSEKOptions (MaxTime:=, Iterations:=, Precision:=, PivotTolerance:=, StepThru:=, AssumeNonneg:=, BypassReports:=,

Ordering:=, Scaling:=, DualFeasibility:=, PrimalFeasibility:=, ModelFeasibility:=, CompGapTol:=, CentralPathTol:=, GapTerminationTol:=, StepSize:=, DualFeasibilityCone:=, PrimalFeasibilityCone:=, ModelFeasibilityCone:=, CompGapTolCone:=, GapTerminationTolCone:=, DualFeasibilityNLP:=, PrimalFeasibilityNLP:=, CompGapTolNLP:=, GapTerminationTolNLP:=, StepSizeNLP:=, NCvxFeasibilityTol:=, NCvxOptimalityTol:=)

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.

PivotTolerance must be a number between zero and one, but not equal to zero or one. It corresponds to the Pivot Tolerance 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 argument, 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, Solver will skip preparing the information needed to create Solver Reports. If FALSE, Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

Ordering is the number 1, 2, 3, 4 or 5 and corresponds to the Ordering option group: 1 for Automatic Choice, 2 for Local Fill-In 1, 3 for Local Fill-In 2, 4 for Graph Partitioning, 5 for Alt. Graph Partitioning, and 6 for No Ordering.

Scaling (or **ScalingOption**) is the number 1, 2, 3 or 4 and corresponds to the Scaling option group: 1 for Automatic Choice, 2 for Aggressive Scaling, 3 for No Scaling, and 4 for Conservative Scaling.

DualFeasibility must be a number between zero and one, but not equal to zero or one. It corresponds to the Dual Feasibility Tolerance edit box on the LP/QP/QCP tab.

PrimalFeasibility must be a number between zero and one, but not equal to zero or one. It corresponds to the Primal Feasibility Tolerance edit box on the LP/QP/QCP tab.

ModelFeasibility must be a number between zero and one, but not equal to zero or one. It corresponds to the Model Feasibility Tolerance edit box on the LP/QP/QCP tab.

CompGapTol must be a number between zero and one, but not equal to zero or one. It corresponds to the Complementarity Gap Tolerance edit box on the LP/QP/QCP tab.

CentralPathTol must be a number between zero and one, but not equal to zero or one. It corresponds to the Central Path Tolerance edit box on the LP/QP/QCP tab.

GapTerminationTol must be a number between zero and one, but not equal to zero or one. It corresponds to the Gap Termination Tolerance edit box on the LP/QP/QCP tab.

StepSize must be a number between zero and one, but not equal to zero or one. It corresponds to the Relative Step Size edit box on the LP/QP/QCP tab.

DualFeasibilityCone must be a number between zero and one, but not equal to zero or one. It corresponds to the Dual Feasibility Tolerance edit box on the Conic tab.

PrimalFeasibilityCone must be a number between zero and one, but not equal to zero or one. It corresponds to the Primal Feasibility Tolerance edit box on the Conic tab.

ModelFeasibilityCone must be a number between zero and one, but not equal to zero or one. It corresponds to the Model Feasibility Tolerance edit box on the Conic tab.

CompGapTolCone must be a number between zero and one, but not equal to zero or one. It corresponds to the Complementarity Gap Tolerance edit box on the Conic tab.

GapTerminationTolCone must be a number between zero and one, but not equal to zero or one. It corresponds to the Gap Termination Tolerance edit box on the Conic tab.

DualFeasibilityNLP must be a number between zero and one, but not equal to zero or one. It corresponds to the Dual Feasibility Tolerance edit box on the Nonlinear tab. (For backwards compatibility only. The current Mosek Solver Engine does not support nonlinear models.)

PrimalFeasibilityNLP must be a number between zero and one, but not equal to zero or one. It corresponds to the Primal Feasibility Tolerance edit box on the Nonlinear tab. (For backwards compatibility only. The current Mosek Solver Engine does not support nonlinear models.)

CompGapTolNLP must be a number between zero and one, but not equal to zero or one. It corresponds to the Complementarity Gap Tolerance edit box on the Nonlinear tab. (For backwards compatibility only. The current Mosek Solver Engine does not support nonlinear models.)

GapTerminationTolNLP must be a number between zero and one, but not equal to zero or one. It corresponds to the Gap Termination Tolerance edit box on the Nonlinear tab. (For backwards compatibility only. The current Mosek Solver Engine does not support nonlinear models.)

StepSizeNLP must be a number between zero and one, but not equal to zero or one. It corresponds to the Relative Step Size edit box on the Nonlinear tab. (For backwards compatibility only. The current Mosek Solver Engine does not support nonlinear models.)

NCvxFeasibilityTol must be a number between zero and one, but not equal to zero or one. It corresponds to the NonConvex Feasibility Tol edit box on the Nonlinear tab. (For backwards compatibility only. The current Mosek Solver Engine does not support nonlinear models.)

NCvxOptimalityTol must be a number between zero and one, but not equal to zero or one. It corresponds to the NonConvex Optimality Tol edit box on the Nonlinear tab. (For backwards compatibility only. The current Mosek Solver Engine does not support nonlinear models.)

SolverMOSEKIntOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the MOSEK Solver is selected in the Solver Engines dropdown list, and selecting the Integer tab. Specifies integer options for the MOSEK Solver.

VBA Syntax

SolverMOSEKIntOptions (MaxSubproblems:=, MaxIntegerSols:=, IntTolerance:=, IntCutoff:=, SolveWithout:=, MaxRootCutPasses:=, MaxTreeCutPasses:=, PreProcess:=, KnapsackCuts:=, GomoryCuts:=, MirCuts:=, ProbingCuts:=, TwoMirCuts:=, CliqueCuts:=, RedSplitCuts:=, FlowCoverCuts:=, LocalTree:=, SOSCuts:=, GreedyCover:=, FeasibilityPump:=, LocalHeur:=, RoundingHeur:=)

The arguments correspond to the options in the Integer tab of the 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.

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 Feasible 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.

MaxRootCutPasses must be an integer greater than or equal to -1. It corresponds to the Maximum Cut Passes at Root edit box. A value of -1 means that the Solver should choose the number of passes automatically.

MaxTreeCutPasses must be an integer greater than or equal to zero. It corresponds to the Maximum Cut Passes in Tree edit box.

PreProcess is a logical value corresponding to the Preprocessing check box. If TRUE, preprocessing will be performed.

KnapsackCuts is a logical value corresponding to the Knapsack check box. If TRUE, Knapsack cuts will be generated.

GomoryCuts is a logical value corresponding to the Gomory check box. If TRUE, Gomory cuts will be generated.

MirCuts is a logical value corresponding to the Mixed Integer Rounding check box. If TRUE, Mixed Integer Rounding cuts will be generated.

ProbingCuts is a logical value corresponding to the Probing check box. If TRUE, Probing cuts will be generated.

TwoMirCuts is a logical value corresponding to the Two Mixed Integer Rounding check box. If TRUE, Two Mixed Integer Rounding cuts will be generated.

CliqueCuts is a logical value corresponding to the Odd Hole check box. If TRUE, Clique cuts will be 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.

FlowCoverCuts is a logical value corresponding to the Flow Cover check box. If TRUE, Flow Cover cuts will be generated.

LocalTree is a logical value corresponding to the Local Tree check box. If TRUE, when new incumbents are found, a local Branch & Bound tree search will be used to seek improved integer solutions.

SOSCuts is a logical value corresponding to the Special Ordered Sets check box. If TRUE, cuts for Special Ordered Sets will be generated.

GreedyCover is a logical value corresponding to the Greedy Cover check box. If TRUE, the Greedy Cover search heuristic will be used to seek improved integer solutions.

FeasibilityPump is a logical value corresponding to the Feasibility Pump check box. If TRUE, the Feasibility Pump heuristic will be used to seek improved integer solutions.

LocalHeur is a logical value corresponding to the Local Search Heuristic check box. If TRUE, the local search heuristic will be used to seek improved integer solutions.

RoundingHeur is a logical value corresponding to the Rounding Heuristic check box. If TRUE, the rounding heuristic will be used to seek improved integer solutions.

Gurobi Solver Basic Functions

The Gurobi Solver provides the following VBA functions to either set (with SolverGurobiOptions) or get (with SolverGurobiGet) the general option and parameter settings that affect the performance of this Solver engine. These options and parameters appear on the General tab in the Gurobi Solver Options tabbed dialog box. The options and parameters that appear on other tabs are described in following sections.

All of these options and parameters may also be set via the object-oriented API, as described above under “Accessing Object Properties.” Engine.Param names are listed for each option in the chapter “Solver Options;” these names normally match the named arguments listed below for SolverGurobiOptions.

SolverGurobiGet

Returns general Gurobi Solver option settings for the current Solver problem on the specified sheet. These settings appear on the General tab in the Gurobi Solver Options tabbed dialog box.

VBA Syntax

SolverGurobiGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

TypeNum	Returns
1	The Max Time value (as a number in seconds)
2	The Iterations value (max number of iterations)
3	The Feasibility Tolerance (as a decimal number)
4	The Optimality Tolerance (as a decimal number)
5	The Integrality Tolerance (as a decimal number)
6	TRUE if the Assume Non-Negative check box is selected; FALSE otherwise
7	TRUE if Show Iteration Result check box is selected; FALSE otherwise
8	TRUE if the Bypass Solver Reports check box is selected; FALSE otherwise.
9	A number corresponding to the Method option: 0 = Primal Simplex 1 = Dual Simplex 2 = Barrier
10	A number corresponding to the Presolve option: -1 = Determine automatically 0 = Presolve off 1 = Conservative 2 = Aggressive
11	A number corresponding to the Pricing option: -1 = Determine automatically 0 = Partial pricing 1 = Steepest Edge 2 = Devex 3 = Quickstart Steepest Edge

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.

SolverGurobiOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the Gurobi Solver is selected in the Solver Engines dropdown list. Specifies options that appear on the General tab in the Gurobi Solver Options tabbed dialog box.

VBA Syntax

SolverGurobiOptions (MaxTime:=, Iterations:=, FeasibilityTol:=, OptimalityTol:=, IntegralityTol:=, AssumeNoneg:=, StepThru:=, BypassReports:=, LPMethod:=, Presolve:=, Pricing:=)

The arguments correspond to the General tab in the Gurobi Solver Options tabbed 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.

FeasibilityTol is a number between zero and one, corresponding to the Feasibility Tolerance edit box.

OptimalityTol is a number between zero and one, corresponding to the Optimality Tolerance edit box.

IntegralityTol is a number between zero and one, corresponding to the Integrality Tolerance edit 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.

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.

BypassReports is a logical value corresponding to the Bypass Solver Reports check box. If TRUE, Solver will skip preparing the information needed to create Solver Reports. If FALSE, Solver will prepare for the reports. For large models, bypassing the Solver Reports can speed up the solution considerably.

LPMethod corresponds to the Method group of options:

LPMethod	**Option specified**
0	Primal Simplex
1	Dual Simplex
2	Barrier method

Presolve corresponds to the Presolve group of options:

Crashing	**Option specified**
-1	Determine automatically
0	Presolve off
1	Conservative
2	Aggressive

Pricing corresponds to the Pricing group of options:

Pricing	**Option specified**
-1	Determine automatically
0	Use Partial pricing
1	Use Steepest Edge pricing
2	Use Devex pricing
3	Use Quickstart Steepest Edge pricing

Gurobi Solver Barrier Functions

The Gurobi Solver provides the following VBA functions to either set (with SolverGurobiBarrierOptions) or get (with SolverGurobiBarrierGet) the option and parameter settings that affect the performance of the Barrier method in this Solver engine. These options and parameters appear on the Barrier tab in the Gurobi Solver Options tabbed dialog box.

All of these options and parameters may also be set via the object-oriented API, as described above under "Accessing Object Properties." Engine.Param names are listed for each option in the chapter "Solver Options;" these names normally match the named arguments listed below for SolverGurobiBarrierOptions.

SolverGurobiBarrierGet

Returns Gurobi Solver Barrier option settings for the current Solver problem on the specified sheet. These settings appear on the Barrier tab in the Gurobi Solver Options tabbed dialog box.

VBA Syntax

SolverGurobiBarrierGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

```
TypeNum   Returns

1          The Iteration Limit value (as a decimal number)

2          The Convergence Tolerance (as a decimal number)

3          A number corresponding to the Crossover Strategy option:
          -1 = Determine automatically
           0 = No crossover
           1 = Primal Simplex (dual variables first)
           2 = Dual Simplex (dual variables first)
           3 = Primal Simplex (primal variables first)
           4 = Dual Simplex (primal variables first)

4          A number corresponding to the Ordering option:
          -1 = Determine automatically
           0 = Approximate Minimum Degree
           1 = Nested Dissection
```

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.

SolverGurobiBarrierOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the Gurobi Solver is selected in the Solver Engines dropdown list. Specifies options that appear on the Barrier tab in the Gurobi Solver Options tabbed dialog box.

VBA Syntax

SolverGurobiBarrierOptions (BarIterLimit:=, BarConvTol:=, CrossOver:=, Ordering:=)

The arguments correspond to the Barrier tab in the Gurobi Solver Options tabbed 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.

BarIterLimit must be an integer greater than zero. It corresponds to the Iteration Limit edit box.

BarConvTol is a number between zero and one, corresponding to the Convergence Tolerance edit box.

CrossOver corresponds to the Crossover Strategy group of options:

CrossOver	**Option specified**
-1	Determine automatically
0	No crossover
1	Primal Simplex (dual variables first)
2	Dual Simplex (dual variables first)
3	Primal Simplex (primal variables first)
4	Dual Simplex (primal variables first)

Ordering corresponds to the Method group of options:

Ordering	**Option specified**
-1	Determine automatically
0	Approximate Minimum Degree
1	Nested Dissection

Gurobi Solver Integer Functions

The Gurobi Solver provides the following VBA functions to either set (with SolverGurobiIntOptions) or get (with SolverGurobiIntGet) the option and parameter settings that affect the mixed-integer performance of this Solver engine. These options and parameters appear on the MIP and Advanced MIP tabs in the Gurobi Solver Options tabbed dialog box.

All of these options and parameters may also be set via the object-oriented API, as described above under “Accessing Object Properties.” Engine.Param names are listed for each option in the chapter “Solver Options;” these names normally match the named arguments listed below for SolverGurobiIntOptions.

SolverGurobiIntGet

Returns Gurobi Solver mixed-integer option settings for the current Solver problem on the specified sheet. These settings appear on the MIP and Advanced MIP tabs in the Gurobi Solver Options tabbed dialog box.

VBA Syntax

SolverGurobiIntGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

```
TypeNum   Returns

1         The Max Subproblems value (as a decimal number)

2         The Max Feasible 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         The Threads value (as a decimal number)

6         The Max Submip Nodes value (as a decimal number)

7         The Heuristics value (as a decimal number)

8         TRUE if the Solve Without Integer Constraints check box is
          selected; FALSE otherwise

9         A number corresponding to the Variable Branching option:
         -1 = Determine automatically
```

```
          0 = Pseudo Reduced Cost Branching
          1 = Pseudo Shadow Price Branching
          2 = Maximum Infeasibility Branching
          3 = Strong Branching

10        A number corresponding to the Cut Generation option:
         -1 = Determine automatically
          0 = No cut generation
          1 = Conservative cut generation
          2 = Aggressive cut generation
          3 = Very aggressive cut generation

11        The Node File Start value (as a decimal number)

12        A number corresponding to the Root Method option:
          0 = Primal Simplex
          1 = Dual Simplex
          2 = Barrier method

13        A number corresponding to the Symmetric Detection option:
         -1 = Determine automatically
          0 = Symmetry detection off
          1 = Conservative
          2 = Aggressive

14        A number corresponding to the MIP Focus option:
          0 = Balanced approach
          1 = Feasible solutions
          2 = Prove optimality
          3 = Improve best bound
```

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.

SolverGurobiIntOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the Gurobi Solver is selected in the Solver Engines dropdown list. Specifies options that appear on the MIP and Advanced MIP tabs in the Gurobi Solver Options tabbed dialog box.

VBA Syntax

SolverGurobiIntOptions (MaxSubproblems:=, MaxIntegerSols:=, IntTolerance:=, IntCutoff:=, Threads:=, SubMips:=, Heuristics:=, SolveWithout:=, VarBranching:=, CutGeneration:=, NodeFileStart:=, RootMethod:=, Symmetry:=, MIPFocus:=)

The arguments correspond to the MIP and Advanced MIP tabs in the Gurobi Solver Options tabbed 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.

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 Feasible 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.

Threads must be an integer greater than or equal to 0. This is the number of threads that will be used in the parallel Branch & Bound algorithm. A value of 0 means the Solver should use as many threads as there are processors available.

SubMips must be an integer greater than or equal to 0. It corresponds to the Max Submip Nodes edit box.

Heuristics is a number between zero and one, corresponding to the Heuristics 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.

VarBranching corresponds to the Variable Branching group of options:

VarBranching	**Option specified**
-1	Determine automatically
0	Pseudo Reduced Cost Branching
1	Pseudo Shadow Price Branching
2	Maximum Infeasibility Branching
3	Strong Branching

CutGeneration corresponds to the Cut Generation group of options:

CutGeneration	**Option specified**
-1	Determine automatically
0	No cut generation
1	Conservative cut generation
2	Aggressive cut generation
3	Very aggressive cut generation

NodeFileStart is a decimal number greater than zero, corresponding to the Node File Start edit box.

RootMethod corresponds to the Root Method group of options:

RootMethod	**Option specified**
0	Primal Simplex
1	Dual Simplex
2	Barrier method

Symmetry corresponds to the Symmetry Detection group of options:

Symmetry	**Option specified**
-1	Determine automatically
0	Symmetry detection off
1	Conservative
2	Aggressive

MIPFocus corresponds to the MIP Focus group of options:

MIPFocus	**Option specified**
0	Balanced approach
1	Feasible solutions
2	Prove optimality
3	Improve best bound

XPRESS Solver Basic Functions

The XPRESS Solver provides the following VBA functions to either set (with SolverXPRESSOptions) or get (with SolverXPRESSGet) the most commonly used option and parameter settings that affect the performance of this Solver engine. These options and parameters appear on the "front row" of tabs in the XPRESS Solver Options tabbed dialog box. The advanced options and parameters that appear on the "back row" of tabs are described in the next section.

All of these options and parameters may also be set via the object-oriented API, as described above under "Accessing Object Properties." Engine.Param names are listed for each option in the chapter "Solver Options;" these names normally match the named arguments listed below for SolverXPRESSOptions.

SolverXPRESSGet

Returns basic XPRESS Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the "front row" of tabs in the XPRESS Solver Options tabbed dialog box.

VBA Syntax

SolverXPRESSGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

```
TypeNum   Returns

1         A number corresponding to the Algorithm option:
          1 = Default (automatically determined)
          2 = Dual Simplex
          3 = Primal Simplex
          4 = Newton Barrier

2         A number corresponding to the Scaling group of options.
          This should be the sum of one or more of the following:
          1  = Row Scaling
          2  = Column Scaling
          4  = Row Scaling Again
          8  = Maximin
          16 = Curtis-Reid
          32 = Scale by Maximum Element (0 for Geometric Mean)

3         A number corresponding to the Presolve option:
         -1 = Do Not Apply Presolve
          0 = Apply Presolve, Do Not Declare Infeasibility
          1 = Apply Presolve
          2 = Apply Presolve, Keep Redundant Bounds

4         TRUE if the Bypass Solver Reports check box is selected;
          FALSE otherwise

5         TRUE if the Assume Non-Negative check box is selected; FALSE
          otherwise

6         The Maximum Time value (as a number in seconds)

7         The RHS Tolerance (as a decimal number)

8         The Markowitz Tolerance (as a decimal number)

9         The Matrix Elements Zero Tolerance (as a decimal number)

10        A number corresponding to the Crashing option:
          0 = No crashing method
          1 = Singletons only (one pass)
```

	2 = Singletons only (multi pass) 3 = Multiple passes using slacks N = 11+ = Multiple passes (max N-10), slacks at end
11	A number corresponding to the Pricing option: 1 = Use Devex pricing -1 = Use Partial pricing 0 = Determine pricing automatically
12	TRUE if the Use 'Big M' Method check box is selected; FALSE otherwise
13	TRUE if the Use Automatic Perturbation check box is selected; FALSE otherwise
14	The Maximum Iterations value (max number of iterations)
15	The Infeasibility Penalty (as a decimal number)
16	The Perturbation Value (as a decimal number)
17	The Markowitz Tolerance for Factorization (as a decimal number)
18	A number corresponding to the Cross-Over Control option: 0 = No cross-over if Presolve is turned off 1 = Full cross-over to a basic solution
19	The Relative Duality Gap Tolerance (as a decimal number)
20	The Cache Size value (as a number in K bytes; -1 to determine automatically)
21	A number corresponding to the Cut Strategy option: -1 = Default (determined automatically) 0 = No cuts 1 = Conservative cut strategy 2 = Moderate cut strategy 3 = Aggressive cut strategy
22	The Absolute Integer Tolerance value (as a decimal number)
23	The Relative Integer Tolerance value (as a decimal number)
24	The Maximum Number of Nodes value (as a decimal number)
25	The Maximum Number of Solutions value (as a decimal number)
26	The Integer Cutoff value (as a decimal number)
27	The Amount to Add to Solution to Obtain New Cutoff value (as a decimal number)
28	The Percent to Add to Solution to Obtain New Cutoff value (as a decimal number)
29	TRUE if the Solve Without Integer Constraints check box is selected; FALSE otherwise
30	A number corresponding to the Node Selection Control option: 0 = Default – Choose automatically based on LP Matrix 1 = Local First search 2 = Best first search 3 = Local Depth first search 4 = Best first for N nodes, then Local first search 5 = Pure Depth first search
31	TRUE if the Assume Quadratic Objective check box is selected; FALSE otherwise

```
32        The number of Nodes, for the Best First then Local First
          option (as a decimal number)

33        A number corresponding to the Heuristics Strategy option:
         -1 = Default – Automatically select heuristics strategy
          0 = Use no heuristics
          1 = Use basic heuristics
          2 = Use enhanced heuristics
          3 = Use extensive heuristics

34        The Maximum Depth for Heuristics (as a decimal number)

35        The Frequency for Heuristics (as a decimal number)

36        The Maximum Nodes for Heuristics (as a decimal number)

37        The Maximum Solutions for Heuristics (as a decimal number)

38        The Number of Threads for MIP problems (as a decimal number)

39        A number corresponding to the Presolve Options group, the sum
          of one or more of the following values:
          1 = Singleton Column Removal
          2 = Singleton Row Removal
          4 = Forcing Row Removal
          8 = Dual Reductions
          16 = Redundant Row Removal
          32 = Duplicate Column Removal
          64 = Duplicate Row Removal
          128 = Strong Dual Reductions
          256 = Variable Eliminations
          512 = No IP Reductions
          1024 = No Semi-Continuous Variable Detection
          2048 = No Advanced IP Reductions
          4096 = Linearly Dependent Row Removal
          8192 = No Integer Variable and SOS Detection
```

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.

SolverXPRESSOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the XPRESS Solver is selected in the Solver Engines dropdown list. Specifies options that are entered in the “front row” of tabs in the XPRESS Solver Options tabbed dialog box.

VBA Syntax

SolverXPRESSOptions (Algorithm:=, Scaling:=, AssumeQP:=, BypassReports:=, AssumeNonneg:=, MaxTime:=, RHSTol:=, MarkowitzTol:=, MatrixTol:=, Crashing:=, Pricing:=, UseBigM:=, UsePerturb:=, Iterations:=, BigMPenalty:=, PerturbTol:=, FactorizationTol:=, Presolve:=, PresolveOptions:=, CrossOver:=, BarrierDualityGapTol:=, CacheSize:=, CutStrategy:=, AbsIntTol:=, RelIntTol:=, MaxNodes:=, MaxSolutions:=, IntCutoff:=, AbsAddCut:=, RelAddCut:=, SolveWithout:=, numThreads:=, NodeSelectionControl:=, BreadthFirst:=, HeurStrategy:=, HeurDepth:=, HeurFreq:=, HeurMaxNodes:=, HeurMaxSol:=)

The arguments correspond to the “front row” of tabs in the XPRESS Solver Options tabbed 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.

General Tab

Algorithm corresponds to the Algorithm group of options:

Algorithm	Option specified
1	Default (determined automatically)
2	Dual Simplex
3	Primal Simplex
4	Newton Barrier

Scaling (or **ScalingOption**, for backward compatibility) corresponds to the Scaling group of options. This should be the sum of one or more of the following values:

Scaling Value	Option specified
1	Row Scaling
2	Column Scaling
4	Row Scaling Again
8	Maximin
16	Curtis-Reid
32	Scale by Maximum Element (0 for Geometric Mean)

AssumeQP is a logical value corresponding to the Assume Quadratic Objective check box. If TRUE, Solver will assume that the objective is a quadratic function of the decision variables, and will seek to obtain its (constant) Hessian matrix from Analytic Solver Comprehensive or Analytic Solver Optimization; if FALSE, it will treat the objective as linear.

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.

AssumeNonneg is a logical value corresponding to the Assume Non-Negative check box. If TRUE, the 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.

MaxTime is an integer value greater than zero. It corresponds to the Maximum Time edit box.

RHSTol is a number between zero and one, corresponding to the RHS Tolerance edit box.

MarkowitzTol is a number between zero and one, corresponding to the Markowitz Tolerance edit box.

MatrixTol is a number between zero and one, corresponding to the Matrix Elements Zero Tolerance edit box.

LP Tab

Crashing corresponds to the Crashing group of options:

Crashing	Option specified
0	No crashing method
1	Singletons only (one pass)
2	Singletons only (multi pass)
3	Multiple passes using slacks
N = 11+	Multiple passes (max *N*-10), slacks at end

Pricing corresponds to the Pricing group of options:

Pricing	Option specified
1	Use Devex pricing
-1	Use Partial pricing
0	Determine pricing automatically

UseBigM is a logical value corresponding to the Use 'Big M' method check box. If TRUE, the 'Big M' method is used; if FALSE, the traditional Simplex Phase I and Phase II method is used.

UsePerturb is a logical value corresponding to the Use Automatic Perturbation check box. If TRUE, the problem will be perturbed automatically if the Simplex method encounters an excessive number of degenerate pivot steps; if FALSE, the problem will not be perturbed automatically.

Iterations is an integer value greater than zero. It corresponds to the Maximum Iterations edit box.

BigMPenalty is a number corresponding to the Infeasibility Penalty edit box.

PerturbTol is a real number greater than or equal to zero, corresponding to the Perturbation Value edit box.

FactorizationTol is a number between zero and one, corresponding to the Markowitz Tolerance for Factorization edit box.

Presolve Tab

Presolve corresponds to the Presolve group of options:

Presolve	Option specified
-1	Do Not Apply Presolve
0	Apply Presolve, Do Not Declare Infeasibility
1	Apply Presolve
2	Apply Presolve, Keep Redundant Bounds

PresolveOptions corresponds to the Presolve Options group. This should be the sum of one or more of the following values:

PresolveOptions	Option specified
1	Singleton Column Removal
2	Singleton Row Removal
4	Forcing Row Removal
8	Dual Reductions
16	Redundant Row Removal
32	Duplicate Column Removal
64	Duplicate Row Removal
128	Strong Dual Reductions
256	Variable Eliminations
512	No IP Reductions
1024	No Semi-Continuous Variable Detection
2048	No Advanced IP Reductions
4096	Linearly Dependent Row Removal
8192	No Integer Variable and SOS Detection

Newton-Barrier Tab

CrossOver corresponds to the Cross-Over Control group of options:

CrossOver	Option specified
0	No cross-over if Presolve is turned off
1	Full cross-over to a basic solution

BarrierDualityGapTol is a number between zero and one, corresponding to the Relative Duality Gap Tolerance edit box.

CacheSize is either -1 or an integer value greater than zero, specifying an amount of memory in kilobytes. It corresponds to the Cache Size edit box.

MIP Tab

CutStrategy corresponds to the Cut Strategy group of options:

Algorithm	**Option specified**
-1	Default (determined automatically)
0	No cuts
1	Conservative cut strategy
2	Moderate cut strategy
3	Aggressive cut strategy

AbsIntTol is a number corresponding to the Absolute Integer Tolerance edit box.

RelIntTol is a number between zero and one, corresponding to the Relative Integer Tolerance edit box.

MaxNodes must be an integer greater than zero. It corresponds to the Maximum Number of Nodes edit box.

MaxSolutions must be an integer greater than zero. It corresponds to the Maximum Number of Solutions edit box.

IntCutoff is a number (any value is possible) corresponding to the Integer Cutoff edit box.

AbsAddCut is a number corresponding to the Amount to Add to Solution to Obtain New Cutoff edit box.

RelAddCut is a number between zero and 100, corresponding to the Percent to Add to Solution to Obtain New 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.

numThreads is an integer greater than or equal to 0, corresponding to the Number of Threads edit box. It is used to limit the number of threads and processors used to solve mixed-integer problems; it is effective only if you've licensed the Parallel MIP option. The default value of 0 means that all of your computer's processors will be used, up to the maximum allowed by your license.

Node Selection Tab

NodeSelectionControl corresponds to the Control group of options on the Node Selection tab:

Algorithm	**Option specified**
0	Default – Choose automatically based on LP matrix
1	Local First search
2	Best First search
3	Local Depth First search
4	Best First for the first *N* nodes, then Local First search
5	Pure Depth First search

BreadthFirst is a number corresponding to the edit box in the "Best First for the first *N* nodes, then Local First search" choice in the Control group of options.

Heuristics Tab

HeurStrategy corresponds to the Strategy group of options on the Heuristics tab:

Algorithm	Option specified
-1	Default – Choose automatically
0	No Heuristics
1	Basic Heuristics
2	Enhanced Heuristics
3	Extensive Heuristics

HeurDepth must be an integer greater than or equal to zero, corresponding to the Maximum Depth edit box.

HeurFreq must be an integer greater than zero, corresponding to the Frequency edit box.

HeurMaxNodes must be an integer greater than zero, corresponding to the Maximum Nodes edit box.

HeurMaxSol must be an integer greater than zero, corresponding to the Maximum Solutions edit box.

XPRESS Solver Advanced Functions

The XPRESS Solver provides the following VBA functions to either set (with SolverXPRESSAdvancedOptions) or get (with SolverXPRESSAdvancedGet) the advanced option and parameter settings that affect the performance of this Solver engine.

All of these options and parameters may also be set via the object-oriented API, as described above under "Accessing Object Properties." Engine.Param names are listed for each option in the chapter "Solver Options;" these names normally match the named arguments listed below for SolverXPRESSAdvancedOptions.

SolverXPRESSAdvancedGet

Returns advanced XPRESS Solver option settings for the current Solver problem on the specified sheet. These settings are entered in the "back row" of tabs in the XPRESS Solver Options tabbed dialog box.

VBA Syntax

SolverXPRESSAdvancedGet (TypeNum:=, SheetName:=)

TypeNum is a number specifying the type of information you want.

TypeNum	Returns
1	The Invert Frequency value
2	The Minimum Number of Iterations Between Inverts value
3	The Reduced Cost Tolerance (as a decimal number)
4	The Eta Elements Zero Tolerance (as a decimal number)
5	The Pivot Tolerance (as a decimal number)
6	The Relative Pivot Tolerance (as a decimal number)
7	The Pricing Candidate List Sizing (as a decimal number)

8	The Degradation Multiplication Factor – Option no longer
9	The Integer Feasibility Tolerance (as a decimal number)
10	The Cut Frequency value
11	The Default Pseudo Cost (as a decimal number)
12	The Maximum Depth Cut Generation value
13	A number corresponding to the Integer Preprocessing group of options: –1 to indicate that the preprocessing options should be chosen automatically, 0 for no preprocessing, or the sum of one or more of the following: 1 = Reduced Cost Fixing at Each Node 2 = Logical Preprocessing at Each Node 4 = Probing at the Top Node
14	The Lifted Cover Inequalities at the Top Node value; -1 = Determine automatically
15	The Gomory Cuts at the Top Node value; -1 = Determine automatically
16	The Lifted Cover Inequalities in the Tree value
17	The Gomory Cuts in the Tree value
18	The (Barrier Method) Maximum Iterations value
19	The Minimal Step Size value
20	The Cholesky Decomposition Tolerance (as a decimal number)
21	The Primal Infeasibility Tolerance (as a decimal number)
22	The Dual Infeasibility Tolerance (as a decimal number)
23	The Column Density Factor value
24	The Maximum Memory value (integer number of megabytes); no longer used
25	A number corresponding to the Ordering Algorithm option: 0 = Choose ordering algorithm automatically 1 = Use Minimum Degree algorithm 2 = Use Minimum Local Fill algorithm 3 = Use Nested Dissection algorithm
26	A number corresponding to the Problem to Solve option: Always 0 = Choose problem to be solved automatically
27	TRUE if the CheckConvexity option is set to True; FALSE otherwise
28	A number corresponding to the Node Selection Criterion option: 1 = Select node based on the Forrest-Hirst-Tomlin criterion 2 = Select node with the best estimated objective value 3 = Select node with the best bound on objective value
29	A number corresponding to the Degradation Estimate dropdown list: Superseded by 32, returns the same value as 32 in V5.5E
30	The Strong Branching Global Entities value; -1 = Determine automatically
31	The Strong Branching Dual Iterations value; -1 = Determine automatically
32	A number corresponding to the Integer Estimates option: 1 = Sum minimum of up and down pseudo costs 2 = Sum all of the up and down pseudo costs

```
          3 = Sum max + 2 * min of up and down pseudo costs
          4 = Sum maximum of up and down pseudo costs
          5 = Sum the down pseudo costs
          6 = Sum the up pseudo costs

33        The Max Number of Indefinite Iterations (as a decimal number)
```

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.

SolverXPRESSAdvancedOptions

Equivalent to choosing Premium Solver... from the Add-Ins taband then choosing the Options button in the Solver Parameters dialog box when the XPRESS Solver is selected in the Solver Engines dropdown list. Specifies options that are entered in the "back row" of tabs in the XPRESS Solver Options tabbed dialog box.

VBA Syntax

SolverXPRESSAdvancedOptions (InvertFrequency:=, InvertMinimum:=, ReducedTol:=, EtaTol:=, PivotTol:=, RelPivotTol:=, PricingCand:=, DegradationFactor:=, IntegerFeasTol:=, CutFrequency:=, PseudoCost:=, CutDepth:=, IntPreProcessing:=, CoverCutsTop:=, GomoryCutsTop:=, CoverCutsTree:=, GomoryCutsTree:=, BarrierIterations:=, BarrierStepSize:=, CholeskyTol:=, PrimalFeasTol:=, DualFeasTol:=, ColumnDensity:=, BarrierIndefLimit:=, CholeskyOrder:=, CheckConvexity:=, BarrierProblem:=, MaxMemory:=, NodeCriterion:=, StrongBranchGlobal:=, StrongBranchDual:=, VarSelection:=, DegradationEstimate:=)

The arguments correspond to the "back row" of tabs in the XPRESS Solver Options tabbed 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.

Advanced LP Tab

InvertFrequency must be either -1 or an integer greater than zero. It corresponds to the Invert Frequency edit box.

InvertMinimum must be an integer greater than zero. It corresponds to the Minimum Number of Iterations Between Inverts edit box.

ReducedTol is a number between zero and one, corresponding to the Reduced Cost Tolerance edit box.

EtaTol is a number between zero and one, corresponding to the Eta Elements Zero Tolerance edit box.

PivotTol is a number between zero and one, corresponding to the Pivot Tolerance edit box.

RelPivotTol is a number between zero and one, corresponding to the Relative Pivot Tolerance edit box.

PricingCand is a number corresponding to the Pricing Candidate List Sizing edit box.

Advanced MIP Tab

DegradationFactor is a number corresponding to the Degradation Multiplication Factor edit box. This option is no longer used, but is included for backwards compatibility.

IntegerFeasTol is a number between zero and one corresponding to the Integer Feasibility Tolerance edit box.

CutFrequency must be an integer greater than zero. It corresponds to the Cut Frequency edit box.

PseudoCost is a number corresponding to the Default Pseudo Cost edit box.

CutDepth must be an integer greater than zero. It corresponds to the Maximum Depth Cut Generation edit box.

IntPreProcessing corresponds to the Integer Preprocessing group of options. This should be –1 to indicate that the preprocessing options should be chosen automatically, 0 for no preprocessing, or the sum of one or more of the following values:

Value	**Option specified**
1	Reduced Cost Fixing at Each Node
2	Logical Preprocessing at Each Node
4	Probing at the Top Node

CoverCutsTop must be an integer greater than or equal to –1. It corresponds to the Lifted Cover Inequalities at the Top Node edit box. (–1 indicates that the number of passes should be chosen automatically.)

GomoryCutsTop must be an integer greater than or equal to **-**1. It corresponds to the Gomory Cuts at the Top Node edit box. (–1 indicates that the number of passes should be chosen automatically.)

CoverCutsTree must be an integer greater than or equal to zero. It corresponds to the Lifted Cover Inequalities in the Tree edit box

GomoryCutsTree must be an integer greater than or equal to zero. It corresponds to the Gomory Cuts in the Tree edit box.

Advanced NB Tab

BarrierIterations must be an integer greater than zero. It corresponds to the Maximum Iterations edit box.

BarrierStepSize is a number between zero and one corresponding to the Minimal Step Size edit box.

CholeskyTol is a number between zero and one corresponding to the Cholesky Decomposition Tolerance edit box.

PrimalFeasTol is a number between zero and one corresponding to the Primal Infeasibility Tolerance edit box.

DualFeasTol is a number between zero and one corresponding to the Dual Infeasibility Tolerance edit box.

ColumnDensity must be an integer greater than or equal to zero. It corresponds to the Column Density Factor edit box. (Zero means the column density factor should be determined automatically.)

BarrierIndefLimit must be an integer greater than zero. It corresponds to the Max. Number Indefinite Iterations edit box.

CholeskyOrder corresponds to the Ordering Algorithm group of options:

CholeskyOrder	**Option specified**
0	Choose ordering algorithm automatically
1	Use Minimum Degree algorithm
2	Use Minimum Local Fill algorithm
3	Use Nested Dissection algorithm

CheckConvexity returns True or False and corresponds to the Check Convexity option.

BarrierProblem is no longer used, but is included for backwards compatibility.

MaxMemory is no longer used, but is included for backwards compatibility.

Advanced Node Selection Tab

NodeCriterion corresponds to the Node Selection Criterion group of options:

NodeCriterion	**Option specified**
1	Select node based on Forrest-Hirst-Tomlin criterion
2	Select node with best estimated objective value
3	Select node with best bound on objective value

StrongBranchGlobal is either –1, or the number of global entities on which to perform strong branching. (-1 means determine the number automatically.)

StrongBranchDual is either –1, or the number of dual Simplex iterations to use when performing strong branching. (-1 means determine the number automatically.)

VarSelection corresponds to the Integer Variable Estimates group of options:

VarSelection	**Option specified**
-1	Determine Automatically
1	Sum minimum of up and down pseudo costs
2	Sum all of the up and down pseudo costs
3	Sum max + 2 * min of up and down pseudo costs
4	Sum maximum of up and down pseudo costs
5	Sum the down pseudo costs
6	Sum the up pseudo costs

DegradationEstimate is no longer used, but is included for backwards compatibility.

Appendix: Solver Engine Methodologies

Introduction

This Appendix briefly describes the technical characteristics and methods used in the Solver Engines.

The Large-Scale SQP Solver Methodology

This section offers some insight into the methods used in the Large-Scale SQP Solver Engine, which is based on the Sparse Nonlinear Optimizer developed by Philip Gill, Walter Murray, and Michael Saunders, and the Evolutionary Solver developed by Frontline Systems. The LSSQP Solver uses a Sequential Quadratic Programming (SQP) method. Like other large-scale Solvers, it is designed to exploit *sparsity* in the problem, and it is particularly effective at exploiting partial linearity in an overall nonlinear problem.

An SQP method typically requires fewer major iterations or trial solutions than a Generalized Reduced Gradient (GRG) method. In both methods, each major iteration requires gradients of the problem functions. When computing gradients is relatively expensive – for example, if you use Solve With = No Action in the Analytic Solver Comprehensive and Analytic Solver Optimization, or if you don't define an Evaluator for derivatives in the Solver SDK – an SQP method that requires fewer major iterations is typically faster than a GRG method, even though it "does more work" at each major iteration – and this speed advantage grows with the size of the problem.

One disadvantage of an SQP method is that its trial solutions often violate the constraints until it is very close to the final solution, whereas a GRG method's trial solutions are typically feasible throughout most of the solution process. Hence, if the Solver must be stopped prior to finding an optimal solution, the GRG method's best trial solution found so far is more likely to be useful.

Sequential Quadratic Programming Method

At each major iteration or trial solution, an SQP method obtains an updated search direction by solving a quadratic programming (QP) subproblem. The objective of this QP subproblem is a quadratic approximation of a modified *Lagrangian* function that depends on the nonlinear problem's objective and constraints; the constraints of the QP subproblem are linearizations at the current point of the nonlinear problem's constraints.

The Large-Scale SQP Solver uses a smooth augmented Lagrangian merit function, and maintains a limited-memory quasi-Newton approximation to the Hessian of the

Lagrangian. Its QP subproblems are solved using a reduced-Hessian active-set method that allows for variables appearing linearly in the objective and the constraints. It uses “elastic programming” techniques to deal with infeasibility in the original problem and the QP subproblems; for infeasible models, it is more likely to arrive at a “close to feasible” solution than most other SQP solvers.

Performance on LP and QP Problems

The QP solver used in the Large-Scale SQP Solver uses a sparse LU decomposition of the matrix representing the linearized constraints. It is highly effective at solving moderately large quadratic programming (QP) problems, and very large-scale linear programming (LP) problems. It offers good performance on linear mixed-integer problems, but the Large-Scale LP/QP Solver and the XPRESS Solver are likely to offer better performance on these problems.

For more information on the Large-Scale SQP Solver’s methods, see Gill, Murray, and Saunders, SNOPT: An SQP algorithm for large-scale constrained optimization, Numerical Analysis Report 97-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997.

Evolutionary Solver Methods

The methods used by the Evolutionary Solver integrated with the Large-Scale SQP Solver are described in the Analytic Solver User Guide. When used to solve a non-smooth problem, the Evolutionary Solver acts as a global search method that uses the SQP Solver for its local searches. The global search uses genetic algorithm *mutation* and *crossover* operations, focusing on “nonsmoothly occurring” variables and variables that belong to “alldifferent” groups, to seek improved solutions. A new best point that passes certain tests (“distance and merit filters”) triggers a local search for further improvement.

The Artelys Knitro Solver Methodology

This section offers some insight into the methods used in the Knitro Solver Engine, which is based on the Knitro code developed by Richard Byrd, Jorge Nocedal and Richard Waltz, first at Northwestern University and now at Ziena Optimization, Inc. The Knitro Solver is a high performance implementation of new state-of-the-art interior point nonlinear methods, the result of intense research in large-scale nonlinear optimization in recent years. Starting in Version 8.0, the Knitro Solver also includes a high performance implementation of ‘scalable’ active set methods, similar in concept to the methods used in the Large-Scale SQP Solver, using a Sequential Linear Quadratic Programming (SLQP) approach.

Interior Point Method

The Knitro Solver’s interior point method (also known as a barrier method) solves a series of barrier subproblems, controlled by a barrier parameter. The algorithm uses trust regions and a merit function to promote convergence. It performs one or more minimization steps on each barrier problem, then decreases the barrier parameter, and repeats the process until the original problem has been solved to desired accuracy. Computational experience suggests that interior point methods are often superior to active set methods for problems with many degrees of freedom.

The Knitro Solver can use either of two procedures in the minimization steps. In the version known as Knitro-CG, each step is the sum of a normal step whose objective is to improve feasibility, and a tangential step that aims toward optimality. The tangential step is computed using a projected conjugate gradient iteration. The version known as Knitro-Direct always attempts to compute a new iterate by solving the primal-dual KKT system using direct linear algebra. In the case when this step cannot be guaranteed to be of good quality, or if negative curvature is detected, then the new iterate is computed by the Knitro-CG procedure.

Interior point methods generally require second derivative information (the Hessian of the Lagrangian of the objective and constraints) at each major iteration. The Knitro Solver has several alternatives for computing derivatives: It can obtain analytic Hessians from the Polymorphic Spreadsheet Interpreter via automatic differentiation; it can obtain analytic first derivatives via automatic differentiation, and use these to construct a Hessian approximation using a quasi-Newton (BFGS) or limited-memory quasi-Newton approach; or it can use analytic or estimated first derivatives to compute approximations of the Hessian-vector products used by the interior point method.

Active Set Method

The Knitro Solver's active set method is best described as a Sequential Linear Quadratic Programming (SLQP or SLP-EQP) method. In the first stage of its subproblems, a linear programming problem is solved to estimate the active set at the solution. This problem is obtained by making a linear approximation to the L1 penalty function inside a trust region. In the second stage, an equality constrained quadratic program (EQP) is solved involving only those constraints that are active at the solution of the first problem. The EQP incorporates a trust-region constraint and is solved (inexactly) by means of a projected conjugate gradient method.

The active set method often outperforms the interior point method on highly constrained problems. It is normally the better choice for mixed-integer nonlinear problems, since the Branch & Bound method will typically create many subproblems that are highly constrained or even infeasible as it tightens bounds on the variables.

Note: The KNITRO Solver does not support semi-continuous variables.

For more information on the Knitro Solver's interior point methods, see Byrd, Gilbert, and Nocedal, A trust region method based on interior point techniques for nonlinear programming, *Mathematical Programming* 89(1): 149-185, 2000, and Byrd, Nocedal, and Waltz, Feasible interior methods using slacks for nonlinear optimization, *Computational Optimization and Applications*, 26(1): 35–61, 2003. For more information on the Knitro Solver's active set methods, see Byrd, Gould, Nocedal, and Waltz, An algorithm for nonlinear optimization using linear programming and equality constrained subproblems, *Mathematical Programming*, Series B, 100(1): 27–48, 2004.

The MOSEK Solver Methodology

This section offers some insight into the methods used in the MOSEK Solver Engine, which is based on the code developed by Erling Andersen at MOSEK ApS in Denmark. The MOSEK Solver includes high performance implementations of both primal and dual Simplex methods nd interior point methods, plus a sophisticated Presolver and an automatic "dualizer." The MOSEK Solver is capable of handling problems with quadratic and second order cone constraints.

Interior Point Method

The MOSEK Solver's interior point (barrier) method is based on the *homogeneous self-dual method*, which does not require a feasible starting point. It internally solves a problem – constructed from the primal and dual forms of the original problem – that is *homogeneous* (all constraint right hand sides are zero, except for one "normalizing" constraint) and *self-dual* (the primal and dual forms of the internal problem are equivalent). Hence, the linear system that is solved on each major iteration has the dimension of the original problem plus one. For linear, quadratic, and second order cone problems, the barrier function can be evaluated efficiently, without requiring re-evaluation of the problem functions at each trial point.

For more information about the MOSEK Solver's interior point method, see Andersen and Ye, A computational study of the homogeneous algorithm for large-scale convex optimization, *Computational Optimization and Applications*, 10:243-269, 1998, and Andersen and Andersen, The MOSEK interior point optimizer for linear programming: an implementation of the homogeneous algorithm, *High Performance Optimization* (Kluwer), 197-232, 2000.

Handling of Quadratic Functions

The MOSEK Solver's interior point method directly solves a second order cone programming (SOCP) problem, which is a linear programming (LP) problem plus one or more second order cone (SOC) constraints. Each SOC constraint specifies that a group of *d* decision variables belongs to the second order cone of dimension *d*.

The MOSEK Solver Engine also handles problems with convex quadratic objectives and constraints. It internally transforms such quadratic functions into an equivalent set of decision variables and linear and SOC constraints.

The Gurobi Solver Methodology

This section briefly summarizes the linear programming, quadratic programming and integer programming methods used by the Gurobi Solver.

Primal/Dual Simplex Methods and Barrier Solver

The Gurobi Solver implements state of the art versions of the primal and dual Simplex methods, with advanced strategies for matrix updating and refactorization, multiple and partial pricing and pivoting and overcoming dependencies. In addition, it also includes a parallel barrier solver. The Gurobi Solver is capable of solving problems with millions of variables and constraints with very high speed and consistency.

Branch and Bound

The Gurobi Solver Engine uses an integrated and highly tuned Branch and Cut strategy, with a variety of node selection and branch variable selection strategies. It was designed to take maximum advantage of multi-core processors by parallelizing the Branch and Bound search. It supports the alldifferent constraint by generating an equivalent matrix of 0-1 variables and incorporating these into the problem and takes advantage of strong branching techniques.

Cutting Planes and Solution Heuristics

For MILP and MIQP problems the Gurobi Solver uses the latest methods including cutting planes and powerful solution heuristics.

Parallel Algorithms for Multi-core Computers

All models also benefit from advanced presolve methods to simplify models and slash solve times. In addition, the Gurobi Solver is able to simultaneously exploit any number of processors and cores per processor. New internal algorithms for load balancing mean that Gurobi performance scales better as the number of processors increases. In addition, the implementation is deterministic, so that two separate runs on the same model will produce identical solution paths.

Performance: New Nonlinear Solver Power

The Gurobi engine in Analytic Solver V2025 Q1 provides exceptional support for mixed-integer nonlinear optimization problems, including those that are often non-convex. These types of problems are common in industries such as specialty chemicals, hydroelectric power, and certain aeronautics and space applications. Unlike other nonlinear solvers (such as LSGRG, LSSQP, and KNITRO), Gurobi employs a different approach that can deliver globally optimal solutions—provided your model is formulated using specific algebraic expressions involving a limited set of mathematical operators (plus., addition, subtraction, multiplication, division, square, square root, exponential, logarithmic, etc.).

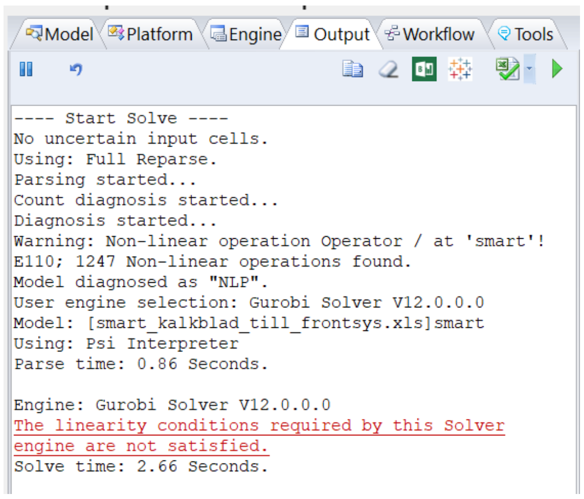
You can create your model in Excel, utilizing any of Excel's built-in mathematical operators and its 470+ functions. Analytic Solver's PSI Interpreter will analyze your formulas (including those dependent on other formulas at any level), diagnose your model as linear, quadratic, smooth nonlinear, or non-smooth, and—starting with V2025 Q1—inform you if your model is "nonlinear and solvable via the new Gurobi Solver." Analytic Solver can also generate the internal "expression trees" required by the Gurobi Solver.

Internally, Analytic Solver will expand your Excel functions into chains of "Gurobi-supported primitive operators" based on their mathematical definitions. For example, if you use the STDEV function in Excel, Analytic Solver will recognize it as "Gurobi-eligible" and create an expression tree internally using square, plus, and square root operators based on the definition of "standard deviation." This allows you to use the Gurobi Solver to solve a much broader range of nonlinear models more quickly and easily than other methods. A few functions, however, are not supported: ACOS, ASIN, ATAN and ACOT.

For "eligible" nonlinear optimization models the Gurobi 12.0 Solver is often faster than other nonlinear solvers, particularly when integer variables are involved, and when a globally optimal solution is required.

Note: This new functionality does not support array formulas in Excel, such as formulas where you select a column range and enter a formula that computes an array (not a scalar). In other words, it does not support functions that "spill." Instead, you will need to calculate each element of the array in its own individual cell.

If the Gurobi Solver is unable to solve a model, the reason will be provided in the Output tab of the Solver Task Pane.



```
---- Start Solve ----
No uncertain input cells.
Using: Full Reparse.
Parsing started...
Count diagnosis started...
Diagnosis started...
Warning: Non-linear operation Operator / at 'smart'!
E110; 1247 Non-linear operations found.
Model diagnosed as "NLP".
User engine selection: Gurobi Solver V12.0.0.0
Model: [smart_kalkblad_till_frontsys.xls]smart
Using: Psi Interpreter
Parse time: 0.86 Seconds.

Engine: Gurobi Solver V12.0.0.0
The linearity conditions required by this Solver
engine are not satisfied.
Solve time: 2.66 Seconds.
```

The XPRESS Solver Methodology

This section briefly summarizes the linear programming, quadratic programming and integer programming methods used by the XPRESS Solver, which is based on the Xpress^{MP} mixed-integer linear optimizer from FICO, Inc.

Primal and Dual Simplex Methods

The XPRESS Solver implements a highly advanced version of the primal and dual Simplex methods for linear programming, capable of solving problems with millions of variables and constraints with very high speed and reliability. Each step in these methods – scaling, presolving, finding an initial basis, basis matrix factorization, basis updates, selection strategies for entering and leaving variables, and more – has been tested and tuned for performance on many challenging LP problems.

Newton Barrier / Interior Point Method

The XPRESS Solver also implements a highly advanced version of the Newton Barrier method, also known as the interior point method for linear and quadratic programming, using advanced linear algebra methods to solve problems with millions of variables and constraints, again with very high speed and reliability.

Branch and Cut Methods

The XPRESS Solver excels at solving mixed-integer programming problems. It implements a wide range of strategies in an overall "Branch and Cut" framework that combines cutting plane methods with a Branch and Bound search. Each step in this framework – preprocessing and probing, cut generation, next node selection, and next branching variable selection – includes a comprehensive set of strategies. These strategies may be tuned by the user, but the XPRESS Solver is also very good at automatically choosing appropriate strategies for a specific problem.

Heuristic Methods

The XPRESS Solver also implements a wide array of heuristic methods, used within its Branch and Cut framework to quickly find good (feasible) integer solutions. Such integer solutions become "incumbents" that allow the XPRESS Solver to rapidly prune the search at other nodes in the Branch and Bound tree. Like other strategies, heuristics may be user controlled or chosen automatically. Although heuristics are never *guaranteed* to speed up the solution process, the heuristics used in the XPRESS Solver have been tested on a very wide range of real user problems and found to improve performance in many cases.

The OptQuest Solver Methodology

This section offers some insight into the procedures implemented in the OptQuest Solver engine that make possible the optimization of complex problems that cannot be easily described with a linear or smooth nonlinear objective and constraints.

Meta-heuristics

As an alternative to classical optimization methods, *heuristic* methods can provide approximate solutions to complex problems. For example, a production heuristic might give priority to jobs with the shortest estimated processing time. Depending on the context, this heuristic might work fairly well. However, in some other situations the results might be very poor. *Meta-heuristics* arose with the goal of providing something better. The aspiration was to integrate intelligent procedures and fast computer implementations to find "high-quality" solutions.

How OptQuest Uses Meta-heuristics

The OptQuest Solver is a generic optimizer for problems represented by Excel formulas, or a user-written Evaluator in Matlab, Java or another language. The disadvantage of this "black box" approach is that the optimization procedure is generic and does not know anything about what goes on inside of the model. The clear advantage is that the same optimizer can be used for many different models.

The optimization procedure performs a special "non-monotonic search," where the successively generated inputs produce varying evaluations, not all of them improving, but which over time provide a highly efficient trajectory to the best solutions. The process continues until it reaches some termination criterion (usually a time limit).

Scatter Search

Two of the best-known meta-heuristics are genetic algorithms and tabu search. Genetic Algorithm (GA) procedures were developed by John Holland in the early 1970s at the University of Michigan. Parallel to the development of GAs, Fred Glover of OptTek Systems established the principles and operational rules for tabu search (TS) and a related methodology known as scatter search.

Scatter search operates on a set of points, called reference points, that result in good solutions. The approach systematically generates linear combinations of the reference points to create new points, each of which maps into an associated point that yields integer values for discrete variables. Tabu search is then superimposed to control the composition of reference points at each stage.

Tabu Search has its roots in the field of Artificial Intelligence. Memory is a fundamental concept in Tabu Search, which uses search history to guide the process. In its simplest form, memory prohibits the search from reinvestigating solutions that have already been evaluated. However, the use of memory in the OptQuest Solver is much more complex and uses memory functions to encourage search diversification and intensification. These memory components let the search escape from locally optimal solutions to find a globally optimal solution.

Unlike genetic algorithm methods, scatter search makes only limited use of randomization when making choices among alternatives. As in probabilistic tabu search, the approach incorporates strategic probabilistic biases, taking account of evaluations and history. Scatter search focuses on generating relevant outcomes while still producing diverse solutions, due to the way the generation process (rounded linear combinations) is implemented. In particular, scatter search considers that the generation of new points might contain information that is not contained in the original points.

How OptQuest Uses Scatter Search

Scatter search is an information-driven approach, exploiting knowledge derived from the search space, high-quality solutions found within the space, and trajectories through the space over time. The combination of these factors creates a highly effective solution process, giving the OptQuest Solver the ability to solve challenging nonsmooth optimization problems.

Tabu search background can be found in a book by Fred Glover and Manuel Laguna *Tabu Search*, Kluwer 1997 (ISBN 0-7923-9965-X). Scatter search background can be found in a book by Manuel Laguna and Rafael Marti *Scatter Search: Methodology and Implementations in C,* Kluwer 2003 (ISBN 1-4020-7376-3).

Index