Analytica Optimizer Guide

Release 4.0

Lumina Decision Systems, Inc.
26010 Highland Way
Los Gatos, CA 95033
Phone: (650) 212-1212
Fax: (650) 240-2230
Web Site: www.lumina.com

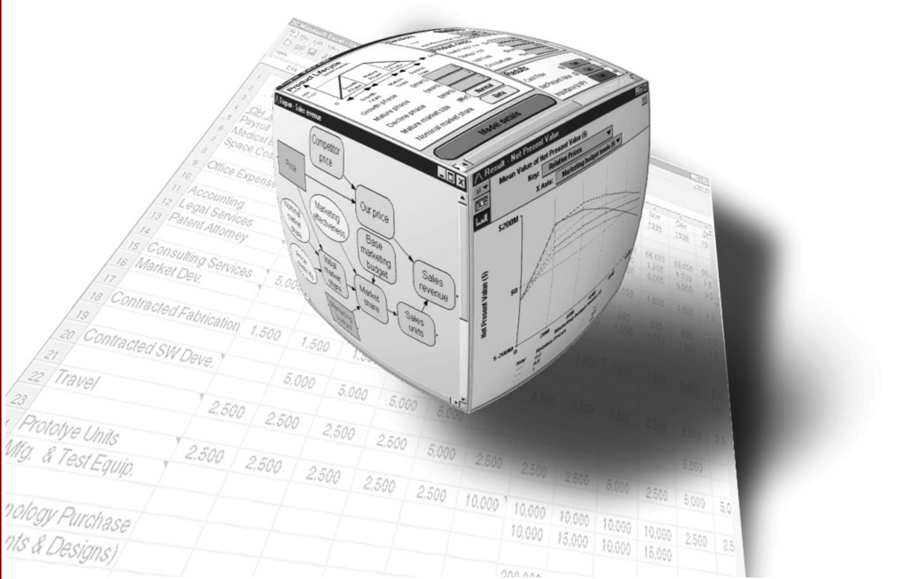
Copyright Notice

Content

Chapter 1 *Introducing the Analytica Optimizer*

This chapter shows you:

- What is the Analytica Optimizer
- How to obtain the Analytica Optimizer
- How to activate the Analytica Optimizer
- How to activate the Analytica Optimizer for ADE
- How to activate add-on engines



Introducing the Analytica Optimizer

This *Optimizer Guide* explains how to use the Analytica Optimizer. The *Quick Start* chapter is a tutorial taking you through the key steps to create some simple example Analytica models that use linear and nonlinear optimization. The chapter on *Formulating an Optimization* helps you to formulate your model for optimizing, and to choose whether it requires linear programming (LP), quadratic programming (QP), and non-linear programming (NLP). The other chapters provide more details on each of these three types of optimization and their many options. The final chapter gives a concise reference for all the optimization functions.

What do I need to know?

This Guide, including the *Quick Start* chapter, assumes you have basic knowledge of building models and writing expressions in Analytica. If you do not, you might first work though the *Analytica Tutorial* and scan through the *Analytica User Guide*.

This Guide provides an introduction to the basic concepts of optimization, including linear, quadratic, and nonlinear programming. It is not, however, a complete textbook on optimization. You may find it useful, especially for more challenging applications, to consult one of the many good textbooks on optimization.

What is the Analytica Optimizer?

The Analytica Optimizer adds to Analytica powerful functions to find optimal decisions and to solve equations. An optimal decision strategy may maximize value, minimize costs, or any quantified objective. The optimization may be subject to a set of constraints. Analytica Optimizer offers linear programming (LP), quadratic programming (QP), and non-linear programming (NLP). LP requires linear objective functions and linear constraints. QP requires quadratic objective functions and linear or convex quadratic constraints. NLP handles general nonlinear objective and constraint functions. All three methods handle decision variables that are continuous, discrete (integer, boolean, grouped), or mixed.

The Analytica Optimizer uses the Premium Solver Platform licensed from Frontline Systems, Inc. Frontline is the world leader in spreadsheet optimization: It developed the optimizer/solvers in Microsoft Excel and other spreadsheets. Their Premium Solver is the leading add-on software for spreadsheet optimization, and incorporates state-of-the-art technologies. The LP and QP methods handle up to 8000 variables and 8000 constraints in addition to variable bounds (up to 2000 variables may be integer, and the limit is 2000 variables when quadratic constraints are present). The NLP methods offer hybrid methods using classical gradient-search and evolutionary (genetic) algorithms for smooth and discontinuous objective functions, with up to 500 decision variables and 250 constraints.

The Analytica Optimizer performs optimization under uncertainty to maximize expected values, minimize loss percentiles, and other statistical functions of objectives and constraints. The LP and QP methods fully support Analytica's Intelligent Arrays: Thus, you can easily create arrays of optimizations conditioned on samples from uncertain variables, for parametric analysis of effects of key assumptions, and for each time period in a dynamic model. The nonlinear programming (NLP) functions do not fully support Intelligent Arrays. But, you can optimize nonlinear objectives that aggregate over dimensions — e.g. expected net present value to aggregate over uncertainty and time, and you can manually configure an optimization problem to abstract over explicitly named dimensions.

The Analytica Optimizer is an Analytica edition that includes all the functionality of the Analytica Enterprise edition. After developing optimizer-based models with Enterprise, you can deliver them to end users on the desktop using Analytica Power Player with Optimizer, or via a Web-browser on a server computer using the Analytica Decision Engine (ADE) with an Optimizer license.

How do I obtain the Analytica Optimizer?

You can purchase a license for the Analytica Optimizer or the Analytica Power Player with Optimizer from Lumina Decision Systems. Or you can purchase an upgrade to Optimizer if you already have a license for Enterprise or Professional editions.

If your copy of Analytica is for release 3.1 or earlier, you will need to upgrade it to release 4.0 to obtain the newest Optimizer features as described in this manual. Substantial discounts are available if you have a maintenance agreement for Analytica 3.1 (included free for 12 months from purchase).

Use of Optimizer from ADE requires a special ADE license. ADE Optimizer licenses include limits on the number of concurrent ADE process instances, with licensing pricing based on the maximum number of concurrent instances allowed on a machine.

For more information, visit the Lumina web site:

http://www.lumina.com

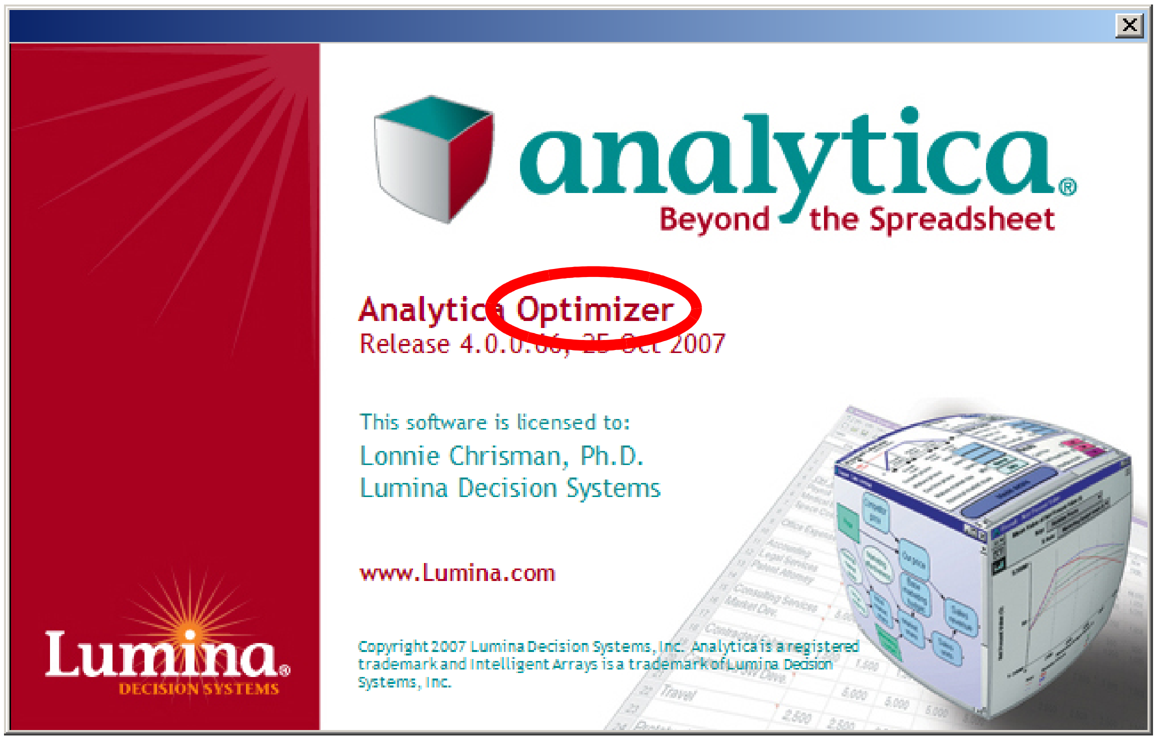
or call Lumina at 650-212-1212.

To Activate the Optimizer for Analytica

If you have already installed any edition of Analytica 4.0, your installation already includes the Optimizer files: There is no need to download new software. To activate the software to use the Optimizer, all you need to do is to enter into Analytica a new license code with the Optimizer option. Follow these steps:

1. Start up Analytica in the usual way, e.g. via the Windows ***Start*** menu, or by double-clicking on an Analytica model file.
2. From Analytica’s ***Help*** menu, select the ***Update license…*** option, to show the ***Analytica Licensing Information*** dialog box.
3. Replace the existing license code at the bottom of the dialog box with a new code that activates the optimizer. If you have received the new license code in an E-mail, you can copy and paste it directly into the dialog box.
4. Click **OK**.
5. Exit and restart Analytica.

You can verify successful activation of Analytica Optimizer by examining the splash screen when Analytica starts up, or by going to ***Help > About Analytica***. The splash screen should display "Analytica <edition> with Optimizer", like this:



Analytica Optimizer
Release 4.0.0.66, 25 Oct 2007

This software is licensed to:
Lonnie Chrisman, Ph.D.
Lumina Decision Systems

www.Lumina.com

Copyright 2007 Lumina Decision Systems, Inc. Analytica is a registered trademark and Intelligent Arrays is a trademark of Lumina Decision Systems, Inc.

To activate Analytica Optimizer for ADE

An Analytica Decision Engine (ADE) Optimizer edition is also available in release 4.0. An ADE Kit license includes a license code for Analytica Enterprise for developing models, as well as a code ADE for the production server. Similarly, ADE Optimizer includes a license code for Analytica Optimizer as well as a code for ADE Optimizer. See the preceding section on how to activate Analytica Enterprise with Optimizer. ADE Optimizer licenses include a maximum limit on the maximum number of concurrent ADE process instances that may be running concurrently on the same computer.

If you currently use ADE release 3.1 or earlier, you will first need to upgrade it to release 4.0.

To upgrade an existing non-Optimizer installation of ADE 4.0 to activate the optimizer, follow these steps to enter a new license code for ADE Optimizer:

1. Open a command prompt. From the ***Start*** menu, select ***Run***, type **`cmd`** and press *OK*.
2. Type: **`cd`** ***ADE_Dir,*** where ***`ADE_Dir`*** is the path to the directory for ADE 4.0.
   On most computers this will be: **`cd "c:\Program Files\Lumina\ADE 4.0"`**
3. Type: **`ade.exe /RegServer`**
   A dialog will appear that will allow you to enter your new license code.
4. Enter the new license code and press ***OK***.

Installing Optimizer add-on Engines

With Analytica Optimizer 4.0, it is possible to add on other engines for solving optimization problems. Some engines provide superior performance on particular classes of optimization problems, and some engines handle larger numbers of variables or constraints. Among the available add-on engines are: MOSEK, Large-scale SQP, XPRESS, KNITRO, OptQuest, large scale GRG, and large scale LP/QP. These add-on engines are available at additional cost and special installation steps.

To install an add-on engine in Analytica Optimizer, you must first receive the special license code and the DLL file containing the engine. With Analytica Optimizer already installed, place the DLL in your Analytica install directory (or, alternatively, remember the full file path to the DLL). Use RegEdit to add the following registry key/folder (if it does not already exist):

```
HKEY_LOCAL_MACHINE/Software/Lumina Decision Systems/Analytica/4.0/
SolverEngines
```

In this folder, create a String Value using the name of the add-on engine, and set its value to the full file path of your DLL. For example:

```
KNITRO : C:\Program Files\Lumina\Analytica 4.0\Knitro.dll
```

Next, find the Solver.lic file in your Analytica install directory and open it in a text editor such as NotePad. Add the license key provided for the add-on engine to this file.

To test for proper installation, open Analytica, and create a variable defined as follows:

```
Variable Engines := SolverInfo("AvailEngines")
```

Show the result for this variable and verify that your engine name appears in the resulting list.

What's new in Analytica Optimizer 4.0

Analytica Optimizer has several new features and has become easier to use since Analytica 3.1. Enhancements include:

- It has been upgraded to Frontline's SDK version 7.1 (from 4.5 formerly).
- There is now a capability to add-in other external solvers engines (at an additional cost), including XPress, Knitro, Mosek, OptQuest and Frontline large-scale engines.
- All optimizer algorithm/engine settings are now specified through two parameters, named ***parameter*** and ***setting***, to **`LpDefine()`**, **`QpDefine()`** and **`NlpDefine()`**. This scheme generalizes to the use of different engines, including new add-on engines.
- A new type of integer constraint, group-integers, is now supported. In this integer type, decision variables belonging to the same group are prohibited from having the same value.
- Quadratic constraints are now allowed in quadratic programs defined using **`QpDefine()`**. Special algorithms are highly effective in solving convex quadratic constraints, and second-order cone programs.
- A new function, **`SolverInfo()`**, provides access to the list of installed optimizer engines, the components of an optimization problem definition, and the attributes of an optimizer engine.

- When specifying non-linear optimization problems, almost all parameters to **`NlpDefine()`** are now optional, making it very simple to specify simple optimization problems. For example, the ***Vars*** and ***Constraints*** indexes can be omitted when there is only a single scalar decision variable, or zero or one constraints. For example, a simple unconstrained scalar optimization requires only two parameters.
- New ***SetContext*** and ***Over*** parameters to **`NlpDefine()`** provide more flexibility for structuring your model so that your optimization array abstracts properly. Also, **`NlpDefine()`** can be specified within a dynamic loop, such that the definition of an optimization problem at time t is based on the optimal solution of a previous optimization problem at time t-1.
- A new ***traceFile*** parameter to **`NlpDefine()`** makes it easy to log the points visited during an optimization search to a file for debugging.
- The **`ObjNL`** and **`LhsNl`** parameters to NlpDefine() allow quadratic dependence to be specified as a further hint.
- A ***MultiStart*** setting can be used with the "GRG nonlinear" engine in **`NlpDefine()`**, which is often quite effective when local minima are present.
- The nonlinear engines, such as "GRG Nonlinear" or "Evolutionary", can optionally be used to solve problems defined using **`LpDefine()`** or **`QpDefine()`**, as an alternative to the default "LP/Quadratic" and "SOCP Barrier" engines. With **`QpDefine()`**, this may be necessary if the quadratic constraints are non-convex.
- **`LpWrite()`**, **`LpRead()`**, and **`LpWriteIIS()`** now support three file formats: "LP", "MPS" and "LPFML". These formats are standards used for exchange of linear programs between other optimizer software products.
- The set of status codes returned from **`LpStatusNum()`** and the set of status messages returned from **`LpStatusText()`** have changed. Legacy models that tested against specific status numbers may need to be adjusted.

Chapter 2 *Quick Start*

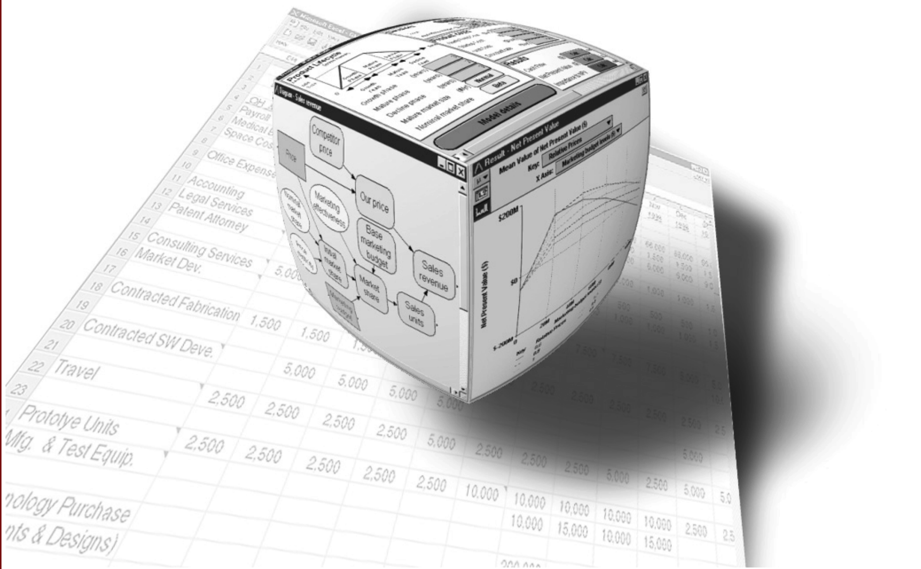
This chapter shows you:

How to browse Analytica Functions

How to obtain the Analytica Optimizer

How to optimize a linear program

How to optimize a non-linear program



Quick Start

Who this is for

This section leads you through a series of steps to create Analytica models that solve some simple linear and non-linear optimization problems. The reader should follow along by performing the steps in Analytica.

If you're already familiar with linear and non-linear programming …

If you are already familiar with concepts of linear, quadratic and non-linear programming, this provides a fast way to get started creating Analytica optimization models. Since this Quick Start chapter does not cover all the functions and features of Analytica Optimizer, and their use in complex situations, you should review the rest of the manual as well, especially "Optimization Function Reference" on page 94.

If you're not an expert already…

If you do not already have a previous background in linear and non-linear programming, performing the step-by-step instructions in this section may be the best place to start, even if you don't yet understand why each step is being done. Afterwards, read the remainder of this manual, returning to the examples in this section as you learn more about Analytica Optimizer. Also, be sure to explore the optimizer example models included with Analytica Optimizer.

Analytica prerequisites …

This manual, including this Quick Start section, assumes a basic knowledge of modeling and writing expressions in Analytica. If you do not yet have this background, you should go through the Analytica Tutorial and Users Guide prior to continuing with this manual.

Browsing Analytica Optimizer Functions

To begin, follow these steps:

1. Start Analytica in the usual way, e.g., using the menus:
   Start > Programs > Analytica 4.0 > Analytica 4.0.
2. In the main application menu, select ***Definition***.
3. Move your cursor down to the ***Optimizer*** submenu.

On the submenu that pops up, take a minute to scan the Analytica Optimizer function names. If you do not have an ***Optimizer*** option on your Definitions menu, it means that you do not have an Analytica Optimizer-activating License Code. You will need to contact Lumina at sales@lumina.com.

4. Select the diagram window and press CTRL-2 to create a new variable, and CTRL-E to edit its definition.
5. Select ***Paste Identifier…*** on the ***Definition*** menu.
6. Using the library pull-down, select ***Optimizer***.

From here you can review the optimizer functions along with parameters and function descriptions. The two main functions to study initially are **`LpDefine()`** (to define a linear program) and **`LpSolution()`** (to solve a linear program).

A Linear Program

This section will take you through the process of encoding a linear program in Analytica Optimizer. The model you create here is included in the **`Example Models/Optimizer Examples`** directory installed with Analytica under the name **`Two Mines.ANA`**. The problem you will encode is described as follows:

The Two Mines Company owns two different mines that produce an ore which, after being crushed, is graded into three classes: high, medium and low-grade. The company has contracted to provide a smelting plant with 12 tons of high-grade, 8 tons of medium-grade and 24 tons of low-grade ore per week. The two mines have different operating characteristics as detailed below.

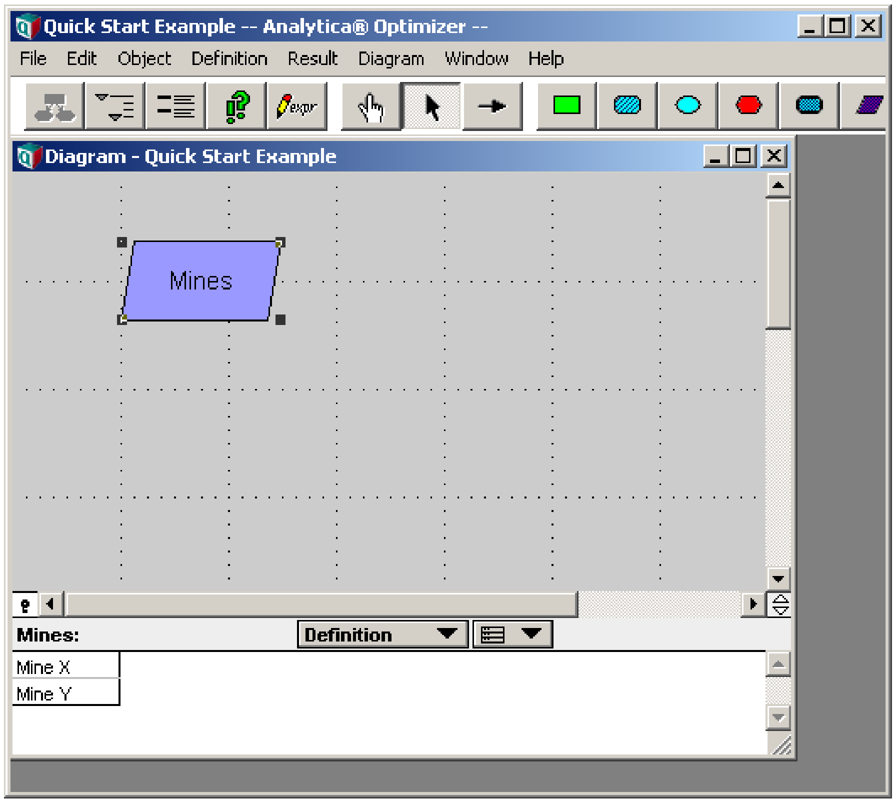
How many days per week should each mine be operated to fulfill the smelting plant contract?[1]

Mine	**Cost per Day ($1000)**	**Production (tons/day)**		
		High Grade	**Medium Grade**	**Low Grade**
X	180	6	3	4
Y	160	1	1	6

The first step is to identify the decision variables, in this case the number of days per week to operate each mine, and then create an index variable naming each decision variable:

1. Create an index name and name it *Mines*.
   We will use this as the index for the objective variables, i.e., the number of days per week to operate each mine.:
2. Edit its definition attribute and set its definition pull-down to ***List Of Labels***.
3. Enter the labels *Mine X* and *Mine Y*:

1. This example was created by J.E. Beasley.
Cf. http://www.brunel.ac.uk/depts/ma/research/jeb/or/contents.html

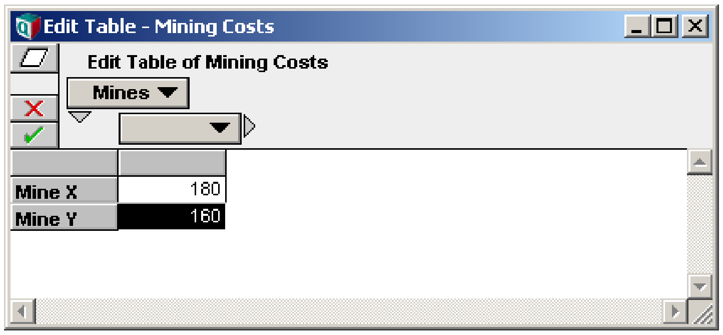

Quick Start Example -- Analytica® Optimizer --
File
Edit
Object
Definition
Result
Diagram
Window
Help
Diagram - Quick Start Example
Mines
Mines:
Definition

Mine X
Mine Y

Next, enter the mining costs, which will become the objective coefficients that define the objective as a linear function of the decision variables:

4. Create a variable and name it *Mining_Costs*. Set its units attribute to **`$K/day`**.

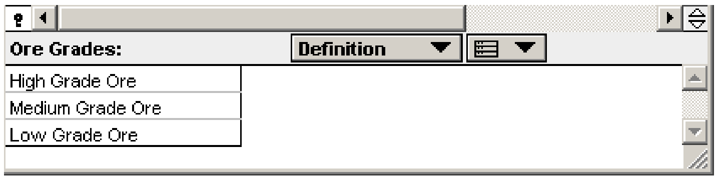
5. Edit its definition attribute and set the definition type to ***Table***. In the index chooser, select *Mines* and press ***OK***. Populate the table with the operating costs as follows:



Mine X	180
Mine Y	160

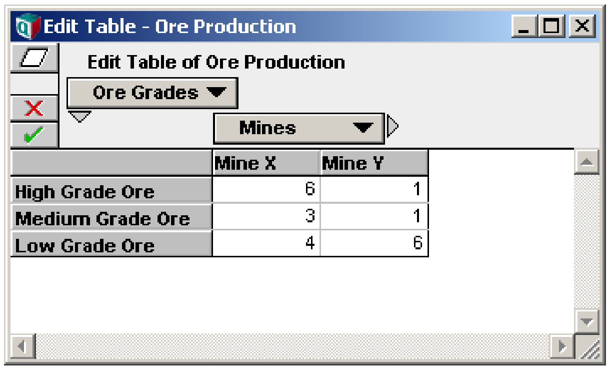
In this problem, there is one production constraint for each grade of ore. Thus, an index for ore-grade can serve as the constraint index:

6. Create an Index variable and name it *Ore_Grades*. Set its definition to a list of labels, thus:



High Grade Ore
Medium Grade Ore
Low Grade Ore

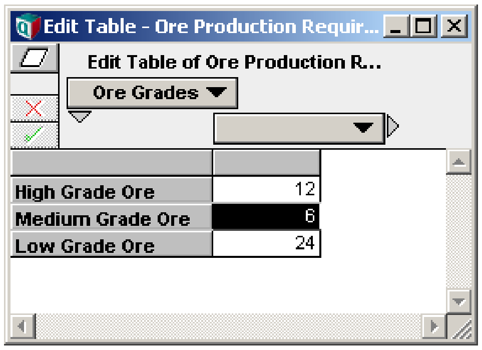
7. Create a decision variable and name it *Ore_Production*. Set its Units to **`tons/day`**. Set its Definition type to ***Table*** and in the Index chooser select both *Mines* and *Ore_Grades*. Fill in the table thus:



	Mine X	Mine Y
High Grade Ore	6	1
Medium Grade Ore	3	1
Low Grade Ore	4	6

8. Create a variable and title it "Ore Production Requirements". For convenience, set its identifier to *Ore_Prod_req*. Set its Units to **`tons/week`**.

9. Edit the definition attribute for Ore Production Requirements and set the definition type to **`Table`**, selecting *Ore_grades* as its index. Fill in the Edit Table thus:


Edit Table - Ore Production Requir...
Edit Table of Ore Production R...
Ore Grades

High Grade Ore	12
Medium Grade Ore	6
Low Grade Ore	24

Note that the constraints for the problem are, for each ore grade:

```
Sum(Ore_production*x, Mines) >= Ore_prod_req
```

where **`x`** is the objective, i.e., the number of days per week to operate each mine.

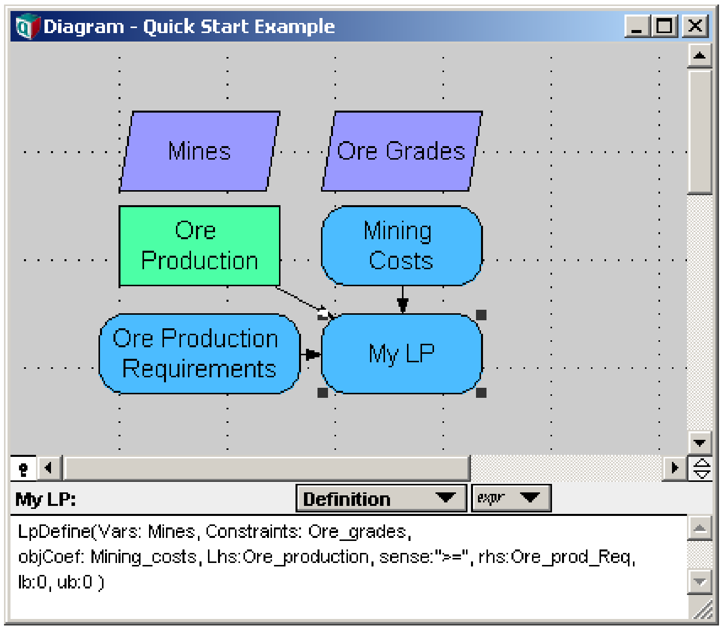
We now have all the inputs required to define the linear program.

To create the linear program to solve this problem:

10. Create a variable and name it *My_LP*. Enter the following definition:

```
LpDefine(Vars: Mines,
constraints: Ore_grades,
objCoef: Mining_costs,
lhs: Ore_production,
sense: ">",
rhs: Ore_prod_req,
lb: 0,
ub: 5)
```

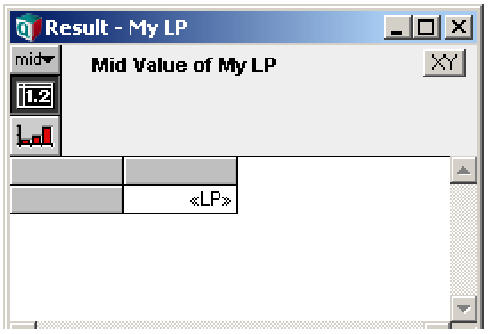
The parameters ***lhs***, ***sense***, and ***rhs*** refer to the left hand side of the constraint equations, the constraint equation comparator (greater than, equals, less than), and the right hand side of the constraint equations, respectively. The last two parameters, ***lb*** and ***ub*** (the lower and upper bounds), specify the limits on the number of days per work week that a mine can operate.



```
LpDefine(Vars: Mines, Constraints: Ore_grades,
objCoef: Mining_costs, Lhs:Ore_production, sense:">=", rhs:Ore_prod_Req,
lb:0, ub:0 )
```

Note that the example above uses ***name-based calling syntax*** for the function **`LpDefine`**: You give each parameter by name, colon, and the expression to be passed, e.g. **`Vars: Mines.`** You can also use more conventional position-based syntax, but that is less comprehensible for functions like **`LpDefine`** with many parameters and options. (See "Name-based calling syntax" in Chapter 20 of the User Guide.)

11. Select the *My_LP* node and press *CTRL*-R to evaluate it.



The **`LpDefine()`** function defines the linear program and returns a special object which displays as **`<<LP>>`**; however, it does not solve for the optimal solution. To do that:

12. Create an objective node and title it "Days per Week to Operate Mines". Set its units attribute to **`Days per week`**, and set its definition to **`LpSolution(My_LP)`**

Your model should now look something like:

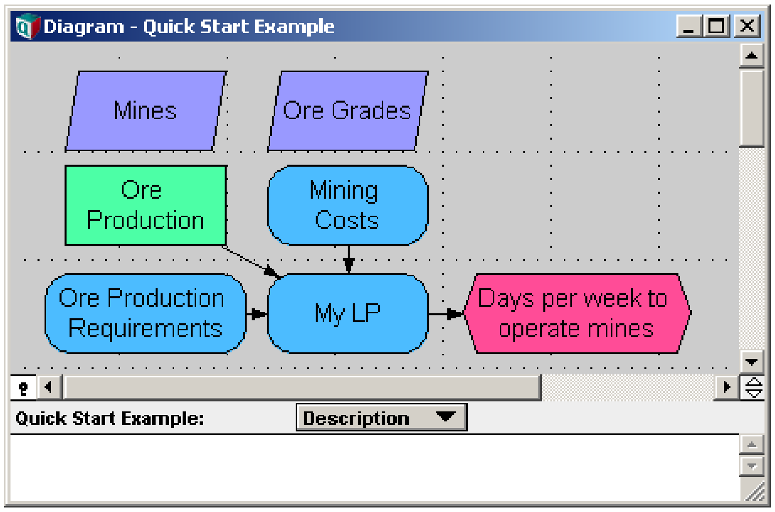
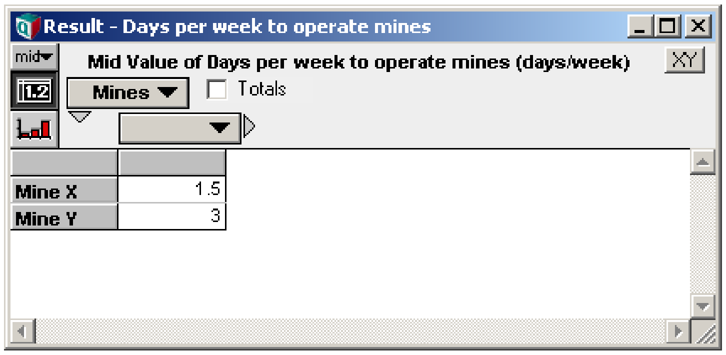


Diagram - Quick Start Example
Mines
Ore Grades
Ore Production
Mining Costs
Ore Production Requirements
My LP
Days per week to operate mines
Quick Start Example:
Description

13. Press *CTRL-R* to evaluate the linear program. The result view shows the optimal number of days per week to operate each mine:



Mid Value of Days per week to operate mines (days/week)

Mine X	1.5
Mine Y	3

It is always a good idea to check the status of the optimization as well. To check on the status of the optimization:

14. Create an objective variable and name it *Status*. Enter the definition: **`LpStatusText(My_LP)`**

15. Evaluate the variable *Status*.

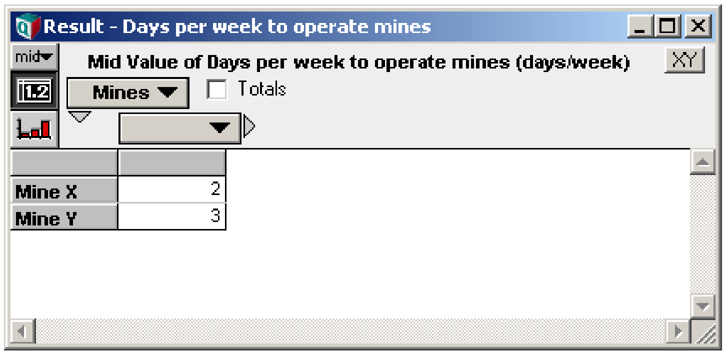
In this case, *Status* should be “Optimal solution has been found,” indicating that the solution viewed earlier was indeed the optimum. If the search had terminated early for some reason, or it could not find a feasible solution, *Status* would show you the situation. See “Obtaining the Solution” on page 26 for the full list of possible status values.

The example produced a non-integer solution. Suppose we needed an integer solution — because you could operate each mine only for an integral number of days, and partial days are not possible. You can easily modify the problem to achieve this:

Click on *My_LP* and change its definition by adding a ***ctype*** (Continuity type) parameter to indicate that you want an integer solution:

```
LpDefine(Vars: Mines,
constraints: Ore_grades,
objCoef: Mining_costs,
lhs: Ore_production,
sense: ">",
rhs: Ore_prod_req,
ctype: "I",
lb: 0,
ub: 5)
```

Click on ***Days per Week to Operate Mine*** and press CTRL-R to view the result.



Mid Value of Days per week to operate mines (days/week)

Mine X	2
Mine Y	3

A Non-linear Program

You will now define and solve a non-linear optimization. Non-linear optimizations are treated differently from linear and quadratic optimizations. In the previous linear programming example, the coefficient matrices completely describe the problem, and the optimum solution is simply computed. A non-linear optimization, by comparison, repeatedly re-evaluates expressions or portions of your model during a search. You will indicate the portion of your model to re-evaluate to the `NlpDefine()` function.

We will formulate the following optimization problem:

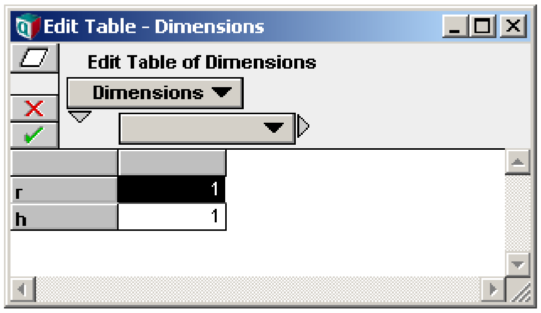
Find the dimensions of a cylinder with minimum surface area with a volume of at least 500 cm^3.

This example can be found in the **`Optimal can dimensions.ANA`** example model in the **`Example Models/Optimizer Examples`** directory installed with Analytica.

To model this, we first create a self-indexed table, *Dimensions*, to index the decision variables and to hold candidate solutions.

1. Start Analytica, or select ***File > New*** to start a new model.
2. Create a decision variable, name it *Dimensions*.

3. Set the definition type to **`Table`**, select **`Dimensions (Self)`** for the index, and fill in the edit table as follows:



r	1
h	1

Since it is self-indexed, the Dimensions variable serves both as the optimization vector and as the **`Vars`** index. During the optimization search, the cell values will be set to candidate solutions and other portions of the model evaluated.

For convenience, we can break out the decision variables as Analytica variables. To do that, follow these steps:

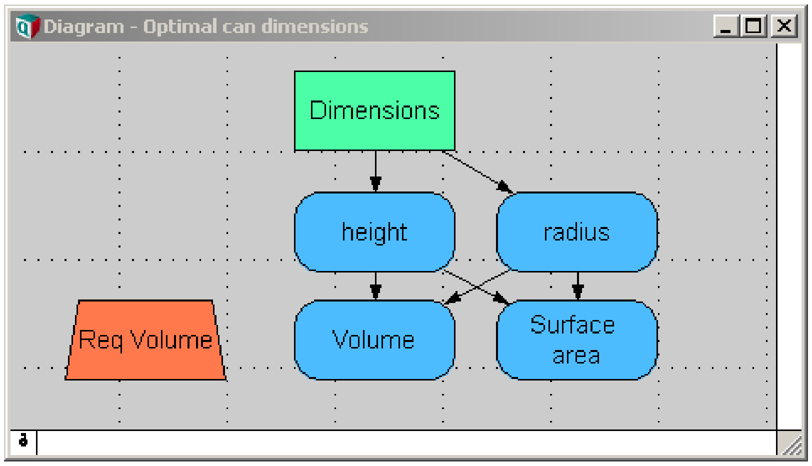
4. Create a variable node, named *Radius*. Give it the definition:
`Dimensions[Dimensions="r"]`
5. Create a variable, named *Height*. Give it the definition:
`Dimensions[Dimensions="h"]`

Next compute the Surface Area and Volume. *Surface_area* will become the objective function. Volume will become a constraint.

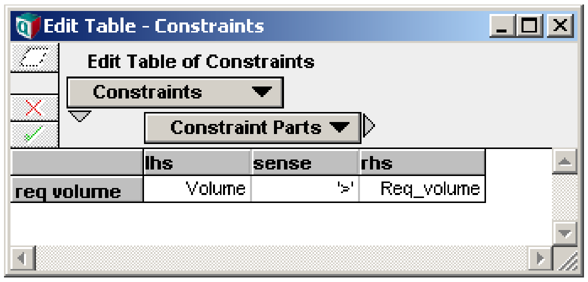
6. Create a variable named *Volume*. Give it the definition:
`height * Pi * radius^2`
7. Create a variable named *Surface_Area*. Give it the definition:
`2 * Pi * radius^2 + 2 * Pi * radius * height`
8. Create a constant named *Req_Volume*
(title: *Required Volume*). Set its value to **`500`**.

Next, set up the constraints, in this case there is only one. For non-linear problems, this involves setting up a constraint index, a left-hand side (which will be a computed expression) and a right-hand side. Sometimes it is convenient to do this as follows:

9. Create an index named *cp* with the title *Constraint Parts*. Define it as a list of labels:
`["lhs","sense","rhs"]`



10. Create a variable named *Constraints*. Set its definition to a table and select *Constraints (Self)* and *Constraint Parts* as the indexes. Set up the edit table so that *Constraint Parts* is on the horizontal dimension and *Constraints* is on the vertical dimension. Fill in the edit table as shown here:



	lhs	sense	rhs
req volume	Volume	'>'	Req_volume

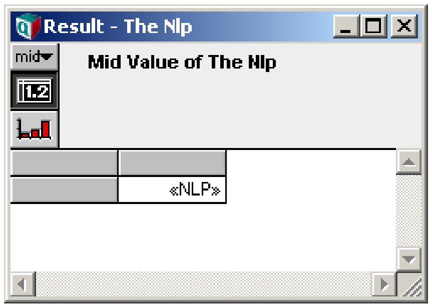
Now, define the non-linear optimization problem:

11. Create a variable named *The_NLP*. Give it the following definition:

```
NlpDefine(Dimensions, Constraints,
x: Dimensions,
obj: Surface_area,
lhs: Constraints[cp="lhs"],
sense: Constraints[cp="sense"],
rhs: Constraints[cp="rhs"])
```

This defines the non-linear optimization problem. The objective function is *Surface_area*, which is computed from the values in the *Dimensions* node. The left-hand side of the constraint is also computed from *Dimensions*.

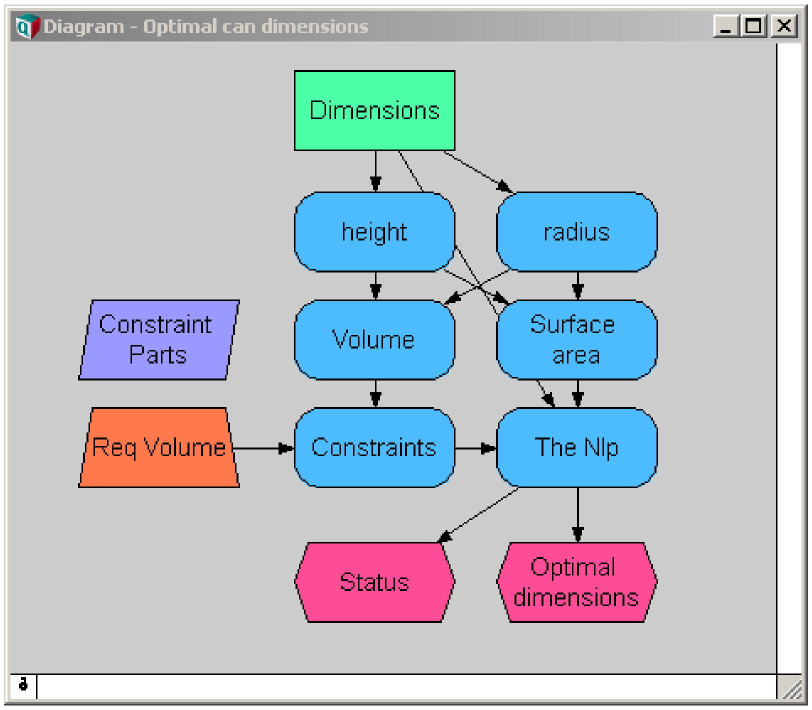
When *The_NLP* is evaluated (by selecting the node and entering *CTRL-R)*, an object is created that displays as **`<<NLP>>.`**



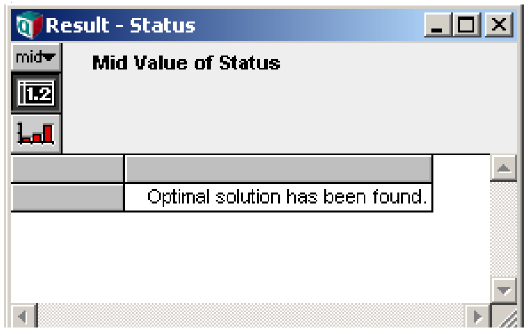
At that point, the NLP is not solved, it is only defined. It is solved when a function such as ***LpStatusText()*** or ***LpSolution()*** is evaluated. To get the solution:

12. Create an objective node named *Status*, and set its definition to:
`LpStatusText(The_NLP)`

13. Create an objective node named *Optimal_Dimensions* and set its definition to:
`LpSolution(The_NLP)`

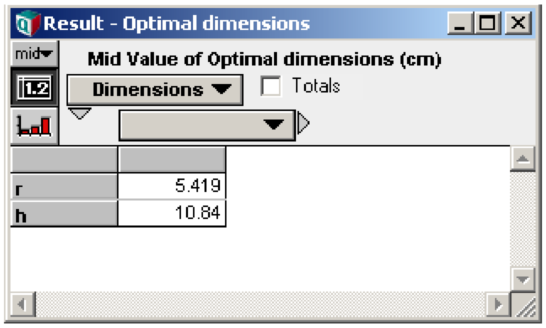


When either of these objective nodes is evaluated, the optimization engine will search for and report the optimal solution. View the *Status* node’s result to make sure the optimization was successful, and view the *Optimal_dimensions* node to view the solution and its status



Mid Value of Status

Optimal solution has been found.



Mid Value of Optimal dimensions (cm)

r	5.419
h	10.84

Chapter 3 *Formulating an Optimization Problem*

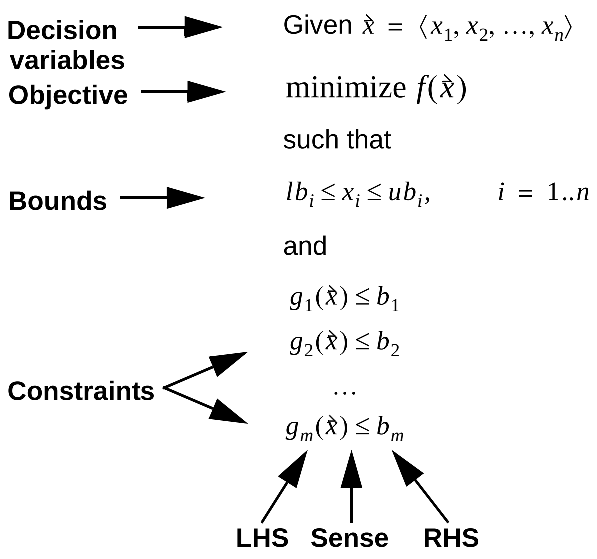
This chapter shows you:

- The different types of optimization problems
- How to choose the proper type of optimization
- How to optimize when solving simultaneous equations

Formulating an Optimization Problem

What are the parts of an optimization problem

The first step in performing an optimization is to formulate the problem appropriately. An optimization problem is defined by four parts: a set of decision variables, an objective function, bounds on the decision variables, and constraints. The formulation looks like this:



Given $\vec{x} = \langle x_1, x_2, \ldots, x_n \rangle$

minimize $f(\vec{x})$

such that

$$lb_i \le x_i \le ub_i, \qquad i = 1..n$$

and

$$g_1(\vec{x}) \le b_1$$
$$g_2(\vec{x}) \le b_2$$
$$\ldots$$
$$g_m(\vec{x}) \le b_m$$

Decision variables A vector (one dimensional array) $\vec{x} = \langle x_1, x_2, \ldots, x_n \rangle$ of the variables whose values we can change to find an optimal solution. A ***solution*** is a set of values assigned to these decision variables.

Objective A function $f(\vec{x})$ of the decision variables that gives a single number evaluating a solution. By default, the Optimizer tries to find the value of the decision variables that minimizes the value of objective. It will instead try to maximize the objective, if you set the optional parameter ***Maximize*** to true. For a linear program (LP), the Objective is defined by a set of coefficients or weights that apply to the decision variables. For a nonlinear program (NLP), the Objective can be any expression or variable that depends on the decision variables.

Bounds A range $lb_i \le x_i \le ub_i, \quad i = 1..n$ on the decision variables, defining what values are allowed. These bounds define the ***search space*** — that is the set of possible solutions. Each decision variable may have a lower bound and/or an upper bound. If not specified, the lower and upper bounds are **`-INF`** and **`+INF`** — that is, there are no bounds.

Constraints The constraints, e.g., $g_1(\vec{x}) \le b_1$, are bounds on functions of the decsion variables. They define which solutions are feasible.

Each constraint consists of a ***lefthand side (LHS)*** $g_1(\vec{x})$, which is a function of the decision variables, $\vec{x}$, a ***Sense***, (<, =, or >) defining the direction of the constraint, and a ***constant***, e.g b_1 .

Continuous, *integer*, and *mixed-integer* programs

Each decision variable may be specified as ***continuous***, meaning it is a real number (between bounds if specified), as ***integer***, meaning a whole number, as ***binary*** or ***Boolean,*** meaning its values may be True (1) or False (0), or as a member of an integer ***group***, where each member of the group must have a different integer value. Optimization problems are classified as ***continuous***, meaning the decision variables are all continuous, ***integer***, meaning they are all integer, binary, or group variables, or ***mixed-integer*** if they are a mixture of continuous and integer, binary or group variables. In this naming convention, binary or Boolean variables are treated as integer variables. The optimizer engine uses these distinctions to select which algorithms to use.

Choosing the type of optimization

A critical issue in formulating an optimization problem is determining whether it is linear, quadratic, or nonlinear. For a ***linear program (LP)***, the objective must be a linear function of the decision variables. For a ***quadratic program (QP)***, the objective and constraints must be linear or quadratic functions of the decision variables. The problem is a ***nonlinear program (NLP)*** if the objective or any of the constraints are nonlinear in any of the decision variables.

You define the type of a problem by using the function **`LpDefine()`**, **`QpDefine()`**, or **`NlpDefine()`**, respectively. You provide the decision variables, objective, bounds, and constraints as parameters to the selected function, along with some other parameters, which are optional.

Linear and convex quadratic optimization problems are often relatively fast to compute. But general nonlinear optimization is a computationally difficult problem. Many of the most famous and notoriously difficult computation problems can be cast as optimization programs, from the traveling salesman to the solution (or non-solution) of Fermat's last "theorem". It is, therefore, unreasonable to expect the Optimizer engine to succeed on any possible nonlinear problem you can formulate. While the Frontline Solver engine used in the Analytica Optimizer is among the best of the general purpose optimization engines available, success with hard optimization problems depends on your ability to formulate the problem effectively, provide appropriate hints for the Optimizer, and adjust the search control settings.

Linear and quadratic optimization in Analytica fully support Intelligent Arrays™ — that is, any of their parameters may be arrays with additional dimensions, and Analytica will perform an array of optimizations to compute an array of optimal values. For example, any parameter may be uncertain, defined as a random sample; and the optimization may be carried out within a dynamic loop, for each time step. In contrast, NLP is subject to restrictions on array abstraction, particularly in models with uncertain factors in the objective or constraints, or when used in dynamic loops. However, there are ways around these limitations, which we describe in "Array Abstraction" on page 48. However, it is easier to manage array abstraction, particularly in dynamic simulation, with linear or quadratic optimization problems.

There are often several ways to formulate the same optimization problem. The greater speed and flexibility of linear and quadratic formulations mean it is worth careful thought to see if it is possible to reformulate a nonlinear optimization into a linear or quadratic optimization. Often a simple transformation, combination, or disaggregation of the decision variables can turn an apparently nonlinear problem into a linear or quadratic problem.

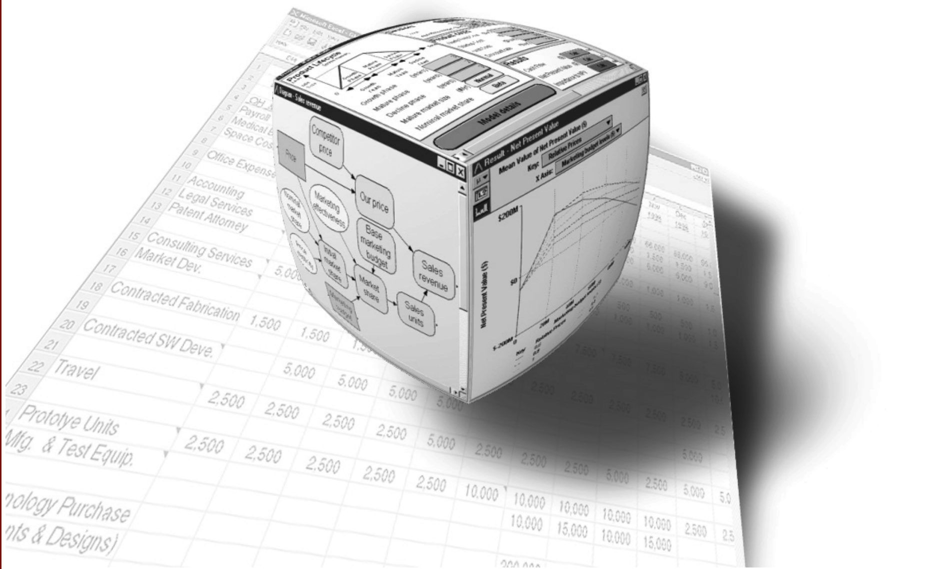
Solving *simultaneous* equations

The optimizer first attempts to find a feasible solution. If found, it then attempts to optimize within the set of feasible solutions. Thus, the solving a set of simultaneous equations is a special case of the optimization problem, where each constraint has a **sense** of "=", the objective is irrelevant (unless you want to express a preference among feasible solutions), and any feasible solution is a solution to the system of equations.

Chapter 4 *Linear Optimization*

This chapter shows you how to:

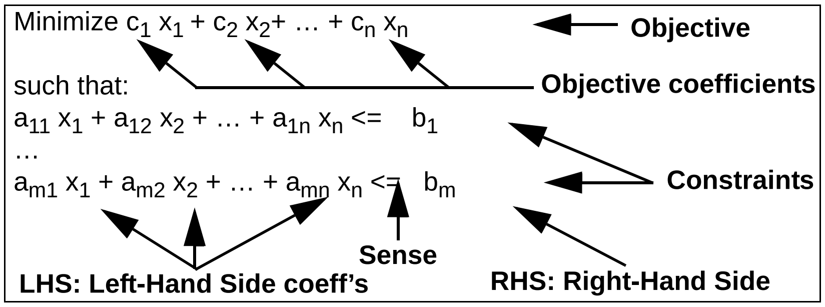
- Define a linear optimization problem
- Obtain a solution
- Deal with integer and binary decision variables
- Control the search



Linear Optimization

Defining a Linear Optimization Problem

A linear optimization problem has the following standard formulation:


Minimize $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$
Objective
such that:
Objective coefficients
$a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n <= b_1$
…
$a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n <= b_m$
Constraints
Sense
LHS: Left-Hand Side coeff's
RHS: Right-Hand Side

In this standard form, all decision variables, x_i, are real-valued and unconstrained, ranging from -INF to +INF ($-\infty$ to ∞).

To encode this in Analytica, use the function **`LpDefine()`**:

```
LpDefine(Vars, Constraints: Index;
        ObjCoef: Number[Vars];
        LHS: Number[Vars, Constraints];
        RHS: Number[Constraints] )
```

with these required parameters:

Vars	An index over the **n** ***Decision Variables***, [x_1, x_2, ... x_n], for which we wish to find the optimal ***solution*** — that is, the values that minimize (or maximize) the Objective. The index has one element for each decision variable. You may define it as a list of numbers, **`1..n`**, or a list of labels to give meaningful names to each Decision variable.
Constraints	An index over the set of **`m`** constraints, with one element for each constraint. Again, you may define it as a list of numbers, **`1..m`**, or a list of labels to give meaningful names to each Decision variable.
ObjCoef	The ***Objective Coefficients***, an array of n coefficients, [c_1, c_2, ... c_n], indexed by Vars. The objective we are trying to minimize (or maximize) is the dot product of these Objective Coefficients and the Decision variables — that is, $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$.
LHS	The Left-Hand Side of the constraints is an n by m array of coefficients, indexed by ***Vars*** and ***Constraints***, a_{11}, a_{12}, …a_{ij} ... a_{mn}].
RHS	The Right-Hand Side of the constraints, being an array of m constants, (b_1, b_2, ... b_m) indexed by ***Constraints***. The constraints, *e.g.*, $g_1(\hat{x}) \le b_1$, are bounds on functions of the decision variables. They define which solutions are acceptable.

Each constraint consists of a ***left-hand side (LHS)*** $g_1(\hat{x})$, which is a function of the decision variables, $\hat{x}$: a ***Sense***, (<, =, or >) defining the direction of the constraint, and a ***constant***, e.g b_1 .

When **`LpDefine()`** is evaluated, the result is a special linear program object, which displays as **`<<LP>>`**. This defines the linear program, but does not compute the optimum; that information is obtained through a series of functions described below under **Obtaining the Solution**.

Optional parameters

You can specify a wide set of optional parameters to **`LpDefine()`** for variations on the basic formulation shown above. These options include lower and/or upper bounds on the decision variables, maximizing instead of minimizing the objective, and changing the direction (***sense***) of the constraints from "<=" to ">=" or "=".

You can specify these optional parameters to **`LpDefine()`** in any order by listing each parameter name, followed by a colon, followed by the value. For example:

```
LpDefine(Vars: VarIndex, Lb: 0,
   Constraints: ConIndex, Lhs: lhs, Sense: ">=", Rhs: rhs,
   ObjCoef: ObjCoef, Maximize: True)
```

In this case, ***Vars***, ***Constraints***, ***Lhs***, ***Rhs***, and ***ObjCoef*** are the required indexes and coefficients as described in the previous section, and the other parameters are optional parameters, specifying that we want to ***`Maximize`*** the objective, each decision variable (x_1, …, x_n) has a lower bound (***`Lb`***) of zero, and all constraints have **`Sense`** ">=", instead of the default "<=".

Lower and upper Bounds on decision variables

You can specify lower and upper bounds on decision variables using the optional parameters:

```
Lb, Ub: Optional Number[Vars]
```

By default, ***Lb*** = -INF and ***Ub*** = +INF. If you give a single number to one of these parameters, it will specify the same bound for all decision variables. To specify a different bound for each decision variable, give it an array of values indexed by ***`Vars`***.

Lower and upper bounds for binary and grouped-integer variables are ignored, even if specified, since these are fixed. Binary variable bounds are 0-to-1, and grouped-integer bounds are 1..N, where N is the number of decision variables in the same group.

Maximizing the objective

The optional parameter **`Maximize`** should be either **`True`** or **`False`**, specifying whether Analytica Optimizer should attempt to maximize or minimize the objective function. If not specified, it defaults to **`False`**, and minimizes the objective function.

Sense of constraints

The ***sense*** of a constraint refers to whether the left-hand side is "<=", ">=", or "=" to the right-hand side. The **`Sense`** parameter:

```
Sense: Optional Text[Constraints]
```

is used to specify the sense for each constraint. When omitted, it assumes "<=" by default. The following text text values are recognized:

`"<", "<=", "L"` : LHS is less-than or equal to RHS
`">", ">=", "G"` : LHS is greater-than or equal to RHS
`"=", "E"` : LHS is equal to RHS

If a single value is passed to the sense parameter, that sense will apply to all constraints. If each constraint has a different sense, then the sense parameter should be an array indexed by constraints.

Obtaining the Solution

The optimal values for the decison variables, $x_1, \ldots, x_n$, are obtained using the **`LpSolution()`** function, which takes as a single parameter the **`<<LP>>`** object created by **`LpDefine()`**, and which returns an array indexed by the ***Vars*** index. The value of the objective function at the optimum is obtained using the **`LpOpt()`** function.

LpSolution(*lp: LpType*)

Returns the optimal solution to the programming problem ***lp*** defined by **`LpDefine()`**. The result is an array of decision variables indexed by *Vars.* If the Optimizer cannot find an optimal solution, it returns the best values found during the search so far.

LpOpt(*lp: LpType*)

Returns the value of the objective function for linear program ***lp*** at the optimum. For a linear problem, the value it returns is equal to:

```
Sum(LpSolution(lp) * ObjCoef, Vars)
```

LpStatusNum(*lp: LpType*) and LpStatusText(*lp: LpType*)

These two functions return, respectively, the status number and the corresponding text describing the status of the solution, or why the optimization search terminated, for the programming problem ***lp***. If the optimization is successful these results will be 0 and "Optimal solution has been found". If not successful **LpStatusNum** will return another number and **LpStatusText** will return different text explaining why it has not found an optimal solution.

Note: *The numeric codes from **LpStatusNum**, and the corresponding text from **LpStatusText** have changed in Analytica Optimizer 4.0. These reflect changes in the status numbers and text returned by the Frontline System's new, and restructured, optimizer.*

Tip Starting with version 4.0 of the Analytica Optimizer, you can use a different optimizer engine than the Frontline optimizer that comes with Analytica Optimizer. If a different optimizer engine is used, different engine-specific codes may be returned. In these cases, **LpStatusText** will return "unknown Frontline solver status code", and the result returned by **LpStatusNum** will depend on the status number returned by the optimizer engine. Consult the documentation for the engine you are using.

Possible outcomes to an optimization include:

1. It found a global optimum.
2. There is no feasible solution, because the constraints are contradictory.
3. The optimal solution is unbounded, because the constraints (if any) do not prevent the objective function from approaching $-\infty$ (for a minimization problem).
4. The search terminates with a feasible solution, but before an optimal solution is found. This happens when the computation time or number of pivots exceeds the termination criteria before a feasible solution has been located (see “Controlling The Search” on page 32).
5. The search terminates before finding a feasible solution.

These different cases can be detected using the **`LpStatusNum()`** or **`LpStatusText()`** functions, both of which take the ***LP*** as a single parameter, and which may return the following values for a continuous linear program:

Status Number	Status Text
-3	Invalid status.
-2	Ignore status. Used when dummy result code needs to be overridden.
-1	Invalid license status. (License expired, missing, invalid, *etc.)*
0	Optimal solution has been found.
1	The Solver has converged to the current solution.
2	"No remedies" status. (All remedies failed to find better point.)
3	Iterates limit reached. Indicates an early exit of the algorithm.
4	Optimizing an unbounded objective function.
5	Feasible solution could not be found.
6	Optimization aborted by user. Indicates an early exit of the algorithm.
7	Invalid linear model. Returned when a linearity assumption renders incorrect.
8	Bad data set status. Returned when a problem data set renders inconsistent.
9	Float error status. (Internal float error.)
10	Time out status. Returned when the maximum allowed time has been exceeded. Indicates an early exit of the algorithm.
11	Memory dearth status. Returned when the system cannot allocate enough memory to perform the optimization.
12	Interpretation error. (Parser, Diagnostics, or Executor error.)
13	Fatal API error. (API not responding.)
14	The Solver has found an integer solution within integer tolerance.
15	Branching and bounding node limit reached. Indicates an early exit of the algorithm.
16	Branching and bounding maximum number of incumbent points reached. Indicates an early exit of the algorithm.
17	Probable global optimum reached. Returned when MSL (Bayesian) global optimality test has been satisfied.
18	Missing bounds status. Returned for EV/MSL Require Bounds when bounds are missing.
19	Bounds conflict status. Indicates <= => = bounds conflict with existing binary or all different constraints.

Status Number	Status Text
20	Bounds inconsistency status. Returned when the lower bound value of a variable is grater than the upper bound value, i.e. lb[i] > ub[i] for some variable bound i.
21	Derivative error. Returned when API_Jacobian has not been able to compute gradients.
22	Cone overlap status. Returned when a variable appears in more than one cone.
999	Exception occurred status. Returned when an exception has been caught by try/catch top-level.
1000	Custom base status. (Base for Solver engine custom results.)
1102	The quadratic constraints are non-convex, the SOCP engine cannot solve this problem.

`LpSolution()` will often return the best solution "point" so far even in the cases in which the global optimum was not located, so it is important to check the status.

Secondary Aspects to Solution

The solution to a linear program contains more information that just the optimal solution, (x_1, …,x_n). Often these secondary elements of the solution are of more value than the solution itself for decision making purposes, since they indicate how changes (*e.g.*, different decisions) impact the optimum. These secondary aspects of the solution are accessed using the functions **`LpSlack()`**, **`LpObjSa()`**, **`LpRHSSa()`**, **`LpShadow()`**, and **`LpReducedCost()`**.

Slack or Surplus: LpSlack(*lp: LpType*)

When you have a constraint

$$a_{i1}\ x_1 + a_{i2}\ x_2 + \ldots + a_{1n}\ x_n <= b_i$$

the slack (or surplus) for that constraint is the positive value that, when added to the LHS, makes both sides equal, *i.e.*,

$$a_{i1}\ x_1 + a_{i2}\ x_2 + \ldots + a_{1n}\ x_n + slack_i = b_i$$

The constraints that have zero slack are of particular interest, since they are instrumental in constraining the optimum. If these constraints are relaxed (*e.g.*, by increasing b_i), a larger maximum value can be obtained. However, as critical constraints are relaxed, other constraints may become relevant. For the constraints the non-zero slack gives an indication of how close they are to becoming critical.

The slack for each constraint is obtained from the function:

```
LpSlack(Lp)
```

It takes as input the object returned from **`LpDefine()`** and returns an array indexed by ***Constraints***, containing the slack at the optimum for each constraint.

Coefficient Sensitivity: LpObjSa() & LpRhsSa()

If we change a coefficient in the objective function, the optimal solution (x_1, …,x_n) will continue to be the optimal solution as long as the coefficient remains within a certain range. Note that the solution point is the same, but the value of the objective function at the optimum is effected. This range can be computed with the function

```
LpObjSa(Lp: LpType; Var: optional)
```

The first parameter, ***Lp***, is a linear program defined using **`LpDefine()`**. When called with only a single parameter, the range is computed for all decision variables, and the result is indexed by the linear program variable array, ***Vars***. If the range for only a single decision variable (or a small subset) is required, the second parameter. ***Var***, is used to indicate the decision variable for which the sensitivity is to be computed. The second parameter should be an element (or a subset) of the ***Vars*** index.

The result returned from **`LpObjSa()`** is dimensioned by a local index, **`.range:= ['lower','upper']`**. Thus, to get the smallest value for each coefficient in the objective that would continue to produce the same solution, you would use an expression such as:

```
Var sa:= LpObjSa(myLp) DO
sa[.range='lower']
```

Note: *The* **`LpObjSa()`** *function can only be used with a linear-program. It is not meaningful for quadratic or non-linear programs.*

The sensitivity of the right-hand side coefficients can be computed using the function:

```
LpRHSSa(Lp: LpType; constraint: Optional)
```

This computes the range over which coefficient in the RHS can vary without changing the basis of the solution. In other words, over the returned range the set of constraints with zero slack remain the set of constraints with zero slack (*i.e.*, the critical constraints).

The result is indexed by a local index, **`.range:= ['lower', 'upper']`**, containing the smallest and largest values for the corresponding RHS coefficient. If the optional second parameter is not specified, the range is computed for all variables and the result is indexed by **`Vars`**. If the range is needed for only a single coefficient, the second parameter specifies an element of the ***Constraints*** index, and only the range for that constraint is computed.

When a coefficient can be changed an arbitrary amount without changing the solution basis, the corresponding entry in the result returned by **`LpRhsSa()`** or **`LpObjSa()`** will be **`-INF`** for the lower value or **`+INF`** for the upper value.

Dual Values: Shadow Prices and Reduced Costs

If a constraint is relaxed, *i.e.*, by increasing the right-hand side, bi, by one unit, how will this impact the objection function? This is referred to as the ***shadow price***, or ***dual value***, of the constraint. A shadow price is valid only for small changes in b_i (the actual range for which it is valid can be obtained from the **`LpRhsSa()`** function), and is computed by the function:

```
LpShadow(lp: LpType)
```

Where ***lp*** is a linear program object returned by **`LpDefine()`**. The result is indexed by ***Constraints***. Mathematically, the shadow price is given by

$$\text{Shadow}_i = \frac{\partial \text{ Obj}}{\partial b_i}$$

i.e., the partial derivative of the objective function relative to the constraint RHS coefficient.

Warning: *Not all linear programming packages use the same convention for the sign of shadow prices. If you have used the LINDO package, note that the convention used by Analytica Optimizer, differs from the sign produced by the LINDO package.*

How far can a coefficient in the objective function be increased (in a minimization program) or decreased (in a maximization program) before the objective function changes? When a decision variable has a non-zero value in the optimal solution, then any change in the objective function coefficient will change the objective value, so for those decision variables the answer would be zero. But for decision variables that are zero, the coefficient can change until that variable eventually enters the basis. This amount is known as the reduced cost (or dual value) of the variables and is returned by the function

`LpReducedCost(`*`lp: LpType`*`)`

The result is indexed by ***Vars***.

The Shadow Price and Reduced Cost are known as dual values, the Shadow Price being a dual to the solution in the original (or "primal") problem, and the Reduced Cost being a dual to the slack price in the original problem. To each problem in the standard form (see "Defining a Linear Optimization Problem" on page 24) there corresponds a dual linear program given by:

$$\text{maximize } b_1 y_1 + b_2 y_2 + \ldots + b_m y_m$$

such that

$$a_{11} y_1 + a_{21} y_2 + \ldots + a_{m1} y_m >= c_1$$

$$\ldots$$

$$a_{1n} y_1 + a_{2n} y_2 + \ldots + a_{mn} y_m >= c_n$$

The new variables in this program, $y_1, y_2, \ldots, y_m$, are the shadow prices, and the slack value for each constraint are the reduced costs in the primal problem. Note that the variables in the primal problem correspond to constraints in the dual problem, and constraints in the primal problem correspond to decision variables in the dual problem.

Examples

Several example linear-programming optimization models are included in the **`Example Models/Optimization Examples`** folder installed with Analytica. The linear program examples include:

- **`Automobile production.ANA`**: Taking differences in unit production cost, and labor and material availability into consideration, figure out how many cars to produce at each factory to meet a production goal. This example demonstrates the use of Linear Program-related sensitivity functions.
- **`Big Mac Attack.ANA`**: Optimize your McDonald's-based diet to fit your budget, nutritional needs, and minimize your calorie or carbohydrate consumption.
- **`Capital Investment.ANA`**: Simple case of selecting which projects to pursue given a fixed budget.
- **`Optimal production planning.ANA`**: A classic textbook linear program: Selecting how much of each product to produce given resource limitations.

- **`Production Planning LP.ANA`**: Another take on the same problem, but demonstrating the interpretation of the secondary solution aspects.
- **`Two Mines Model.ANA`**: Schedule production at multiple mines to meet production goals given capacity constraints. (This is the example used in Chapter 2, “Quick Start,”)
- **`Sudoku with Optimizer.ANA`**: Solves Sudoku puzzles. Demonstrates use of grouped integer variable types with many groups.

Integer, Binary, and Grouped Decision Variables

In a standard linear program the decision variables are assumed to be continuous (real-valued) numbers. However, you can also use Analytica Optimizer to define and optimize a linear program with some or all of the decision variables constrained to be integers, boolean or binary, grouped-integer, or a mixture of continuous and integer, binary and grouped-integer variables (a ***mixed integer program***).

You can specify the type of each decision variable as continuous, integer, binary or grouped using the optional parameter:

ctype**`: Optional Text[Vars]`**

which takes one of the following values:

- **`"C"`**: Continuous
- **`"I"`**: Integer
- **`"B"`**: Binary or Boolean value, i.e. 0 or 1
- **`"G"`**: Grouped integer, i.e., one of 1..N, where N is the number of decision variables in the same group, and such that no variables in a group have the same value.

If you give the ***ctype*** parameter a single text character, it specifies the same type for all decision variables, *e.g.*:

```
LpDefine(…, ctype: "B")
```

specifies that all decision variables are binary. To specify a ***mixed-integer program***, you supply an array of characters, indexed by ***Vars***, specifying the type of each decision variable.

When you have grouped-integer variables partitioned into two or more groups, the ***group*** parameter specifies which group each of the grouped-integer variables belongs to:

```
group : Optional Number[Vars]
```

For example, if your grouped-integer decision variables are partitioned into three groups, the elements of the array passed to **`group`** would be 1, 2, or 3 to indicate the group that each variable belongs to, and 0 for each of the non-grouped-integer decision variables. Each decision variable can belong to at most one group, and each group should have at least two grouped-integer decision variables. The example model **`"Sudoku with Optimizer.ana"`** demonstrates the use of multiple grouped-integer groups.

In general, Integer and mixed-integer linear programs are harder to solve than linear programs with exclusively continuous variables. The Optimizer uses a combination of a Simplex algorithm with a memory-efficient branch-and-bound algorithm.

In some cases, the Optimizer may fail to find a solution to a large integer or mixed-integer linear program. Use the **`LpStatusNum()`** and **`LpStatusText()`** functions to see whether it has been successful, and if not, why not. For a complete list of the possible values returned by **`LpStatusNum()`**, see “LpStatusNum(lp: LpType)” on page 97.

Controlling The Search

A linear program having all continuous decision variables is solved using a simplex algorithm. The space of feasible solutions is called a simplex and is a convex polyhedron in N-dimensional space, where N is the number of decision variables. A simplex algorithm traverses the simplex from corner to corner, moving to an adjacent corner with an improved objective value at each iteration (***pivot***). The objective is improved with each pivot until the global optimum is reached. The same algorithm is used on an augmented simplex initially to find an initial feasible solution.

An integer, binary, or mixed-integer program uses the simplex algorithm in combination with a branch-and-bound algorithm. It first uses the simplex to solve the continuous version of the problem. This bounds the optimal objective from one side and provides a starting point for a search. Whenever it finds a feasible integer solution, this provides a bound on the optimal objective on the other side and allows the branch-and-bound search to prune alternative integer solutions that would be provably inferior to the ones already found. As the algorithm explores solutions having one integer decision variable set to a particular integer value, the continuous LP sub-problem is solved again using repeated invocations of the simplex algorithm on increasingly constrained problems. It terminates the search when the search space has been exhausted (*i.e.*, the global optimum located), when the termination criteria has been exceeded, or when the best solution found is within the solution (gap) tolerance. In addition, logical implications of integer constraints can often be deduced before Simplex is even run. Various algorithms for finding these constraints are reformed to as Cuts, and various algorithms for recognizing and utilizing cuts of different types may be switched on or off.

Many settings controlling the precise behavior of the optimizer can be altered using the two parameters to **`LpDefine()`** named ***parameter*** and ***setting***. The list of all possible parameters is the topic of Chapter 7, “Control Settings”.

Viewing and specifying control settings

Once you have defined a linear program using **LpDefine( )**, the following function returns the set of control settings used by the engine:

```
SolverInfo("Setting", Lp: myLp )
```

where you replace ***myLp*** with the name of the variable holding the result from **LpDefine( )**.

You can also access the range of allowed values for each setting, as well as the default value, using **SolverInfo(**). For this, you need to know the name of the optimizer engine used on your problem. For linear programs, this will always be "LP/Quadratic" unless you have installed an add-on engine. To obtain the name of the engine used in the general case, use:

```
SolverInfo("Engine", Lp: myLp )
```

Using the name of the engine, the range (min/max) of possible values for each setting, and the default value, can be obtained using:

```
SolverInfo( ["MinSetting","MaxSetting","Defaults"],
    Engine: "LP/Quadratic")
```

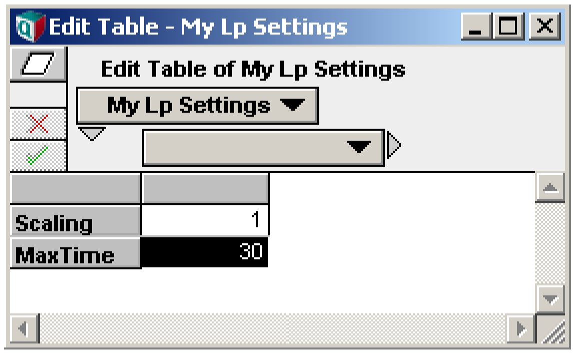
If you want to change the value for a single control setting, you can specify values for two optional parameters, ***parameter*** and ***setting***, to **LpDefine( )**, providing the name of the setting to ***parameter***, and the value to ***setting***. For example, if you wished to set the `Scaling` parameter to 1, you would modify your call to **LpDefine( )** as follows:

```
LpDefine( .., Parameter: "Scaling", Setting: 1 )
```

To alter more than one control setting, you need to supply arrays to these parameters. The arrays passed to ***parameter*** and ***setting*** should have a single common index. If the index of the array passed to ***setting*** is a list of labels, where the index labels contain the name of each control setting, then you only need to include the ***setting*** parameter.

It is often convenient to specify control settings in a self-indexed edit table. The following steps illustrate this:

1. Drag a variable node to your diagram, title it "*My Lp Settings*".
2. In the definition pane, set the definition type to **Table**.
3. In the Index Chooser dialog, select "My Lp Settings (Self)" as the table index.
4. Click on the row heading cell, and change "Item 1" to "Scaling".
5. With the row header still selected, press down-arrow to add a row.
6. Change the second row header cell to "MaxTime"
7. Enter 1 into the first table body cell.
8. Enter 30 into the second body table cell.



Scaling	1
MaxTime	30

9. In your call to **LpDefine( )**, insert a setting parameter as follows:

```
LpDefine( ..., setting: My_lp_settings )
```

The optimizer will scale parameters and terminate after 30 seconds if the optimum has not been found. A self-indexed table set up in this fashion makes it easy to adjust multiple control settings if the need arises.

Note: *In Analytica Optimizer 3.1, control settings were specified as optional parameters to* **LpDefine( )**. *These legacy parameters are still supported for backward compatibility; however, use of the* ***setting*** *parameter is recommended. This change reflects a change in Frontline's architecture, and more readily generalizes to other add-on engines and future optimizer engine extensions.*

Linear Programming Settings

The list and descriptions of all settings to Optimizer engines is covered in Chapter 7, “Control Settings”. Continuous linear problems will usually solve very quickly and reliably, and seldom require much tuning. Integer programs, on the other hand, can be very complex to solve in some cases, so that experimentation with engine settings may be justified with hard problems. Here is a brief list of some of the settings that may be of interest, but see Chapter 7 for detailed descriptions.

Termination Control

Iterations: Maximum number of pivots by the simplex algorithm.

MaxTime: Maximum number of seconds the optimizer will spend on the problem.

MaxTimeNoImp: The maximum number of seconds with no substantial improvement.

Tolerance: The amount of improvement required within ***MaxTimeNoImp*** in order to continue.

IntTolerance: If branch-and-bound can prove its current solution is within this percentage of the true optimal, it will stop.

Preprocessing

Scaling: Whether to scale decision variables and constraints for Simplex. If coefficients vary by many orders of magnitude, numeric instabilities may result without scaling.

Presolve: When on, the engine performs a presolve step removing singleton rows or columns, fixed variables and redundant constraints, and tightening bounds.

StrongBranching: Performs experiments prior to solving to estimate the impact of branching on each integer variable. The up-front cost may pay off later in more efficient branch-and-bound searches.

Branch & Bound Hints

IntCutoff: A bound you provide in advance on the objective function value for the optimal solution -- an upper bound for a minimization, or a lower bound for a maximization. If you can provide such a bound, the branch-and-bound algorithm may be able to prune huge portions of its search space.

UseDual: Controls whether the dual or primal basis is used by Simplex to solve subproblems generated by branch-and-bound.

ProbingFeasibility: Engine attempts to deduce values of certain binary variables based on settings of others, prior to actually solving a subproblem.

Primal Heuristic: Uses a heuristic method to attempt to discover a solution early in the branch and bound process. This can be an effective way to locate an IntCutoff early, to dramatically prune the branch and bound search space later.

Cut Generation

MaxRootCutPasses, MaxTreeCutPasses: Controls how many cut passes are carried out at each step of the solution process.

GomoryCuts, KnapsackCuts, ProbingCuts, OddHoleCuts, MirCuts, TwoMirCuts, RedSplitCuts, SOSCuts, FlowCoverCuts, CliqueCuts, RoundingCuts, LifeAndCoverCuts: These are all different "cut" methods that attempt to deduce constraints from existing integer commitments, thus reducing the space of feasible solutions. Each introduces a time overhead, that could be spent searching instead, but in problems where a method is effective, speed-up can be dramatic.

Tolerance and Precision

ReducedTol, PivotTol: The control which rows or columns are candidates for pivots during the Simplex algorithm.

Precision, PrimalTolerance: Controls the amount of numeric error by which a constraint can be violated and still be considered satisfied. For example, in an equality constraint, you don't want 1-bit of numeric error to result in an unsatisfied constraint.

Array Abstraction

As with most Analytica functions, **`LpDefine()`** and all the functions used to retrieve the solutions to a linear program are fully array-abstractable. If, for example, you supply an array of coefficients to the ***ObjCoef*** parameter of **`LpDefine()`** that is indexed by index *In1* in addition to the *Variables* index, **`LpDefine()`** will return multiple **`<<LP>>`** objects, with the collection being indexed by *In1*. When such a result is solved, multiple optimization problems will be run.

If any parameter that expects a particular dimension is supplied an object without that dimension, **`LpDefine()`** will treat it as if that dimension were specified with the value constant across that dimension. So, for example, specifying the parameter

```
RHS: 1
```

would treat the right-hand-size of every constraint has having the value 1.

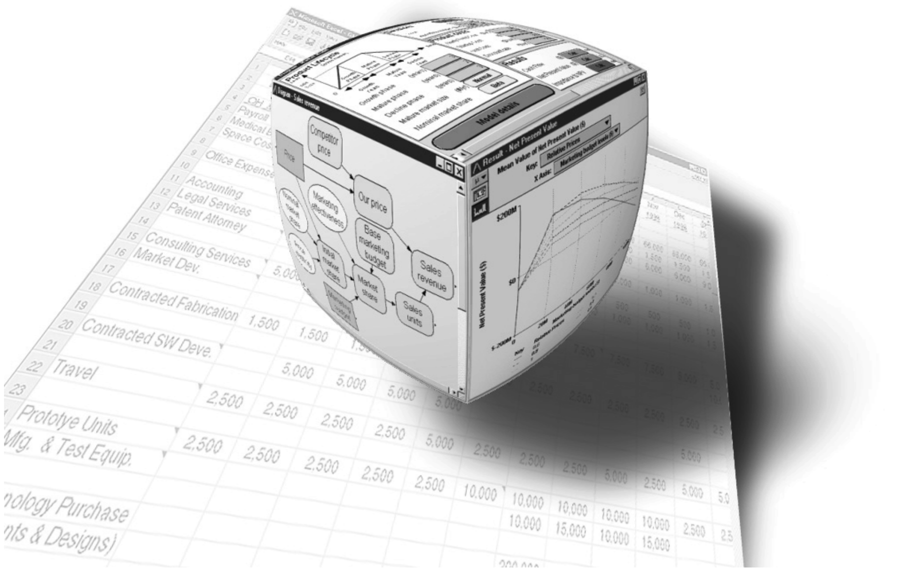
Because these functions are fully array abstractable, any coefficient, bound, or other parameter may be uncertain, evaluated as a sample (indexed by **`Run`**), computed from probability distributions or chance variables. When evaluated in probabilistic mode, these models will solve a separate optimization problem for each sample.

Linear programs involving time can also be embedded in Dynamic loops (see Chapter 17 in the Analytica User's Guide: "Modeling Changes over Time"). By specifying a parameter value that is a function of a previous time step, and using **`LpDefine()`** from within a Dynamic loop, a separate optimization can be performed at each Time point.

Chapter 5 *Quadratic Optimization*

This chapter shows you how to:

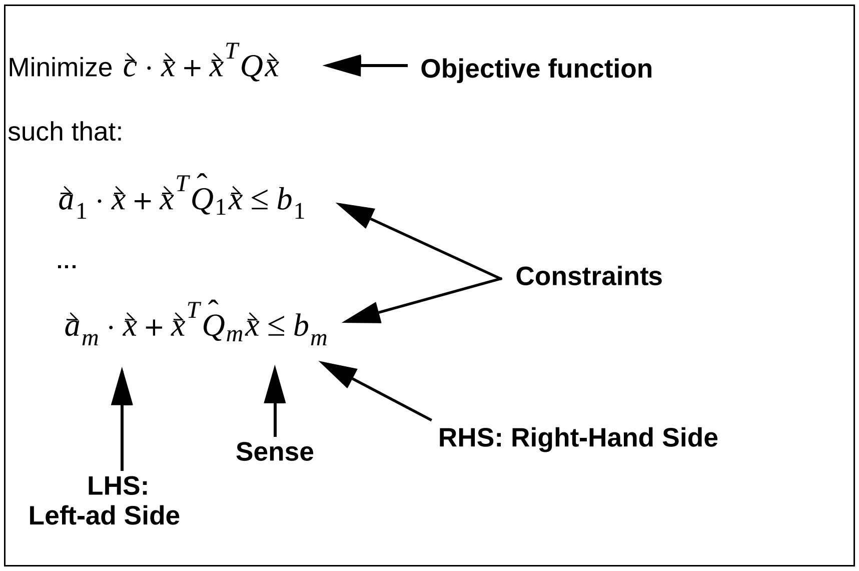
- Define a quadratic optimization
- Solve a quadratic optimization
- Use sample quadratic optimizations as a starting point for your own optimizations



Quadratic Program Optimization

Defining a Quadratic Program

The general form for a *quadratically-constrained quadratic program* accepted by Analytica Optimizer is:



Minimize $\vec{c} \cdot \vec{x} + \vec{x}^T Q \vec{x}$

such that:

$\vec{a}_1 \cdot \vec{x} + \vec{x}^T \hat{Q}_1 \vec{x} \le b_1$

...

$\vec{a}_m \cdot \vec{x} + \vec{x}^T \hat{Q}_m \vec{x} \le b_m$

The objective and constraint left-hand sides are written here in matrix notation. The term $\vec{c} \cdot \vec{x}$ is the linear part of the objective, and each constraint has a linear part $\vec{a}_i \cdot \vec{x}$. These linear parts are the same as the objective and constraint left-hand sides of a linear program, with coefficient vectors $\vec{c}, \vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m$. This formulation augments the linear program by adding quadratic terms to the objective and to constraints. In a pure quadratic program, only a quadratic objective is used, and the constraints are all linear, so that the Q_i matrices are omitted (or 0). The more general case in which constraints may also be quadratic is refered to as a quadratically constrained quadratic program, but for conciseness, we use *quadratic program* to cover the general case.

The quadratic terms, i.e., $\vec{x}^T Q \vec{x}$ in the objective and $\vec{x}^T \hat{Q}_i \vec{x}$ in the constraint are specified by the $n \times n$ matrices $Q, Q_1, Q_2, \ldots, Q_m$, where n is the number of decision variables and m is the number of constraints, and $\vec{x}^T$ denotes the vector transpose of the decision variables.

To ensure that the **Q** matrices are square, you need to specify a second index (***Vars2*** in the example on the next page), with the same number of elements as the first index, ***Vars***. (An array in Analytica may be indexed only once by the same index.)

A quadratic program is defined with these parameters:

{required parameters:}

```
QpDefine(Vars, Vars2, Constraints: Index:
    c: Optional Number[Vars];
    Q: Numeric[Vars, Vars2];
    Lhs: Numeric[Vars, Constraints];
```

```
        LhsQ : Number[Vars,Vars2,Constraints];
        Rhs: Numeric[constraints];
```

{Optional parameters:}

```
        sense: Optional Text[Constraints];
        maximize: Optional Boolean;
        lb,ub: Optional Number[Vars];
        ctype: Optional Text[Vars];
        group : Optional Text[Vars];
        guess: Optional Number[Vars];
        Parameter : Optional Text;
        Setting : Optional Number;
        Engine: Optional Text
    )
```

Note: *The parameters to* **`QpDefine( )`***are not shown here in exactly the same order that the function expects. The* ***LhsQ*** *parameter has been moved up to its logical position in the text for clarity, and a several deprecated parameters are not shown here. Because there are so many optional parameters to this function, you should always use the named-parameter calling convention when using* **`QpDefine()`**. *In the named calling convention, you preceed each argument the parameter name, as in the following example:*

> ***QpDefine (Vars:*** *Part,* ***Vars2:*** *Part2,* ***Constraints:*** *ConstraintIndex,*
> ***c:*** *coef,* ***Q:*** *Qcoef,* ***Lhs:*** *LhsCoef,* ***LhsQ:*** *LhsQcoef,* ***Rhs:*** *b*
> ***sense:*** *'<=',* ***maximize:*** *True* ***)***

When evaluated, **`QpDefine()`** returns a quadratic program object, which displays as **`<<QP>>`**. The optimum solution is not solved until one of the routines to access the solution, such as **`LpStatusNum()`** or **`LpSolution()`** is called.

Optional Parameters

The optional parameters ***sense*** ('<=', '=', or '>='), ***maximize*** (true to maximize objective), ***lb*** and ***ub*** (upper and lower variable bounds), ***ctype*** (continuous/integer type, 'C', 'B', 'I', or 'G'), ***group*** (integer-group), and ***parameter*** and ***setting*** (for specifying search control settings) are all also optional parameters of **`LpDefine()`** and are described in the chapter on linear programming. As with linear programs, Analytica Optimizer supports integer, binary, and mixed-integer quadratic programs.

The optional ***guess*** parameter provides an initial guess for a solution which may or may not be utilized by the optimization engine to disambiguate multiple extrema when a Q matrix is indefinite (see below).

Engine

The optional ***engine*** parameter can be used to explicitly select which optimization engine to use to solve the problem. By default, ***engine:****"LP/Quadratic"* is used with a quadratic objective and linear constraints, and ***engine:****"SOCP Barrier"* is used when quadratic constraints exist. If the constraints are not convex, the "SOCP Barrier" engine may return a status 1102 ("The quadratic constraints are non-convex"). If this happens, you may need to use ***engine:****"GRG Nonlinear"* to obtain a solution.

Solution Properties

The Hessian of the objective function is a second partial derivatives of the objective relative to each pair of decision variables, and is given by $Q + Q^T$. Depending on the values in this Q matrix, the objective function may have a number of different shapes, and the objective may contain a single extreme (minimum or maximum), an infinite number of extrema, or no extreme values. The optimum value to a quadratic program may lie at the objective's extrema, or it may exist on a constraint boundary.

Positive & negative-definiteness

When the **Q** matrix of a minimization problem is *positive-definite*, meaning that for all non-zero $\vec{x} : \vec{x}^T \mathbf{Q} \vec{x} > 0$: the objective function has a "bowl" shape with a single extrema. Similarly, for a maximization problem if it is *negative-definite* it will have a bowl-shape with a single extrema. When the extrema is a feasible solution, it will be the unique optimal solution to the quadratic program. The quadratic programming algorithms are optimized for this case.

Semi-definiteness

When the **Q** matrix is *positive semi-definite* (or *negative semi-definite* for a maximization problem), the objective will have a "trough" with infinitely many extrema. In such a case, the optimizer will find one of the feasible points in the trough.

Indefinite objective

If the **Q** matrix is *indefinite*, the objective will have a "saddle point". Like an extrema, a saddle point has a zero gradient, but is not an actual optimum. The true optima (one or many) will lie on the constraint boundaries. In an indefinite case, the optimizer will converge either to the saddle point, or to one of the optimum solutions on the constraint boundaries. If it converges to the saddle point, which might not be optimal, **`LpStatusNum()`** will return 65 ("objective changing too slowly"). The final point reached by the optimization depends on the initial starting point for the search, which may optionally be specified using the parameter guess to **`QpDefine()`**:

```
guess: Optional Number[Vars];
```

The guess parameter is only relevant if **Q** is indefinite, otherwise the same end result will be reached regardless of the starting point.

Because **`QpDefine()`** is totally array-abstractable, you can provide multiple guesses by dimensioning the argument to this parameter by an index other than **`Vars`**, with different starting points. In that case, multiple quadratic optimizations will be solved, each at different starting points.

Convexity

The set of constraints are said to be *convex* when the set of feasible solutions is a convex subset of all potential solutions. A convex subset is one in which for every two points in the set, all points on the line segment connecting them are also in the subset.

A quadratic constraint is convex in these cases:

- The matrix $\hat{Q}_i$ is positive semi-definite and the constraint sense is '<='
- The matrix $\hat{Q}_i$ is negative semi-definite and the constraint sense is '>='
- The constraint is linear.

When all constraints are convex, then the set of feasible solutions is convex. Quadratic programs can be solved efficiently when the set of feasible solutions is convex. Positive (or negative) semi-definiteness of Q can be tested for using Analytica's **EigenDecomp()** function. The matrix is positive (negative) semi-definite if all Eigenvalues are non-negative (non-positive). Note: if your Q is not symmetric, use EigenDecomp on $Q + Q^T$.

Common Quadratic Situations

Quadratic programs arise in several applications, one of the most common being portfolio optimization.

Portfolio allocation

Assume there are *N* investments, each with an uncertain outcome. The investments are not independent; for example, two investments in the same sector may be influenced by similar market forces and thus be highly correlated. Other pairs of investments may be negatively correlated. A symmetric covariance matrix, ***Q***: can be used capture the pair-wise covariances between investments, as well as the variances of the individual investments (the diagonal elements of ***Q***). Letting each element of the vector $\vec{x}$ be the fraction of the total portfolio allocated to investment, the variance of the complete portfolio is $\vec{x}^T \mathbf{Q} \vec{x}$. As a result, various objective functions used in portfolio optimizations will depend on the net variance of the portfolio, and lead to quadratic programs of the form show under "Choosing the type of optimization" on page 21. Two such examples are demonstrated in the example model **`Asset Allocation.ana`** found in the **`Example Models\Optimizer Examples`** directory.

When sample covariances are computed from historical data, and the number of time periods used is greater than the number of dimensions (*e.g.*, the number of investments), the resulting ***Q*** matrix is guaranteed to be positive-definite. As discussed in the previous section, then property lends itself well to solution by quadratic programming.

Obtaining the Solution

The **`QpDefine()`** function defines the quadratic program, but does not solve it. The optimum is solved for when **`LpSolution()`**, **`LpStatusNum()`**, **`LpStatusText()`**, or any of the other functions that use the solution are called.

The functions **`LpSlack()`**, **`LpShadow()`**, and **`LpRhsSa()`** are all available for quadratic programs (see the discussion for each of these in Chapter 4, "Linear Optimization")

The **`LpReducedCost()`** function can also be called on a quadratic program.

Search Control Settings

In addition to settings found on linear programs, several additional settings apply to quadratically constrained problems. These can be altered from their default values using the two parameters to **`QpDefine()`** named ***parameter*** and ***setting***. See Chapter 7, "Control Settings", for details on using search control settings, and for descriptions of available settings.

Examples

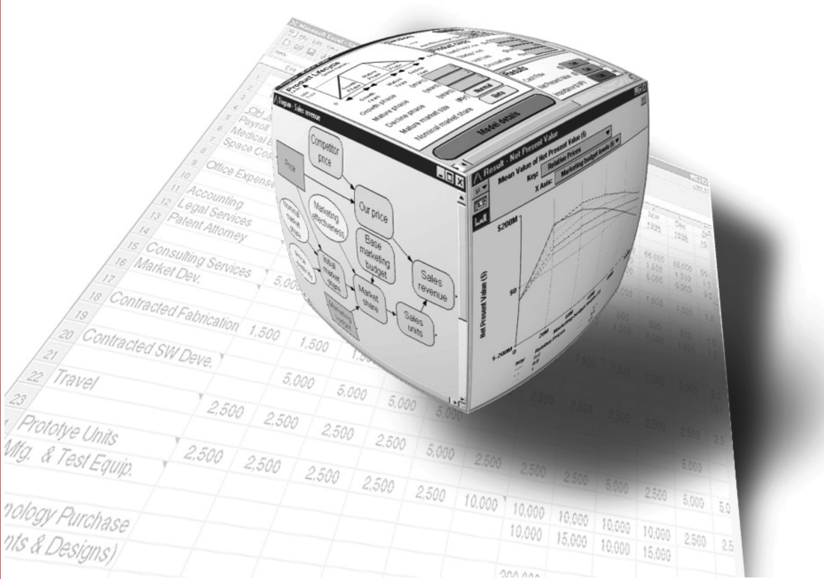
The **`Example Models/Optimization Examples`** directory, installed with Analytica, contains an example model demonstrating quadratic optimization:

- **`Asset Allocation.ANA:`** Portfolio optimization is a classic quadratic programming application. This example demonstrates four formulations of an asset allocation problem, two of which are quadratic programs.

Chapter 6 *Non-Linear Optimization*

This chapter shows you how to:

- Formulate a non-linear optimization problem
- Obtain the solution for a non-linear optimization problem
- Give hints to help the optimizer
- Control the search



Non-linear Optimization

Non-Linear Programs

A non-linear program (NLP) is the most general formulation for an optimization. The objective and the constraints can be arbitrary functions of the decision variables, continuous or discontinuous. This generality comes at the price of longer computation times, less precision than linear and quadratic programs (LP and QP). There is also the possibility with smooth NLPs, that the Optimizer will return a local optimum that is not the global optimum solution. In general, it is hard to prove that a solution is globally optimal or not. For these reasons, it is better to reformulate nonlinear problems as linear or quadratic when that is possible.

Linear and quadratic problems define the objective function as arrays of linear or quadratic coefficients.They pass these arrays as parameters to the Optimizer, which operates on them directly to find a solution without further interaction with the rest of the Analytica model. For nonlinear problems, the objective function is defined as an Analytica expression or variable that depends on the decision variables. In this case, the Optimizer repeatedly evaluates the objective function as it tries assigns different values to the decision variables in its search for a solution. It does the same with expressions passed to **`Lhs`**, the left-hand side of the constraints.

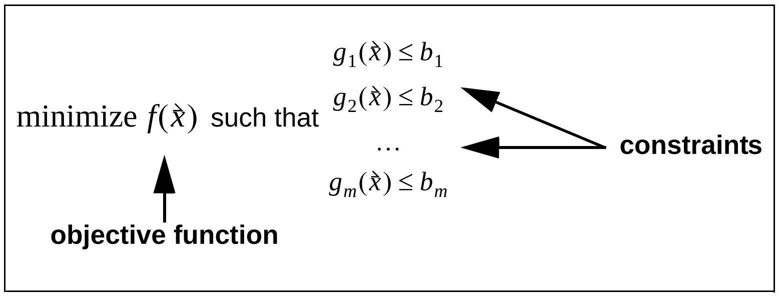
This approach imposes certain restrictions on array abstraction (support for Intelligent Arrays) for NLPs — for example, requiring the objective function to return a single (scalar) number. We devote a section of this chapter to showing how to work with these restrictions so that you can apply NLP optimization to create arrays of optimizations for models with uncertainty (samples indexed by Run), for parametric analysis, and dynamic models over time, or other Indexes.

The Optimizer has a variety of methods, including gradient-based search, branch-and-bound, and genetic algorithms, from which it chooses to try to suit the problem. In many cases, you can give it information about the problem that can help it choose the most appropriate methods, and so work faster and more reliably. Such hints include:

- The type of dependence — i.e., whether the objective or constraint functions vary linearly, smoothly, or discontinuously with each decision variable.
- The gradient and Jacobian expressions to compute the needed partial derivatives for the objective much faster at each search point
- Control parameters to influence how the search is performed.

Problem Formulation

The basic formulation for a non-linear optimization is:



where $\vec{x}$ is a vector denoting the *n*-dimensional candidate solution. A non-linear optimization problem is defined using the function **`NlpDefine()`**, shown here without optional parameters:

```
NlpDefine(Vars, Constraints: Optional Index;
             X: Variable;
             Obj, LHS: Optional Expression;
             RHS: Optional Numeric[Constraints])
```

Vars	An index for the decision variable, ***X*** below. This can be omitted when there is only one scalar decision variable.
Constraints	An index for the constraints, ***LHS*** and ***RHS*** below. This is optional when there are fewer than two constraints.
X	The decision variables, indexed by ***Vars***. The parameter passed to ***X*** must be the name (identifier) of a global variable or decision, or a local variable, not an expression: As the NLP Optimizer searches for better solutions, it assigns new values to the decision variable, and computes the corresponding value of the Objective.
Obj	The objective to maximize or minimize, according to the setting of optional parameter **`Maximize`**. It may be a variable or an expression. It must depend on the decision variables, ***X***, directly, or indirectly via other variables. When no objective is specified, the optimizer stops when a feasible solution, satisfying all the constraints, is found.
LHS	The Left-Hand Side of the constraints, indexed by ***Constraints***, if the ***Constraints*** index is specified. You may omit ***LHS*** if you have no constraints. If you have a single constraint (and ***Constraints*** is omitted), the expression should evaluate to a scalar. If you have more than one constraint, then ***Lhs*** should be indexed by Constraints. Each element of ***LHS*** may be an expression or a variable. They must depend on the decision variables ***X***, directly or indirectly.
RHS	The Right-Hand Side of the constraints, indexed by ***Constraints***, if the ***Constraints*** index is specified. You may omit this if you have no constraints. If you have a single constraint, ***RHS*** should evaluate to a single number, and if you have multiple constriants, each element of the array passed to ***RHS*** must evaluate to a single number. It must not depend on the decision variables. By default, feasible solutions are those in which the ***LHS*** is less than or equal to the corresponding value of ***RHS***. You can change this with the optional ***Sense*** parameter, described below.

Optimization without constraints

To solve for the unconstrained minimum of `f(x)` where x is a scalar, only the decision variable, ***X*** and the objective need to be specified. For example, the following defines an NLP to find the minimum of $f(x) = \cosh(1 + \sinh(x))$:

```
var x:=0;
NlpDefine( X:x, Obj: cosh(1+sinh(x)) )
```

In this example, the decision variable, `x`, is a local variable, and the objective expressed explicitly. Alternatively, the decision variable and objective may be variables in your model, as in this example:

```
NlpDefine( X:annualMaintExpense, Obj:costOfOwnership )
```

To minimize over several decision variables, ***X*** is a vector, and the index of that vector is specified using the parameter ***Vars***. The objective must be a variable or expression that depends on ***X***, but must evaluate to a scalar. For example, to find a vector of coefficients, *Coeff*, indexed by *K*, that minimize *errorMeasure*, the problem is formulated as

```
NlpDefine( Vars: K, X: Coeff, Obj: errorMeasure )
```

Representing Constraints

A system of equations and inequalities can be represented as a set of constraints without an objective. Discrete constraint satisfaction problems can also be encoded using integer-valued variables. When there is no objective, the optimizer will generally terminate when any feasible solution has been located. Systems of nonlinear equations can be extremely difficult to solve, and it may be unrealistic to expect an optimizer to find a solution just because it can be expressed, so a certain degree of realism is essential.

GoalSeek

A goal-seek functionality seeks a value for a scalar *x* that causes scalar *y* to be equal to *g*, where *y* depends on *x*. Since x is scalar, the ***Vars*** index does not need to be specified, and because there is a single constraint, no ***Constraints*** index is necessary. When we use GoalSeek, equality is usually desired, so the ***Sense*** parameter is specified as ’=’ (otherwise it defaults to ’<=’). Since any feasible solution is sought, no object needs to be specified. Hence, simple GoalSeek requires parameters ***X***, ***Lhs***, and ***Rhs***.

```
NlpDefine( X:GrossIncome, Lhs:NetIncome, Sense:’=’, Rhs: 1M )
```

or

```
NlpDefine( X:Price, Lhs: Supply(Price)-Demand(Price), Sense:’=’, Rhs:0)
```

or using a local variable:

```
Var x:=0;
NlpDefine( X:x, Lhs:cosh(1+sinh(x)), sense’=’, rhs:1.001 )
```

Multiple Constraints

When there is more than one constraint, an index parameter, ***Constraints***, must be specified. When ***LHS*** is evaluated during the optimization search, the result must be indexed by the ***Constraints*** index, and only by the ***Constraints*** index. Often it is convenient to set up a variable that computes ***LHS***, defined as a table indexed by ***Constraints***, with a separate expression in each cell (the $g_i(\vec{x})$ in the basic formulation).

RHS should also evaluate to an array indexed by ***Constraints***; however, unlike ***LHS***, ***RHS*** is constant during the optimization search, so ***RHS*** should not depend on ***X***.

Obtaining the Solution

The same functions used to obtain the solution to LP and QP optimizations also work for NLP. These include: **`LpStatusNum(), LpStatusText(), LpOpt(), LpSolution(), LpSlack(), LpShadow()`**, and **`LpReducedCost()`**.

For more, see Chapter 9, "Optimizer Function Reference," on page 94.

Optional Parameters for NLP

Every parameter of **`NlpDefine( )`** except ***X*** is optional; however, either an objective or at least one constraint is required. In addition to the core parameters used for specifying the objective and constraints, the following are optional parameters to **`NlpDefine()`** can also be specified. :

Maximize

By default, **`NlpDefine()`** defines a minimization problem. You should set the optional parameter

```
Maximize: Optional Boolean
```

to **`True`** when you wish to maximize the objective.

Sense

By default, each constraint specifies that the left-hand side is less-than or equal to the right-hand side. Using the optional ***Sense*** parameter, you can change the relationship between left-hand and right-hand sides:

```
Sense: Optional Text[Constraints]
```

- **`"<"`**, **`"<="`**, or **`"L"`**: ***LHS*** is less-than or equal to ***RHS***
- **`">"`**, **`">="`**, or **`"G"`**: ***LHS*** is greater-than or equal to ***RHS***
- **`"="`** or **`"E"`**: ***LHS*** is equal to ***RHS***

If you pass a single text value, such as **`"="`**, to the ***Sense*** parameter, that sense will apply to all the constraints. If you want a different sense for each constraint, pass an array indexed by ***Constraints***, with each cell containing its own text value **`"<="`**, **`">="`**, **`"E"`**, etc.

Bounds

You can define upper and lower bounds on each decision variable for an NLP problem, as for LP and QP problems, using these optional parameters:

```
Lb, Ub: Optional Number[Vars]
```

If not explicitly specified, the optimizer assumes bounds of **`-INF`** and **`+INF`**, i.e., that the decision is unbounded. If you pass a single number to either of these parameters, that bound applies to all decision variables. So, for example:

```
NlpDefine(…, Lb:0,Ub:1 …)
```

specifies that all decision variables are in the range 0 to 1. If lower or upper bounds are different for each decision variable, pass them arrays of numbers indexed by ***Vars***.

Array Abstraction

`NlpDefine()` does not automatically array abstract over extra dimensions that appear in the results of evaluating the objective or constraint left-hand side expressions. When the objective function, ***Obj***, is evaluated, it should be a single number with no extraneous indexes. Similarly, when ***Lhs*** is evaluated, it should contain only the index passed to Constraints parameter. If you have multiple objectives, you must combine them (for example, as a weighted average) if they are all criteria that you wish to optimize simultaneously. If instead you desired multiple optimization problems, for example, for each combination of scenarios, then you must tell **`NlpDefine()`** about these dimensions explicitly, otherwise an error will result when ***Obj*** or ***Lhs*** is discovered to have extra dimensions.

Two parameters of **`NlpDefine()`** can be used in some situations to specify dimensions that must be iterated, in which a separate optimization should be conducted for each combination of elements in these extra dimensions.

```
Over : ... optional atomic
SetContext : ... optional Variable
```

Each of these are optional repeated parameters, allowing several indexes (in the case of ***Over***) or several parametric variables (in the case of ***SetContext***) to be specified.

The ***Over*** parameter specifies a list of indexes that should be abstracted over. For example, if we wish to run a separate optimization problem for each possible discount_rate, and each possible Initial_Investment option, where Discount_rate and Initial_Investment are variables that may contain lists, we would specify:

```
NlpDefine(..., Over: Discount_rate, Initial_investment)
```

If Discount_rate rate contains 3 elements, Initial_investment contains 5 elements, then 15 separate optimization problems will be defined. No error results if Discount_rate or Initial_investment is not a list, which means that values that might be set to a list during a parametric exploration, but normally are not, can be listed here. In addition, if there is a result in your model that contains the extra dimensions, the identifier of the result array can be listed, and each of the dimensions in that result are used. This fact is useful for defining an NLP that abstracts across the Run dimension in sample mode, but not in Mid mode, using e.g.,:

```
NlpDefine(..., Over: Uniform(0,1) )
```

The Over parameter does two things. It instructs Analytica to define multiple NLPs, and instructs NLP define to pay attention to one particular slice of ***Obj*** or ***Lhs*** in each NLP when there are extra dimensions. It is important to note, however, that ***Over*** does not restrict how the rest of your model is evaluated when computing ***Obj*** and ***Lhs***. If you rely only on the ***Over*** parameter, your evaluation of ***Obj*** and ***Lhs*** may be computing far more than necessary, most of which gets ignored, resulting in slow evaluations during optimization.

The ***SetContext*** parameter provides a mechanism to restrict your computation to only the slice that is used by a particular NLP. For example, instead of computing ***Obj*** for every possible discount_rate, and then throwing away all results except the one corresponding to the current NLP, it is more efficient to set discount_rate to a single value before running the optimization search, thus limiting the computation of Obj at each cycle to the single discount_rate that is used. ***SetContext*** specifies a list of variables that will be restricted to a single value for the duration of one particular optimization.

To use ***SetContext***, the variable listed must contain a list-based domain attribute, containing the set of possible values for that variable. Each instance of NLP corresponds to one of those possible values, and when that instance runs, the variable is set to that single value. For example

```
Discount_rate := Choice(Self,0)
Domain of Discount_rate := [8%,10%,12%]

NlpDefine(..., SetContext: Discount_rate)
```

If the ***Over*** parameter is also specified, for example with other indexes, the parameter specified in ***SetContext*** should also be specified in ***Over***. If the context variable evaluates to a single number, it is not set (hence, it will work fine with **`Choice()`**, abstracting only when ***All*** is selected in the choice pulldown).

The use of ***Over*** and ***SetContext*** may allow array abstraction in some situations, but they are not always applicable, and not always the best way to achieve an array-abstractable optimization. In the **`The Airline Example for NLP`** below, a variety of other techniques are demonstrated.

Integer, Binary and Mixed-Integer Programs

Like the LP and QP optimizers, the NLP optimizer can handle discrete decision variables — that is, integer or binary (Boolean) — as well as continuous values. Use the parameter

```
Ctype: Optional Text[Vars]
```

to specify the continuity type of each decision variable by providing one of the following text values for each variable:

- **`"C"`**: Continuous
- **`"I"`**: Integer
- **`"B"`**: Binary or Boolean
- **`"G"`**: Grouped integer

The non-linear optimizer uses a genetic, or evolutionary, algorithm, when discontinuous variables are present.

When the grouped integer parameter is used, each grouped integer variable belongs to one group, where each group contains at least two grouped integer variables. Within a group, a feasible solution assigns the integers 1..N to each variable, where N is the number of variables in the group, such that all variables have different values.When you have more than two groups, you should also specify the ***group*** parameter:

```
Group : Optional Number[Vars]
```

This parameter specifies the group number for each grouped integer variable. If ***group*** is omitted, all variables are assumed to belong to the same group. The **`"traveling salesman.ana"`** example model demonstrates the use of grouped-integer variables (belonging to a single group).

Hard integers vs. soft integer constraints

The ***CType*** parameter can be used to indicate that a decision variable in any feasible solution is integer-valued. However, during the course of the search, the non-linear optimizer may explore non-integer values for these variables while exploring the rate of change within the search space. If your model produces well-defined results for non-integer values, then this type of *soft-integer* constraint does not present a problem, and the optimizer can benefit from being able to explore non-integer solutions.

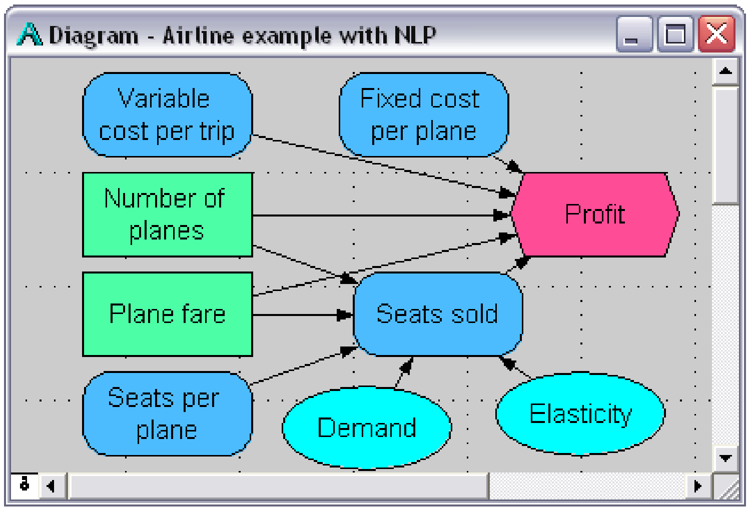
In some cases, however, you may need to enforce a *hard-integer* constraint. For example, a table lookup may encounter an error when a non-integer lookup is attempted. Or your model may encounter errors for other reasons when non-integer values are attempted. If you have hard-integers, you can either modify your model to return something sensible for non-integer values (for example, by linearly-extrapolating), or you should explicitly select the "Evolutionary" engine using the parameter:

```
Engine : "Evolutionary"
```

When using the Evolutionary engine, two control settings are relevant (these are set using the ***parameter*** and ***setting*** parameters to **NlpDefine()**). First, at each iteration, a new sample point is generated by mutating or combining two members of the population via crossover. Then, that sample may be improved further using a local search before adding it to the population. For example, a gradient descent may be utilized to find the nearest local optima before adding it. ***LocalSearch*** controls whether, any by what method, local searches are performed. By leaving at its default, off, position, non-integer solutions will not be explored. If you do elect to utilize local search, then the ***FixNonSmooth*** setting controls whether gradient information in local searches is limited to continuous variables. If you have hard-integer constraints with local search, then you should ensure that ***FixNonSmooth*** is indeed on. These parameter settings do not impact the "GRG Nonlinear" engine.

The Airline Example for NLP

Here we introduce the airline decision problem. We will use this example in the rest of this chapter with eight cases that illustrate how to formulate problems for NLP, including situations in which parameters have extra indexes, for dealing with uncertainty, parametric analysis, and dynamic models over time. You can find this example in the **`Example Models/Optimizer Examples/Airline NLP.ANA`**. It includes the eight different cases described below. Please open the model in Analytica to see full details.



A small airline is trying to decide how many planes to lease and what fare to charge on a new route. It has two decision variables — **`Num_planes`**, the number of planes allocated for this route, and **`Fare`**, the price charged for trips on this route — and two chance variables — the **`Base_demand`** for seats (assuming the fare is $200) and the **`Elasticity1`** of demand with respect to price:

```
Decision Num_planes := 2
Decision Fare := 200 ($/passenger trip)
Chance Base_demand :=
   Triangular(300K, 400K, 500K) (trips/year)
Chance Elasticity1 := Triangular(2, 3, 4)
```

Note: *When you first load the model,* **`Num_planes`** *and* **`Fare`** *will not be set to these single numbers. You can change them at this point to single numbers, or even to the following sequences to view* **`Profit`** *parametrically.*

```
Decision Num_planes := 1..5
Decision Fare := 100..300
```

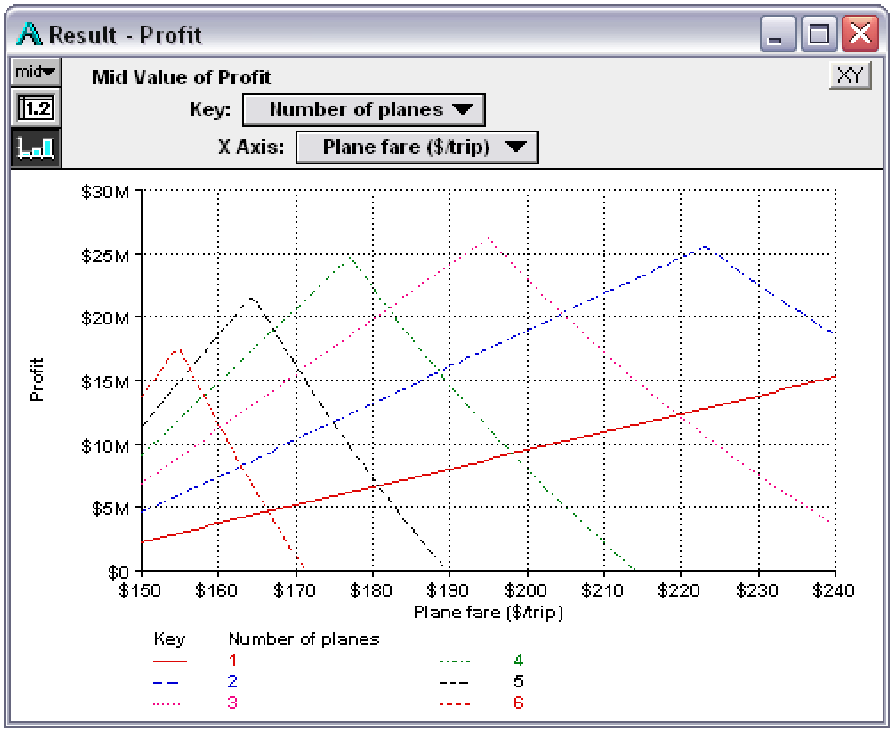
We assume that the demand is elastic with respect to changes in price, using a demand function that decreases as **`Fare`** increases at a exponential decay rate determined by **`Eliasticity1`**. At a base fare of $200, the demand is equal to the **`Base_Demand`**. We compute the actual **`Seats_sold`** as the lesser of the demand modified for price elasticity and the actual seats available, the product of the number of planes and annual **`Seats_per_plane`**:

```
Variable Seats_per_plane := 200 * 360 * 2
Variable Seats_sold := Min([Base_demand *
   (Fare/200)^-Elasticity1,
   Num_planes * Seats_per_plane])
```

Finally, we model the Objective variable **`Profit`** as the difference between revenues and costs, including **`Fixed_cost`**, the annualized fixed cost of leasing and operating each plane, and **`Var_cost`**, the incremental cost for each new passenger:

```
Variable Fixed_cost := 12M ($/plane/year)
Variable Var_cost := 100 ($/passenger trip)
Objective Profit := Seats_sold*Fare
 - Seats_sold*Var_cost - Num_planes*Fixed_cost
```

This graph shows **`Profit`** as a function of the two decision variables, using parametric approach to visualize the effects. Note that for each number of planes, 1 to 5, the profit is a sharply peaked function of the fare. The optimum fare is at the highest peak, $195 with 3 planes.



In this simple case, with only two decision variables, you can visualize the objective function and find the optimal values (or close) by parametric analysis. For more complex problems, the Optimizer is essential. We now show how to apply that.

Reformulating the decision variables for NLP

We usually need to reformulate a decision problem, at least a little, to apply NLP. One reason is that **`NLPDefine()`** expects a single, array-valued decision variable for parameter **`X`**. So, if you want to apply NLP to optimize a model, like the airline example, whose decision variables are two or more separate Analytica variables, you need to combine these decisions into a single array-valued decision. If the model has *n* scalar decision variables, you should define a decision variable **`Decisions`** as a one-dimensional array with an index containing *n* elements. For the airline example, we define **`Decisions`** with two elements, corresponding to its two decisions, **`Num_planes`** and **`Fare`**:

```
Index Dvars := ['Number of planes', 'Plane fare']
Decision Decisions :=Table(Dvars)(3, 200)
```

The values in the table are the initial values, prior to optimizing. We must now redefine the individual decision variables so that they obtain their values from the corresponding elements of **`Decisions`**:

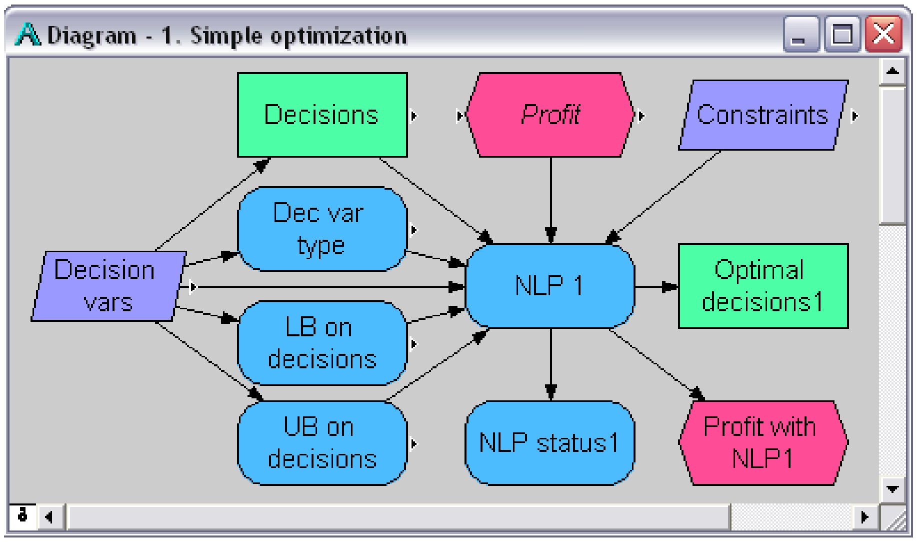
```
Num_planes:= Decisions[Dvars ='Number of planes']
Fare := Decisions[Dvars = 'Plane fare']
```

As the Optimizer searches for optimal values, it will assign successive new candidate solutions to **`Decisions`**, and get the resulting value of **`Profit`**, which in turn gets its values from **`Decisions`**, via **`Num_planes`** and **`Fare.`**

If one or more of the original decision variables is an array, the new decision variable **`Decisions`** passed to **`X`** must still have only one dimension. Its size should be the sum of the sizes of all the original decision variables. Again, you should assign the current initial values of the original decision variables to the corresponding elements of **`Decisions`**. Then you redefine each original decision variable so that it gets each element from the corresponding element of **`Decisions`**. See **Case 7. NLP with Optimizations over time** below for an example, where we add the **`Time`** dimension to **`Num_planes`** and **`Fare`**.

Case 1. Simple NLP Optimization

We will now complete the formulation of the NL P for the airline problem introduced above, creating a model that looks like this:



We need to specify the type of each decision — **`'I'`** (integer) for Number of planes, and **`'C'`** (continuous) for Fare — the lower and upper bounds for the two decisions, and the **`Constraints`** index:

```
Variable Dec_type := Table(Dvars)('I','C')
Variable Lb_decisions := Table(Dvars)(1, 100)
Variable Ub_decisions := Table(Dvars)(5, 300)
```

We can now define the NLP using these parameters:

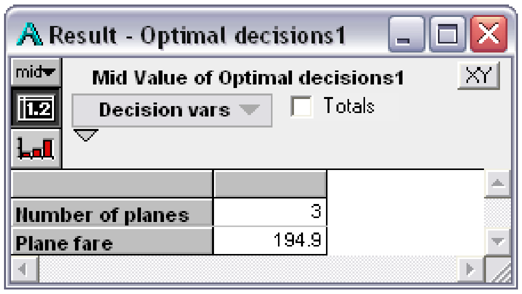
```
Variable NLP_1 := NLPDefine(
    Vars: Dvars, X: Decisions, Ctype: Dec_type,
    LB: Lb_decisions, UB: Ub_decisions,
    Obj: Profit, Maximize: True )
```

Since we want the largest **`Profit`**, we set **`Maximize`** to True. Finally, we define the key results of the optimization: The optimal decisions, the profit with these decisions, and the status of the optimization:

```
Decision Optimal_decisions1 := LPSolution(Nlp_1)
Objective Profit_with_nlp1 := LPOpt(Nlp_1)
```

```
Variable Nlp_status1 := LPStatusText(Nlp_1)
```

When we display the result of any of these three variables, it will perform the optimization. For example, **`Optimal_decisions1`**, gives this table (agreeing closely with the parametric analysis):



Number of planes	3
Plane fare	194.9

Intelligent Arrays, array abstraction and NLP

Unlike most other Analytica functions, including linear and quadratic optimization, nonlinear optimization does not fully support Intelligent Arrays — that is, it will not automatically generalize over extra dimensions for all parameters. Below we show how you can work around these restrictions to create and solve arrays of NLP problems, including handling uncertainty, parametric analysis, and dynamic optimization over time.

NLP's limitations are that the following required parameters must be dimensioned by the specified indexes *and no other indexes*:

X must be indexed only by the index supplied to ***Vars***

Obj must be scalar — a single number with no indexes

LHS must be indexed by the index supplied to ***Constraints***, or have no index.

Similarly, these optional parameters, if specified, must also be dimensioned by only the specified indexes:

Gradient must be indexed only by the index supplied to ***Vars***

Jacobian must be indexed only by the indexes supplied to ***Vars*** and ***Constraints***

See page 66 for details on ***Gradient***, and ***Jacobian***.

Note that **`NlpDefine()`** *does* generalize fully over extra dimensions for all parameters other than those seven listed above. But, for those seven parameters, it is up to you, the modeler, to make sure that they have only the required indexes. Otherwise it will flag an error. Read on to see how to get around these limitations.

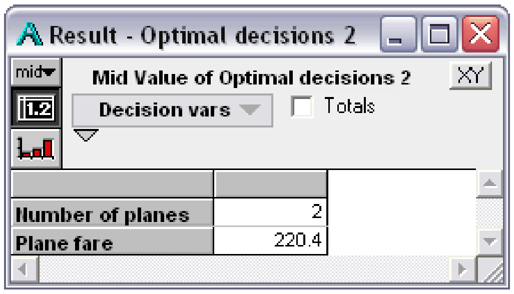
In many of the cases that follow, there are two alternative approaches, one in which we encapsulate the NLP in a user-defined function, and a second utilizing the ***SetContext*** parameter. Since both approaches usually entail certain changes to the underlying model, there are tradeoffs between the two approaches, and in any particular problem you may find one more convenient than the other. The cases that follow demonstrate both alternatives when both are applicable.

Case 2. Maximize expected value: NLP with uncertainty

If you want to find the optimal decisions with an uncertain model, the most common approach is to define the objective as maximizing the *expected value* (i.e. mean) of the objective function — for example, maximizing the expected profit, or the expected utility in a decision analysis formulation. For the Airline example, we define **`NLP2`**, which differs from **`NLP1`** only in that the objective takes the mean of the profit:

```
Variable NLP2 := NLPDefine(... ,
    X: Decisions, Obj: Mean(Profit), ...)
```

In this case, the objective is a single scalar number (i.e., the expected value). Although it is a function of an uncertain quantity, it is not itself uncertain. So you can apply **`NLPDefine()`** directly, and the restrictions on array abstraction mentioned above cause no problems. Note the results of doing the optimization using expected value are a bit different from the deterministic analysis, because the profit function is not symmetric:



Mid Value of Optimal decisions 2

Decision vars	
Number of planes	2
Plane fare	220.4

The same approach works if you want to maximize a statistic of the objective other than mean, such as to minimize the 1st percentile of an uncertain profit (loss), e.g. **`Getfract(Profit, 1%)`**. If there is uncertainty in the constraint functions, you may define the constraints using percentiles (using **`Getfract()`** or other statistical functions) — for example, the constraint that the cumulative cashflow has a >95% chance of being nonnegative.

In these cases, you are trying to find the optimal decision now, *before* resolving the uncertainties that affect the objective or constraints. You can set the model to perform a single optimization and the result is a single optimal solution (set of decisions) and corresponding maximum expected value (or other statistic) of the objective. Given the optimal solution, you can then compute a probability distribution over the objective function to model the uncertainty over the value outcome.

Case 3. NLP with uncertainty: Probabilistic optimization

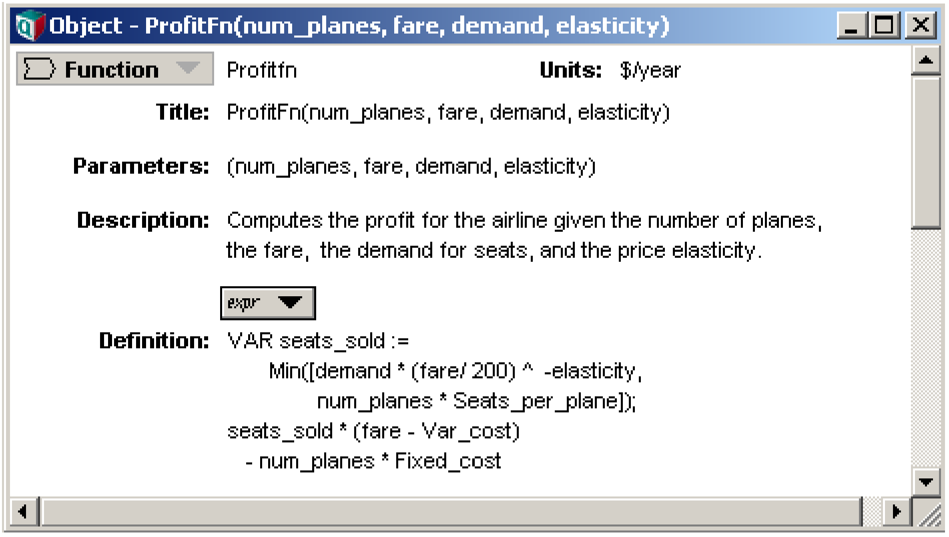
The second type of optimization under uncertainty is less common: The optimal decisions will be made *after* resolving the uncertainty, and you want to compute probability distributions over what those optimal decisions will be now while still uncertain. This is sometimes known as *preposterior* analysis because the optimization is performed *a posteriori* — after the uncertainty is resolved — but you are performing the analysis now, *before* the uncertainty is resolved. (Not to be confused with *preposterous* analysis, which we try to avoid.) This situation requires a sample of optimizations to be performed. It results in a random sample of optimal decisions, and a sample of corresponding values of the objective for each solution.

You might try simply to compute a probabilistic value of the optimal decision in case 1, from NLP_1, by selecting a uncertain view, e.g. ***`Sample`***, in the Result for **`Optimal_decisions1`**, shown previously: But, this would generate the error "The expression for the objective function in NlpDefine must evaluate to a single numeric value during optimization..." This is because the objective, Profit, is no longer a single number at each iteration, but rather a Monte Carlo sample of numbers.

Instead, we need to create an NLP that abstracts over the Chance variables, so that the **`Run`** index does not cause problems for **`NLPDefine`**. Two techniques for accomplishing this are demonstrated here. The first encapsulates the NLP in a user-defined function, and the second utilizes the ***SetContext*** parameter.

UDF Encapsulation

For convenience, we define two functions, first **`ProfitFn()`** that encapsulates the Objective **`Profit`** as a function of the decisions and chance variables as parameters. This function replicates **`Profit`** in the simple airline model. (See next page.)



Profitfn **Units:** $/year

Title: ProfitFn(num_planes, fare, demand, elasticity)

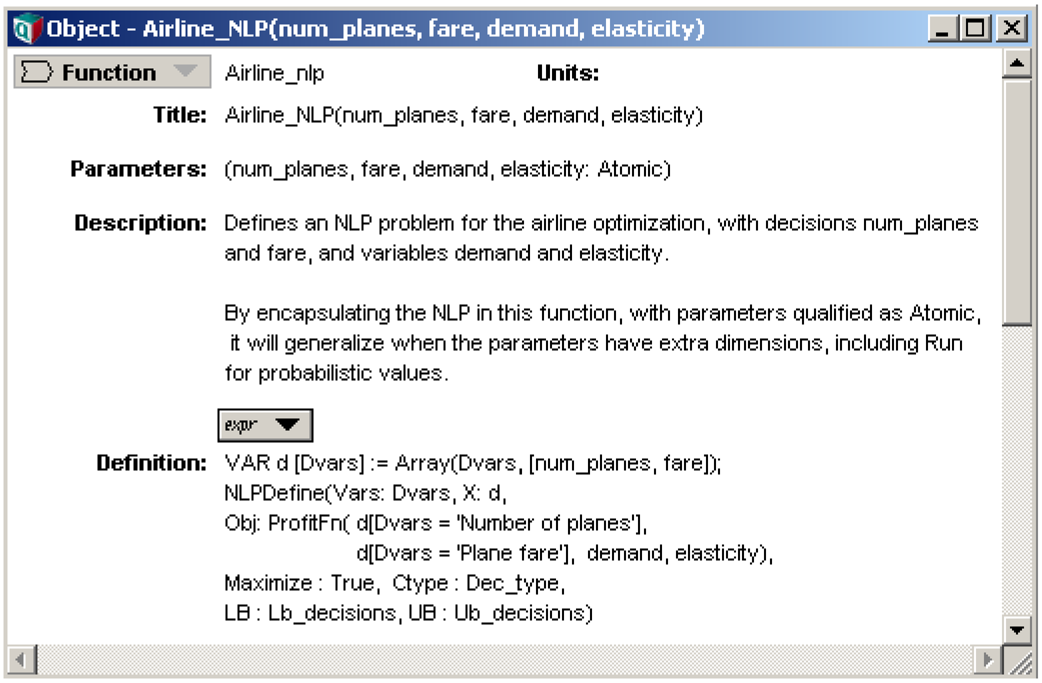
Parameters: (num_planes, fare, demand, elasticity)

Description: Computes the profit for the airline given the number of planes, the fare, the demand for seats, and the price elasticity.

Definition:

```
VAR seats_sold :=
     Min([demand * (fare/ 200) ^ -elasticity,
            num_planes * Seats_per_plane]);
seats_sold * (fare - Var_cost)
  - num_planes * Fixed_cost
```

Then we define a function **`Airline_nlp()`** that defines an NLP using **`ProfitFN()`** that we just defined for the objective:


Object - Airline_NLP(num_planes, fare, demand, elasticity)

Function Airline_nlp Units:

Title: Airline_NLP(num_planes, fare, demand, elasticity)

Parameters: (num_planes, fare, demand, elasticity: Atomic)

Description: Defines an NLP problem for the airline optimization, with decisions num_planes and fare, and variables demand and elasticity.

By encapsulating the NLP in this function, with parameters qualified as Atomic, it will generalize when the parameters have extra dimensions, including Run for probabilistic values.

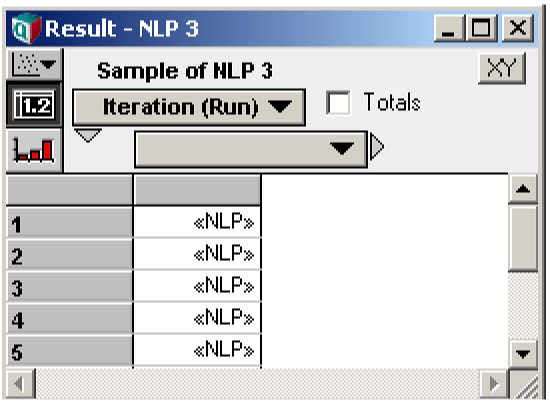
Definition:
```
VAR d [Dvars] := Array(Dvars, [num_planes, fare]);
NLPDefine(Vars: Dvars, X: d,
Obj: ProfitFn( d[Dvars = 'Number of planes'],
               d[Dvars = 'Plane fare'],  demand, elasticity),
Maximize : True,  Ctype : Dec_type,
LB : Lb_decisions, UB : Ub_decisions)
```

`Airline_nlp()` qualifies its parameters as **`Atomic`**. This means means that, if the actual parameters are arrays, indexed by **`Run`** or anything else, it will reduce them all to scalar values and call the function multiple times, once for each combination of scalar values. it calls multiple times, Each time it passes scalar parameters to **`ProfitFn()`**, so that the objective passed to **`Obj`** in **`NLPDefine()`** is scalar, as required. In this way, it restores the Intelligent Array behavior that NLP otherwise lacks.

We now define a variable using this function:

```
Variable Nlp_3 := Airline_nlp( Num_planes, Fare,
                      Demand, Elasticity1)
```

If you show the result of this variable in a sample view (with Samplesize set to 5 for rapid execution), it shows a sample of NLP problems:


Result - NLP 3

Sample of NLP 3

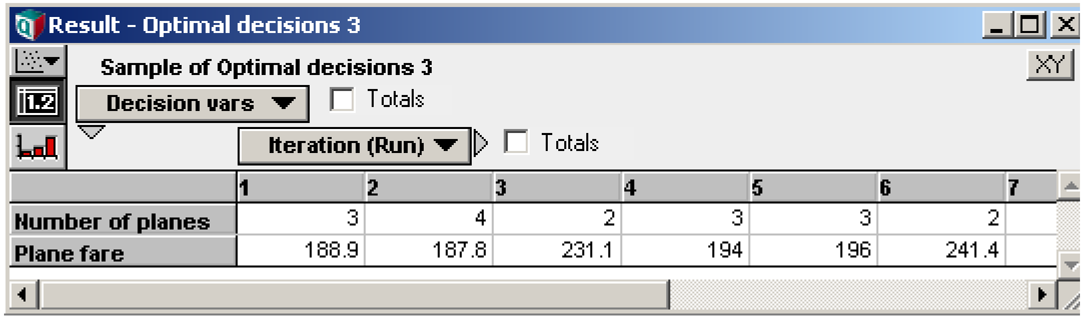
Iteration (Run) ☐ Totals

1	«NLP»
2	«NLP»
3	«NLP»
4	«NLP»
5	«NLP»

When we show the result of the resulting optimal decisions

```
Decision Optimal_decisions_3:=LPSolution(Nlp_3)
```

it evaluates each sample of the NLP and generate a corresponding sample of optimal decisions:



	1	2	3	4	5	6	7
Number of planes	3	4	2	3	3	2	
Plane fare	188.9	187.8	231.1	194	196	241.4	

This computation involves doing **Samplesize** optimizations. So, it could take a long time if the NLP problem is difficult and the sample size is large.

Use of SetContext

The uncertainty in the objective comes from the uncertainty in **Seats_per_plane**, which is uncertain as a result of its uncertain inputs **Base_demand** and **Elasticity1**. When we use the value of **Seats_per_plane**, we can slice out only the sample corresponding to the current optimization. To do this, we set up a context variable:

```
Run_context := if IsSampleEvalMode then Run else 1
domain of Run_context := Index Run
```

Next we modify our NlpDefine() call, specifying Run_context as a context for the optimization:

```
Nlp_3b := NlpDefine(..., SetContext:Run_context )
```

When **Nlp_3b** is viewed in Sample mode, a separate NLP appears for each element of the Run index. When the NLP corresponding to **Run=3** is evaluating, **Run_context** is set to 1 during that search. Finally, we must slice out a single element of the sample for Seats_per_plane. We do this by modifying the definition of **Profit** (in the example, a copy of Profit as been placed in this module named **Profit_in_context**):

```
Profit_in_context :=
    Seats_sold[Run=Run_context] * (Fare - Var_cost)
        - Num_planes * Fixed_cost
```

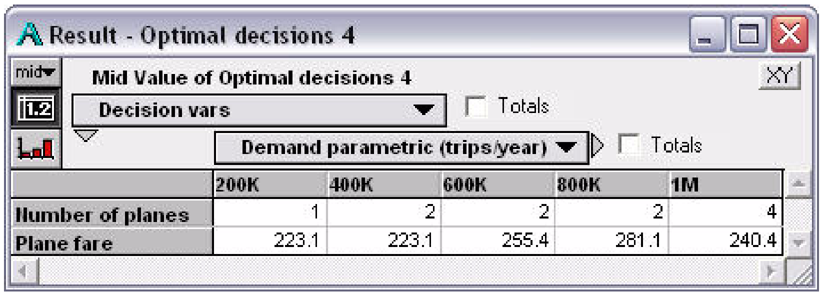
The choice was made here to slice on the current ***RunContext*** at **Profit**, rather than within **Seats_per_plane**, where each of **Base_demand** and **Elasticity1** could have been sliced. A speed advantage can be obtained by using your context as late in the computation as possible. In this way, **Seats_sold** is computed once, and does not have to be re-evaluated at each point in the search space.

Case 4. NLP and parametric analysis

What if you want to examine how the optimal decisions vary as you change one or more input parameters, such as **Demand**? (See *User Guide* Chapter 4 "Analyzing Model Behavior" for more on parametric analysis.) In this case, the variables you treat parametrically will have multiple values, so you cannot apply **NLPDefine()** to them directly. However, the function **Airline_nlp()** that we just defined comes in handy again. Suppose we define:

```
Variable Demand_param :=[200K,400K,600K,800K,1M]
Variable NLP_4 := Airline_nlp( Num_planes, Fare,
        Demand_param, Elasticity1)
    Decision Optimal_decisions4 := LPSolution(Nlp_4)
```

Because **`Airline_nlp()`** qualifies its parameters as **`Atomic`**, **`NLP_4`** generates an array of NLPs, one for each value of **`Demand_param`**. The Result for **`Optimal_decisions4`** shows corresponding optimal values for each value of **`Demand_param`**:



Mid Value of Optimal decisions 4

Decision vars / Demand parametric (trips/year)

	200K	400K	600K	800K	1M
Number of planes	1	2	2	2	4
Plane fare	223.1	223.1	255.4	281.1	240.4

Note how the optimal number of planes increases from 1 to 4, as the demand increases, and the optimal fare varies nonmonotonically.

Once again, the parametric analysis can also be accomplished using the ***SetContext*** parameter:

```
    NlpDefine(Vars: Dvars, X: Decisions, Ctype : Dec_type,
        Obj: Profit_4b, Maximize : True,
        Lb : Lb_decisions, Ub : Ub_decisions,
        SetContext:Demand_param)
```

Case 5. NLP over time using NPV

The most common formulations for optimization over time involve finding a set of decisions to optimize an objective that measures overall performance over multiple time periods, such as the net present value (NPV). In these cases, the objective function returns a single number that aggregates over the time periods, so it poses no problem for direct application of **`NLPDefine()`**.

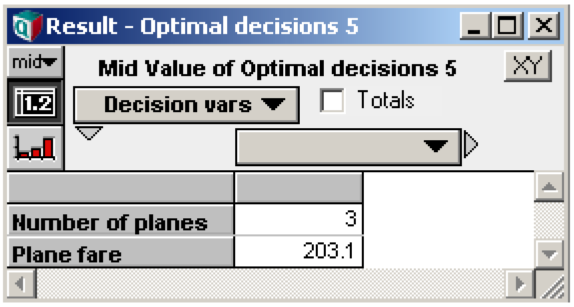
Consider the airline example again. We add an uncertain annual compound growth in demand, define **`Time`** for years from 2005 to 2010, and compute the resulting **`Demand_by_time`**:

```
    Chance Demand_growth_rate := Triangular(0%, 10%, 20%)
    Time := 2005 .. 2010
    Variable Demand_by_year := Dynamic(Base_demand,
            Self[Time-1] * (1 + Demand_growth))
```

We now define the objective of the NLP using mean of the net present value (NPV):

```
    Variable Nlp_5 :=
        NLPDefine(Vars: Dvars, X: Decisions, CType:Dec_type,
            Obj: Mean(NPV(Discount_rate,
                ProfitFn( Num_planes, Fare,
                Demand_by_year, Elasticity1), Time)),
            Maximize:true,
            Lb:Lb_decisions, Ub:Ub_decisions)
```

This causes no array-abstraction issues for the objective since the mean of the NPV is a scalar. Notice that we are finding a single optimal value for the decisions, **`Num_planes`** and **`Fare`**, for all time periods: We are assuming that these decisions stay the same over the six years. Because of the growth in demand, the optimal number of planes is three, larger than before:



Mid Value of Optimal decisions 5

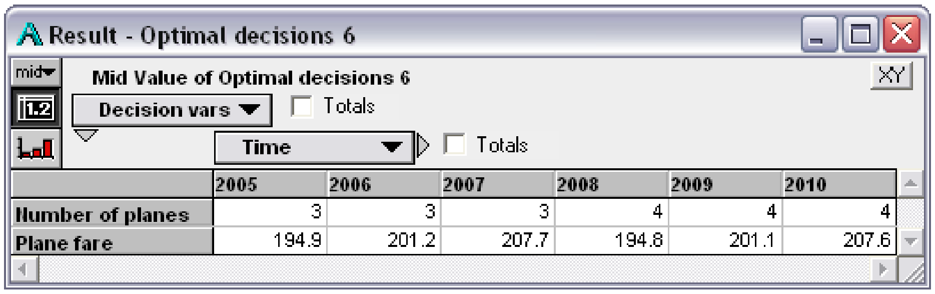
Number of planes	3
Plane fare	203.1

Case 6. Optimize for each year

What if you want to change the decisions, **`Num_planes`** and **`Fare`**, in each time period? One approach is to perform a separate optimization in each time period. This formulation models a process in which the decisions are made at the start of each time period to maximize profit for that time period. In this case, the decisions and objectives (and possibly constraints) are indexed by time. Again, the function **`Airline_NLP()`**, which we defined earlier, comes in handy.

```
Variable Nlp_6 := Airline_nlp( Num_Planes, Fare,
          Demand_by_year, Elasticity1 )
Decision Optimal_decisions4 := LPSolution(Nlp_4)
```

Since **`Demand_by_year`** is indexed by **`Time`**, **`Airline_nlp()`** creates an array of NLPs over time. The optimal decisions4 are then computed separately for each year:



Mid Value of Optimal decisions 6

	2005	2006	2007	2008	2009	2010
Number of planes	3	3	3	4	4	4
Plane fare	194.9	201.2	207.7	194.8	201.1	207.6

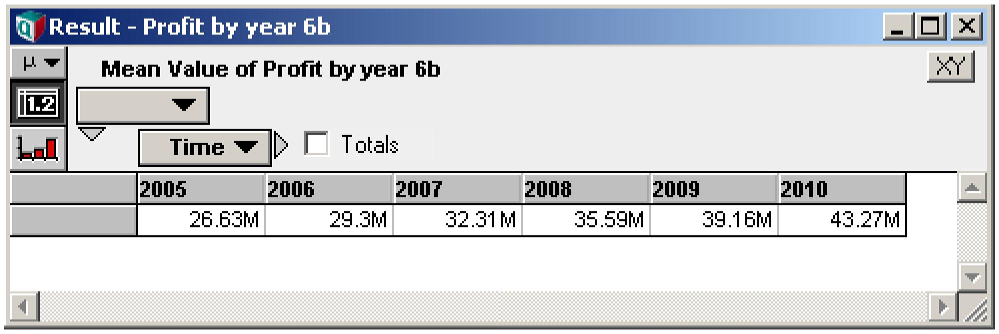
`Demand_by_year` has uncertainty. Since we have formed the objective given the value for **`Demand_by_year`**, in the probabilistic result we have actually created a separate optimization problem for each Monte Carlo sample at each time period. In terms of the problem, this is saying that we are uncertain today exactly what the demand will be in future years, but we will know the demand each year before we make our decision. Hence, we are array abstracting across two dimensions, Run and Time.

In the alternative formulation, ***`SetContext`*** can also be used in this case to array abstract across both Run and Time, as demonstrated in **`NLP_6b`**. In this case, we need to introduce two context variables, **`Run_context`** (already introduced in Case 3) and **`Time_context`**. When computing the objective profit, we need to restrict our inputs to these contexts:

```
Profit_in_context6 := ProfitFn( Num_planes,
        Demand_by_year[Time=Time_context, Run=Run_context],
        Elasticity1[Run=Run_context] )
```

Because we are array-abstracting across two contexts, both appear in the ***SetContext*** parameter:

```
NLPDefine(Vars: Dvars, X: Decisions, Ctype : Dec_type,
    Obj: Profit_in_context6,Maximize : True,
    LB : Lb_decisions, UB : Ub_decisions,
    SetContext: Run_Context, Time_Context)
```



	2005	2006	2007	2008	2009	2010
	26.63M	29.3M	32.31M	35.59M	39.16M	43.27M

Case 7. NLP with Optimizations over time

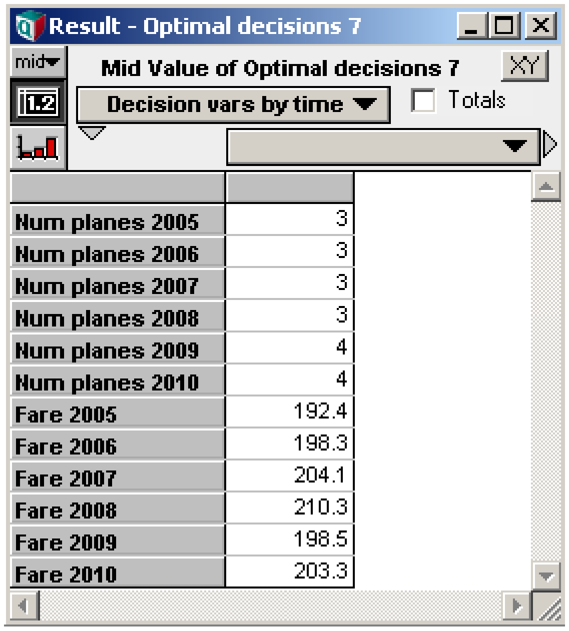
If there are interactions between decisions in different years, you may want to find the decisions in each year that collectively maximize the NPV (or other objective that aggregates over time). In this case, we want to perform only one optimization, but with an expanded set of decisions, that comprises both decisions over all time period. With 2 decisions in each of 6 time periods, we define a `Decisions` vector of 12 elements. Note that Decisions must be a one-dimensional vector with 12 elements, not a two-dimensional table with 2 by 6 elements.

In this case, we choose to create a single table with the decision settings -- initial values, Ctype, lower and upper bounds, for all 12 elements:

We derive the `Decisions_by_time` as a slice of this table:

```
Decision Decisions_by_time :=
    Decision_params[Decision_settings='Initial']
```

See the module in the example model for details of how the NLP is defined. We have a single optimization problem here with no array abstraction considerations. Here are sample results for the optimal decisions:



Mid Value of Optimal decisions 7

Decision vars by time

Num planes 2005	3
Num planes 2006	3
Num planes 2007	3
Num planes 2008	3
Num planes 2009	4
Num planes 2010	4
Fare 2005	192.4
Fare 2006	198.3
Fare 2007	204.1
Fare 2008	210.3
Fare 2009	198.5
Fare 2010	203.3

The time to perform NLP optimization typically increases superlinearly with the number of decision variables. So this approach can become time consuming if you have many decision variables and time periods. In general, it takes longer than **Case 6. Optimize for each year**, which is linear in the number of time periods.

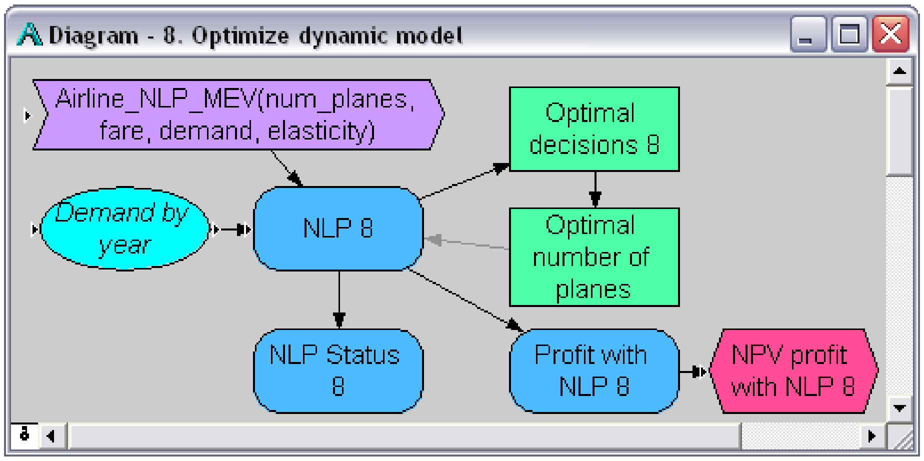
Case 8. NLP with a dynamic model

The previous three cases are dynamic in the sense that the model changes over time. However, they do not need to use the **`Dynamic()`** function explicitly because the decisions in each year do not depend on the results of the previous year. In this final case, the optimization at each time step depends on the results of the optimization at the previous time step, so we *must* use **`Dynamic()`**: We assume that the planes are on long-term leases: We can lease more planes each year, but cannot decrease them because of the lease agreement. This means that the lower bound on the number of planes decision in each period is the value of the optimal number of planes computed in the previous time period, thus:

```
Variable Nlp_8 := Dynamic(
    Airline_nlp_mev(Num_planes, Fare, Demand,
        Elasticity1),
    Airline_nlp_mev(Optimal_num_planes[Time-1],
        Fare, Demand_by_year, Elasticity1 ))
Decision Optimal_decisions_8:= LPSolution(Nlp_8)
Decision Optimal_num_planes :=
    Optimal_decisions_8[Dvars = 'Number of planes']
```

Note that this creates a dynamic loop, with the time lagged dependence shown in the diagram in gray:

When embedding an NLP inside a dynamic loop, as a general principle, only a local variable should ever be used for your decision variable, ***X***, and ***SetContext*** should not be used. If you were to use a global variable for ***X*** or ***SetContext***, each optimization would end up



invalidating the results of optimizations at other time steps in the dynamic loop. However, you can include your decision variables as global decision variables in your dynamic loop, defining them using **`LpSolution()`**, but using a local variable for the decision vector passed to parameter **`X`** of **`NlpDefine()`**. This is demonstrated in **`Nlp_8b`**. The expressions for ***Obj*** and ***Lhs*** may contain time offsets (in this case they do not). Keep in mind that **`Self[Time-1]`** in one of these expressions refers to an NLP object, not the previous value of the ***Obj*** or ***Lhs*** expression. To use the optimum objective value in the previous time step, you would use **`LpOpt(Self[Time-1])`**, for example. A portion of the definition of Nlp_8b is as follows:

```
Variable Nlp_8b := Dynamic(
   Var d[DVars] := Array(Dvars,[num_planes,fare]) do
      NlpDefine( Vars: DVars, X:d, ... ),
   Var d[DVars] := Array[Dvars,[num_planes8b[Time-1],far]) do
      NlpDefine( Vars: DVars, X:d, ...
         Lb: if DVars='Number of planes' Then Num_planes8b[Time-1]
                                           Else Lb_decisions )
   )
Variable Num_planes8b := LpSolution(Nlp_8b)[DVars='Number of planes']
```

The critical thing to notice here is that the decision variables appearing in the dynamic loop, in this case **`Num_planes8b`**, are defined using **`LpSolution`**. The expressions at **`Time=t`** can make use of the optimal solutions at **`Time=t-1`** in the definition of **`NlpDefine()`**, in its parameters, or in the expressions. The decision vector passed to **`X`** must be a local variable, and **`SetContext`** cannot be used.

Summary of array abstraction for NLP

These airline problem Cases 1 to 8 shown above illustrate ways to reformulate a problem for NLP to deal with various issues of array abstraction and Intelligent Arrays. Case 1 shows how to combine multiple scalar decisions into a single vector of decisions, as needed for **`NlpDefine()`**. Case 7 shows how to assemble array-valued decisions into a single vector of decisions. Case 2 shows that you require no special reformulation for NLP to maximizes expected value (or other statistical function of an uncertain objective), since the objective is a scalar, even if the underlying model has uncertainty. Similarly, Cases 5 and 7 illustrate that maximizing the net present value (or another objective that aggregates over time) produces a scalar value for the objective, so you can apply **`NlpDefine()`** directly.

In the other cases, the objective is intrinsically an array of values, indexed by **`Run`** for uncertainty in Case 3, by a parametric analysis (**`Demand`**) in Case 4, and by **`Time`** in Cases 6 and 8.

We handle these cases in a similar way: We encapsulate the **`NLPDefine()`** in a function whose parameters are qualified as **`Atomic`**, so that each call to **`NLPDefine()`** is made with the required inputs and hence the Objective passed to ***X*** as scalar. The result of calling these functions is an array of NLPs. Functions of this result, such as the optimal decisions, **`LPsolution()`**, status, **`LPStatusText()`**, and optimal value, **`LPOpt()`**, are therefore similarly indexed by these extra dimensions. For Cases 3, 4, and 6, we've also shown an alternative approach, in which we utilize the ***SetContext*** parameter to **`NlpDefine()`** restrict the model to a restricted slice while the optimization search takes place. In Case 8, with dynamic dependence, we see that an NLP can be embedded in a dynamic loop, refering to previous time points, as long as a local variable is used for the ***X*** parameter and ***SetContext*** is not used. If dimensions other than **`Time`** must be abstracted over, with each NLP depending on the solution to an earlier one, then encapsulating the NLP in a user-defined function is the only viable option.

For more details, look at the example Analytica file that contains these cases:
`Example Models/Optimizer Examples/Airline NLP.ANA.`

These examples show how to deal with array abstraction for the objective ***Obj***. The same approach will work for the other parameters that are repeatedly evaluated during an optimization, i.e. ***LHS***, ***Gradient***, and ***Jacobian***. *All other parameters array-abstract automatically.*

Solving Systems of Equations

Solving a system of non-linear equations is a special case of a non-linear program. The set of solutions is the set of feasible points. The non-linear optimizer can be used to find a solution to a system of equations by encoding the system of equations as the set of constraints, using a ***Sense*** of "=". You can omit the objective function (the ***Obj*** parameter) if you simply care about finding any solution, or you can use the objective to express a preference among solutions when the system of equations has, or may have, multiple solutions.

Other examples

If you haven't already, you may find it useful to follow through the steps in the "Quick Start" section for creating a non-linear optimization model (see "A Non-linear Program" on page 14).

The **`Example Models/Optimizer Examples`** directory, installed with Analytica, contains several models demonstrating non-linear optimization. These models include:

- **`Asset Allocation.ana:`** A classic portfolio optimization problem, formulated in four ways. One formulation uses a linear objective with a quadratic constraint, which qualifies as a non-linear problem. Another formulation maximizes expected utility, thus demonstrating the use of stochastic simulation within a non-linear optimization. The other two formulations are quadratic programs.
- **`NLP with Jacobian.ana:`** A very simple non-linear program demonstrates the use of a gradient and Jacobian, as well as the use of a local variable for X.
- **`Optimal can dimensions.ana`**: The example is the one used in Chapter 2, "Quick Start," of this manual. The problem is to find the dimensions for a cylindrical can to hold a given volume using the minimum surface area.

- **`Solve using NLP.ana:`** A very simple example of using the non-linear optimizer to solve a non-linear system of equations.
- **`Problems with local optima.ana`**: Demonstrates several techniques for overcoming the problem of local optima in non-linear optimizations.
- **`Traveling salesman.ana`**: Demonstrates the use of grouped-integer decision variables.

Giving hints to help the Optimizer

The Optimizer tries to identify characteristics of your NLP problem so that it can choose the most efficient and reliable algorithms. In some cases, you can improve its performance by telling it things about the problem that it may not be able to figure out on its own.

Type of dependence

If the Optimizer knows that the objective has smooth nonlinear dependence on some or all of the decision variables, it can use much faster gradient-based algorithms than in the general case that allows discontinuous functions. You can provide this information using these two optional parameters to **`NlpDefine()`**.

```
objNl: Optional Text[Vars]
lhsNl: Optional Text[Vars, Constraints]
```

You should provide each of these parameters with one of these text values:

- "**`L`**": Linear or no dependence
- "**`Q`**": Quadratic dependence
- "**`N`**": Smooth non-linear dependence
- "**`D`**": Discontinuous

You can provide a single text value to each parameter, e.g., "**`N`**", to specify the same type of dependence for all decision variables and, to ***lhsNl***, for all constraints. Otherwise, if the type of dependence varies by variables and constraints, you will probably create a variable defined as an edit table indexed by **`Vars`** and **`Constraints`**, to specify each dependency type.

When the objective has linear, quadratic, or smooth non-linear dependence on continuous decision variables, the optimizer uses an efficient gradient-based search method. If it knows that the dependence is linear (and so has constant derivative), or quadratic (and so has a constant second derivative) it can drastically speed the search by reducing the number of re-evaluations of the objective. If one or more decision variables are discontinuous, the Optimizer uses a genetic (evolutionary) algorithm, in which multiple candidate solutions are maintained, and the search is performed by mutating and recombining members of the population based on a fitness metric.

If you do not indicate the type of dependence, the optimizer will assumes smooth non-linear, and the "GRG Nonlinear" engine is used. If any discontinuous dependence is indicated, the "Evolutionary" algorithm is selected. In some cases, you may find the "Evolutionary" engine performs better even on smooth non-linear problems, in which case you can force use of the Evolutionary engine using the parameter:

```
Engine: "Evolutionary"
```

Gradient and Jacobian Functions

If the decision variables are all continuous, you can speed up the optimizer considerably if you can give it an analytical expression for the gradient of the objective function and/or the Jacobian of the constraint left-hand sides. The gradient and Jacobian enable the Optimizer to avoid most re-evaluations of the objective and LHS expressions, respectively, which it uses estimate the partial derivatives based on small changes to each decision variable.

The ***gradient*** of the objective function is a vector indexed by ***Vars***, where each element is the partial derivative:

$$\frac{\partial}{\partial x_i} f(\grave{x})$$

where $f(\grave{x})$ is the objective function.

The ***Jacobian*** of the left-hand side of the constraints is a matrix, indexed by ***Vars*** and ***Constraints***, where each element is the partial derivative:

$$\frac{\partial}{\partial x_i} g_j(\grave{x})$$

where $g_j(\grave{x})$ is the left-hand side of constraint *j*.

The ***gradient*** and ***Jacobian*** parameters accept an Analytica variable or expression, which should depend on ***X***, directly or indirectly. The Optimizer evaluates these parameters deterministically repeatedly at each step of the search process Assuming ***X*** is indexed only by ***Vars***, the ***gradient*** must be indexed only by ***Vars***, and the ***Jacobian*** must be indexed only by **Vars** and ***Constraints***. See "Intelligent Arrays, array abstraction and NLP" on page 54 for information on coping with these restrictions.

It is important for your ***gradient*** and ***Jacobian*** expressions to be correct, otherwise you will mislead the optimizer and it may move away from the optimum. Debugging a ***Jacobian*** expression can be challenging. However, you can check whether the Jacobian is correct using the optional parameter, ***DerivMethod***, to `NlpDefine()`:

```
NlpDefine(…, DerivMethod: "check", …)
```

When ***DerivMethod*** is set to `"check"`, the Optimizer compares the supplied ***Jacobian*** expression, with the Jacobian that it estimates using finite differencing. If they are not within a small difference, the Optimization will stop with `LpStatusNum`() = 67 ("error in evaluating problem functions"). Once you have confirmed the supplied Jacobian is correct, remember to reset **Derivmethod** to `"Jacobian"` so that the Optimizer reaps the benefits of not having to estimate the Jacobian itself at each search point.

Initial Guess

If you know the approximate region that contains the optimal solution, you can speed the Optimizer by giving it an initial solution in that region. You specify this starting solution as an array indexed by Vars for the optional parameter ***guess***:

```
guess: Optional Number[Vars]
```

If you do not provide this parameter, and if you provide a global variable (as opposed to a local variable) for ***X***, the Optimizer users the current value of ***X*** as its starting solution.

Dealing with Local Optima

A difficult problem common to many hard non-linear optimization problems is the existance of local optima. Once a local optima is reached, it is impossible for the optimizer to know where, or even if any, better solutions exist. If you think you are having problems with local optima, there are several settings that can be manipulated.

MultiStart

If you have a continuous non-linear problem, enabling the MultiStart setting of the "GRG Nonlinear" engine is often the quickest and easiest recourse. This can be tried quickly simply by adding the following parameters to **`NlpDefine()`**:

```
NlpDefine(..., parameter:"MultiStart", setting:1 )
```

Multistart will require more search time, as it tries multiple starting points. When using Multisearch, you should also specify finite lower and upper variable bounds using the ***Lb*** and ***Ub*** parameters to **`NlpDefine()`**. Narrow bounds produce better results.

If turning ***MultiStart*** on alone is inadequate, you can further enhance exploration by enabling topographic search, via the ***TopoSearch*** setting, which improves the selection of starting points, and by increasing the number of starting points by increasing ***Population Size***. See "Specifying Settings" in Chapter 5.

Engine Selection

Two non-linear optimization engines come with Analytica Optimizer:

- "GRG Nonlinear" : A gradient-descent search.
- "Evolutionary" : A genetic-algorithm search.

If you have purchased other add-on engines, other options may also be available to you. To explicitly select the engine to be used, include the ***Engine*** parameter to **`NlpDefine()`**:

```
Engine : Optional Text
```

If you have indicated that your problem is discontinuous, the GRG engine cannot be used.

By default, the Evolutionary engine does not utilize gradient information. However, if the ***LocalSearch*** setting is on, then it optimizes sample points before adding them to the population using various techniques including gradient-based search.

To view the list of possible engines installed, evaluate the following Analytica expression:

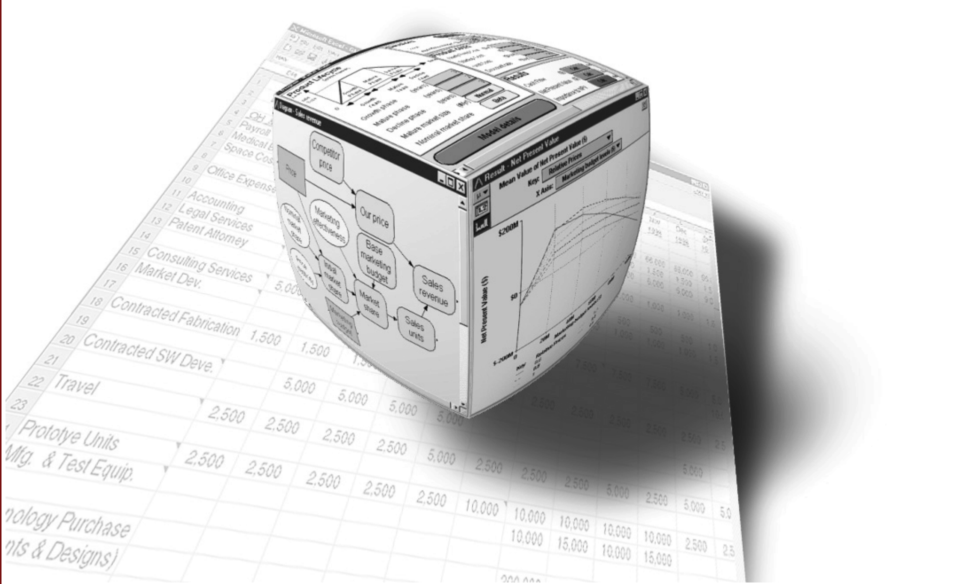
```
SolverInfo( "AvailEngines" )
```

If your problem is highly discontinuous or contains many local optima, then the "Evolutionary" engine is a better choice. If your problem is relatively smooth with relatively few local optima, then the "GRG Nonlinear" engine is likely to obtain results more quickly.

Chapter 7 *Control Settings*

This chapter shows you how to:

- specify optimizer engine settings to LpDefine, QpDefine and NlpDefine
- determine what setting are available for each engine, defaults, and possible range
- determine size capacities for installed engines.
- control termination criteria during optimization
- select search algorithms
- specify numeric precision



Controlling the search

The optimization engine exposes several settings that you can change to influence how the search for the optimum proceeds and when it terminates. The specific collection of settings that are available is a function of which engine is used to solve the optimization, so that if you install and use an add-on engine, other than the engine that comes standard with Analytica Optimizer, the possible settings may be different. The **`SolverInfo()`** function can be used to view all available settings, the range of possible values, their defaults, and their current values for a problem. Settings can be changed for a particular problem by specifying values for the ***parameter*** and ***settings*** parameters to **`LpDefine(), QpDefine(),`** or **`NlpDefine().`** The first subsection below describes how you specify and view settings, while the subsequent sub-sections detail particular settings used by engines the come standard with Analytica Optimizer.

Selecting the Optimization Engine

Four optimization engines come standard with Analytica Optimizer:

- "LP/Quadratic": Used for LPs and QPs with linear constraints.
- "SOCP Barrier": QPs with quadratic constraints.
- "GRG Nonlinear" : Smooth NLPs: A gradient-descent search
- "Evolutionary" : NLPs: A genetic-algorithm search.

If you have purchased other add-on engines, other options may also be available to you. You can obtain a full list of installed engines by evaluating the following Analytica expression:

```
SolverInfo( "AvailEngines" )
```

To explicitly select the engine to be used, include the ***Engine*** parameter to **`LpDefine()`**, **`QpDefine()`**, or **`NlpDefine()`**:

```
Engine : Optional Text
```

For example:

```
NlpDefine( ..., Engine: "Evolutionary" )
```

The following engines can be used with each function:

- **`LpDefine()`**: "LP/Quadratic", "SOCP Barrier", "GRG Nonlinear", "Evolutionary".
- **`QpDefine()`**: "LP/Quadratic" (but only if constraints are linear, i.e., parameter ***LhsQ*** not specified), ’SOCP Barrier", "GRG Nonlinear", "Evolutionary".
- **`NlpDefine( )`**: "GRG Nonlinear" (but only if any ***objNl*** or ***lhsNl*** has been marked ’D’ for discontinuous), "Evolutionary".

If you do not specify the engine, Analytica will select an appropriate engine based on the function you’ve used to define your problem, and the properties of the problem that you’ve specified. However, if the engine does not perform satisfactorily on that problem, you might obtain better results with a different engine.

To determine what engine is actually used on a problem, evaluate the Analytica expression:

```
SolverInfo( "Engine", prob )
```

where **`prob`** is the object returned by **`LpDefine()`**, **`QpDefine()`**, or **`NlpDefine()`**.

The "LP/Quadratic" engine uses a dual simplex method combined with branch-and-bound for mixed-integer constraints, with a variety of integer cut-set procedures. This is generally the engine of choice for LPs and Mixed-integer LPs. However, for hard mixed-integer LPs, since the Evolutionary engine utilizes a very different approach, that engine may be worth trying.

SOCP uses a second-order cone programming technique designed specifically for quadratically-constrained convex problems. The GRG Nonlinear engine is often a good alternative for problems formulated with **`QpDefine()`**, especially if the constraints end up being non-convex.

For non-linear problems, if your problem is highly discontinuous or contains many local optima, then the "Evolutionary" engine is a better choice. If your problem is relatively smooth with relatively few local optima, then the "GRG Nonlinear" engine is likely to obtain results more quickly, and if gradients and jacobians can be analytically computed, it is likely to be dramatically faster. If non-integer values cannot be explored during the intermediate steps of a search, the "Evolutionary" engine should be used.

By default, the Evolutionary engine does not utilize gradient information. However, if the ***LocalSearch*** setting is on, then it optimizes sample points before adding them to the population using various techniques including gradient-based search.

Examining Engine Capabilities

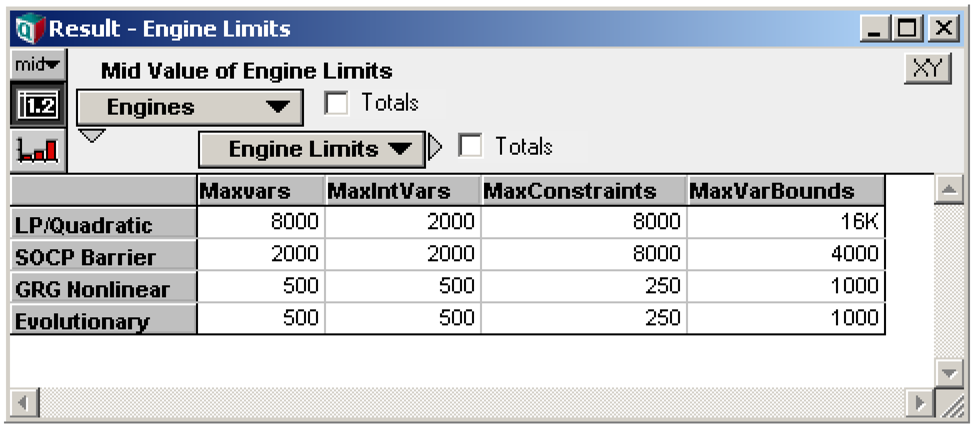
Information about an installed optimization such as the maximum number of variables or constraints allowed can be accessed using:

```
SolverInfo( Item: "<item>", Engine: <engineName> )
```

where ***engineName*** is a value returned by **`SolverInfo("AvailEngines")`**, and item is one of **`"MaxVars", "MaxIntVars", "MaxConstraints"`**, or **`"MaxVarBounds"`**. These return a result indexed by a local index named ***ProblemType***, having elements ["LP", "QP", "QCP", "CVX", "NLP", "NSP"]. (QP=quadratic objective, linear constraints, QCP=Quadratic with convex quadratic constraints, CVX=non-convex quadratic, NLP=smooth non-linear, NSP=non-smooth non-linear). For example:

```
Index Engines := SolverInfo("AvailEngines");
SolverInfo( ["Maxvars", "MaxIntVars", "MaxConstraints", "MaxVarBounds"],
            Engine: Engines) [.ProblemType='LP']
```

returns:



Mid Value of Engine Limits

	Maxvars	MaxIntVars	MaxConstraints	MaxVarBounds
LP/Quadratic	8000	2000	8000	16K
SOCP Barrier	2000	2000	8000	4000
GRG Nonlinear	500	500	250	1000
Evolutionary	500	500	250	1000

Specifying Settings

If you want to change the value for a single control setting, you can specify values for two optional parameters, ***parameter*** and ***setting***, to **LpDefine( )**, **`QpDefine()`** or **`NlpDefine()`**, providing the name of the setting to ***parameter***, and the value to ***setting***. For example, if you wished to set the ***"Scaling"*** parameter to 1, you would modify your call to **LpDefine( )** as follows:

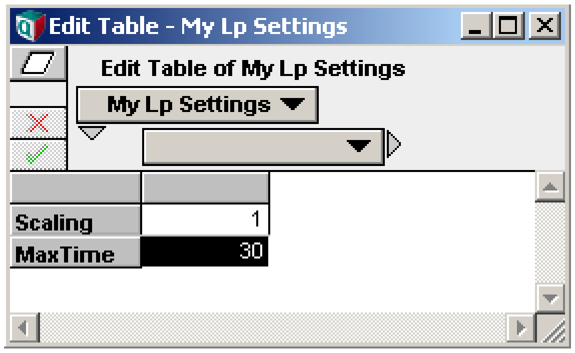
```
LpDefine( .., Parameter: "Scaling", Setting: 1 )
```

To alter more than one control setting, you need to supply arrays to these parameters. The arrays passed to ***parameter*** and ***setting*** should have a single common index. If the index of the array passed to ***setting*** is a list of labels, where the index labels contain the name of each control setting, then you only need to include the ***setting*** parameter.

It is often convenient to specify control settings in a self-indexed edit table. The following steps illustrate this:

1. Drag a variable node to your diagram, title it "*My Lp Settings*".
2. In the definition pane, set the definition type to **Table**.
3. In the Index Chooser dialog, select "My Lp Settings (Self)" as the table index.
4. Click on the row heading cell, and change "Item 1" to "Scaling".
5. With the row header still selected, press down-arrow to add a row.
6. Change the second row header cell to "MaxTime"
7. Enter 1 into the first table body cell.

8. Enter 30 into the second body table cell.



Scaling	1
MaxTime	30

9. In your call to **LpDefine( )**, insert a setting parameter as follows:

```
LpDefine( ..., setting: My_lp_settings )
```

The optimizer will scale parameters and terminate after 30 seconds if the optimum has not been found. A self-indexed table set up in this fashion makes it easy to adjust multiple control settings if the need arises.

Note: *In Analytica Optimizer 3.1, control settings were specified as optional parameters to* **`LpDefine( )`**, **`QpDefine()`** *and* **`NlpDefine()`**. *These legacy parameters are still supported for backward compatibility; however, use of the* ***setting*** *parameter is recommended. This change reflects a change in Frontline's architecture, and more readily generalizes to other add-on engines and future optimizer engine extensions.*

Examining Available Settings

Once you have defined an optimization problem using **LpDefine( )**, **`QpDefine()`** or **`NlpDefine()`**, the following function returns the set of control settings used by the engine:

```
SolverInfo("Setting", Lp: myLp )
```

where you replace ***myLp*** with the name of the variable holding the result from **LpDefine( )**.

You can also access the range of allowed values for each setting, as well as the default value, using **SolverInfo(**). For this, you need to know the name of the optimizer engine used on your problem. For linear programs, this will always be "LP/Quadratic" unless you have installed an add-on engine. To obtain the name of the engine used in the general case, use:

```
SolverInfo("Engine", Lp: myLp )
```

Tip For quadratic and non-linear problems, to be certain you get the correct engine, evaluate **SolverInfo("Engine", Lp:*myLp*)** after you've attempted to find a solution -- after **LpSolution( )**, **LpStatusText( )**, or **LpOpt( )** has been evaluated. If you have not specified the ***engine*** parameter explicitly, the Optimizer may change to a different, non-default engine based on the properties of your problem.

Using the name of the engine, the range (min/max) of possible values for each setting, and the default value, can be obtained using:

```
SolverInfo( ["MinSetting","MaxSetting","Defaults"],
Engine: "LP/Quadratic")
```

Termination Controls

***Iterations*:**

Specifies the maximum number of iterations (pivots) by the Simplex Algorithm during the optimization. If this is exceeded, **`LpStatusNum()`** returns 3 (Iterates limit reached. Indicates an early exit of the algorithm). Maximum number of generations in Evolutionary solver. Maximum number of gradient descent steps by GRG Nonlinear.

Default: no limit.

***MaxTime*:**

Maximum number of seconds the optimizer will spend on the problem. If exceeded, **`LpStatusNum()`** will be 10 (Time out status. Returned when the maximum allowed time has been exceeded. Indicates an early exit of the algorithm).

Default: no limit.

MaxTimeNoImp:

The maximum number of seconds that the Optimizer will continue without finding any improvement in the best solution.

Default: 30 seconds.

Allowed range: Positive

***IntTolerance*:**

In a MIP optimization, if the branch-and-bound algorithm can determine that the best solution found so far is within this relative tolerance of the true optimal, it will terminate the search and return the best solution found so far. The bound is relative, meaning a value of 10% guarantees a solution within 10% of the optimal. Often, the branch-and-bound algorithm will quickly locate a nearly optimal solution, but then spend a large amount of refining its best solution to the true optimum. Specifying a non-zero gap tolerance can eliminate this additional search, thus in some cases drastically reducing computation time. The gap is computed as the absolute value of the difference between the best solution so far, and the best bound on the optimum, divided by the best bound on the optimum. With zero gap (default), the search will continue until the entire search space is eliminated so that the global optimum is reached.

Default: 0%

Allowed range: 0 to 1

Convergence

The evolutionary solver will stop with status "Solver has converged to the current solution" when nearly all members in the current population have very similar fitness values. This stopping criteria is satisfied when 99% of the population members all have fitness values within *Convergence* tolerance of each other.

The fitness value is a combination of the objective function value and a penalty for constraints still violated. If you think the evolutionary solver is terminating too quickly, you can make this tolerance smaller, but you may also want to increase ***MutationRate*** or ***PopulationSize*** in order to increase the diversity of trial solutions.

Default: 10^{-4}

Allowed range: 0 or 1

Tolerance

If the relative (i.e., percentage) improvement observed during the previous ***MaxTimeNoImp*** seconds does not exceed this value, then evolutionary solver will terminate. See ***MaxTimeNoImp***.

Default: 0

Allowed range: 0 to 1

MaxTimeNoImp

Controls the amount of time (in seconds) that the evolutionary solver is willing to spend without making any significant progress. If the relative improvement during this time has not exceeded the setting specified by ***Tolerance***, it terminates with status (Solver cannot improve the current solution) or (Solver could not find a feasible solution).

Default: 10^{-5}

Allowed range: 10^{-9} to 10^{-4}

MaxSubProblems:

Maximum number of subproblems explored by Evolutionary algorithm before terminating.

Default: no limit

Allowed range: Positive

MaxFeasibleSolutions

The maximum number of feasible solutions found by the Evolutionary algorithm before terminating.

Default: no limit

Allowed range: Positive

MaxIntegerSols

The optimizer will terminate after this many feasible solutions have been found by the branch & bound algorithm.

Default: no limit

Allowed range: Positive

Algorithm Selection

Preprocessing

Scaling

When this is **True,** the optimizer will attempt to rescale decision variables and constraints internally for the Simplex algorithm, which usually leads to be reliable results and fewer iterations. A poorly scaled model, in which values of the objective, constraints, or intermediate results differ by several orders of magnitude, may result in numeric instabilities within the optimizer when scaling is turned off, due to the effects of finite precision computer arithmetic.

Default: False

Allowed range: 0 or 1

Presolve

When this is **True**, the LP/Quadratic engine performs a presolve step to detect singleton rows and columns, remove fixed variables and redundant constraints, and tighten bounds, prior to applying the Simplex method.

Default: True

Allowed range: 0 or 1

Engine: "LP/Quadratic"

PreProcess

Turns on or off all integer pre-processing. (on by default).

Default: 1

Allowed range: 0 or 1

Engine: "LP/Quadratic"

StrongBranching

This setting applies to integer and mixed-integer problems. When this is on, the optimizer estimates the impact of branching on each integer variable of the objective function prior to beginning the branch and bound search. It does this by performing a few iterations of the Dual Simplex method after fixing each variable. This "experiment" provides the search with an estimate of which integer variables are likely to be most effective choices during the branch and bound search. Although the time spent in this estimation process may be moderately expensive, the cost is often regained many times over through a reduction in the number of branch-and-bound iterations that must be explored to find an optimal integer solution.

Default: 1

Allowed range: 0 or 1

Engine: "LP/Quadratic"

Debugging

SolveWithout

When this is **True**, any integer (***ctype***) constraints are ignored, and the continuous, and the continuous version of the problem is solved instead. The effect is the same as changing the ***ctype*** parameter to ’C’, but may be more convenient in some cases when debugging.

Default: True

Allowed range: 0 or 1

IISBounds

Determines whether vraiable bounds should be included in the infeasibility search conducted by **`LpFindIIS()`** or **`LpWriteIIS()`**. When set to 1, only a subset of constraints along the ***Constraints*** index is considered. When set to 0, variable bounds may be eliminated in order to find an IIS with a greater number of constraints. This parameter is only used by **`FindIIS()`** when the second optional parameter, ***newLp***, is true. When ***newLp*** is true, **`FindIIS()`** returns a new LP object, from which you can use **`SolverInfo()`** to access the list of constraints and list of variable bounds present in the IIS. When ***newLp*** is false, since only a subset of hte constraints index is returned, **`LpFindIIS()`** relaxes only constraints, leaving variable bounds in tact.

Default: 0

Allowed range: 0 or 1

Numeric Estimation

***Derivatives*:**

The ***Derivatives*** setting controls how derivatives are computed. These values are possible:

- **`1 = forward`**: This is the default if Jacobian and gradient parameters are not supplied. The optimizer estimates derivatives using forward differencing, i.e., $\frac{\partial}{\partial(x)} \approx \frac{f(x+\Delta)-f(x)}{\Delta}$
- **`2 = central`**: The optimizer estimates derivatives using central differencing, i.e., $\frac{\partial}{\partial x} \approx \frac{f(x+\Delta)-f(x-\Delta)}{2\Delta}$
- **`3 = jacobian`**: The optimizer computes derivatives using the supplied Jacobian and gradient expressions. This is the default if these are supplied.
- **`4 = check`**: The optimizer computes derivatives using the supplied Jacobian expression and also estimates the Jacobian using finite differencing. If they don't agree to within a small tolerance, the optimization aborts with **`LpStatusNum()`** = 67 ("error in evaluating problem functions"). This option is useful for testing whether the Jacobian is accurate.

***StepSize*:**

The step size used to estimate derivatives numerically. This is the Δ value in the estimates listed in the preceding ***Derivatives*** description.

Default: 10^{-6}

Allowed range: 10^{-9} to 10^{-4}

SearchOption:

Controls how the gradient-based search determines the next point to jump to during search:

- **`0 = Newton`**: Uses a quasi-Newton method, maintaining an approximate Hessian matrix for the reduced gradient function.
- **`1 = Conjugate-gradient`**: Use a conjugate gradient method, which does not require the Hessian.

Default: 0

Allowed range: 0 or 1

Estimates:

The **`Estimates`** setting controls the method used to estimate the initial values for the basic decision variables at the beginning of each one-dimensional line search:

- **`0 = linear`**: Uses linear-extrapolation from the line tangent to the reduced objective function.
- **`1 = quadratic`**: Extrapolates to the extrema of a quadratic fitted to the reduced objective at its current point.

Default: 0

Allowed range: 0 or 1

***RecognizeLinear*:**

When set to 1, the optimizer will attempt to detect automatically decision variables that influence the objective and constraints in a linear fashion. It can then save time by pre-computing partial derivatives for these variables for the rest of the search. This aggressive strategy can create problems when a dependence changes dramatically throughout the search space, particularly when a decision variable is near linear around the starting point, but the gradient changes elsewhere in the search space. When the solution is reached, the optimizer will recompute the derivatives and verify them against the assumed values. If they do not agree, the status text "The linearity conditions required by this solver engine are not satisfied" is returned.

Engine: "GRG Nonlinear"

Default: 0 (select default)

Allowed range: 0 or 1

SOCP Barrier Search

In addition to the many search control settings available of linear programs, covered in the previous chapter, a few additional settings can be used to control the search when solving quadratically constrained problems using the "SOCP Barrier" engine.

These parameters are set using the parameter and settings parameters to **`QpDefine()`**, as described for **`LpDefine()`** in the previous chapter.

***SearchDirection*:**

Controls the search direction on each iteration of the SOCP Barrier engine. The Power Class method is a technique with the long-step barrier algorithm leading to a polynomial complexity. The dual scaling method uses HKM (Helmberg, Kojima and Monteiro) dual scaling, in which a Newton direction is found from the linearization of a symmetrized version of the optimality conditions. Either of these may be further modified by a predictor-corrector term.

Default: 0 (off)

Allowed range: 1=Power class, 2=Power class with predictor-corrector, 3=dual scaling, or 4=dual scaling with predictor-corrector.

Engine: "SOCP Barrier"

***PowerIndex*:**

This parameter is used to select a particular search direction when the ***SearchDirection*** is set to 1 or 2.

Default: 1

Allowed range: Non-negative integer

Engine: "SOCP Barrier"

***StepSizeFactor*:**

The relative step size (between 0 and 1) that the SOCP Barrier engine may take towards the constraint boundary at each iteration.

Default: 0.99

Allowed range: 0.00 to 0.99

Engine: "SOCP Barrier"

***GapTolerance*:**

The SOCP Barrier Solver uses a primal-dual method that computes new objective values for the primal problem and the dual problem at each iteration. When the gap or difference between these two objective values is less than the Gap Tolerance, the SOCP Barrier Solver will stop and declare the current solution optimal.

Engine: "SOCP Barrier"

Default: 10^{-6}

Allowed range: 0 to 1

FeasibilityTolerance

The SOCP Barrier engine considers a solution feasible when the constraints are satisfied to within this relative tolerance.

Engine: "SOCP Barrier"

Default: 10^{-6}

Allowed range: 0 to 1

Evolutionary Search Controls

***PopulationSize*:**

Controls the population size of candidate solutions maintained by the "Evolutionary" engine, or the number of starting points for ***MultiStart*** in the "GRG Nonlinear" engine. ***MultiStart*** has a minimum population size of 10. If you specify 0, or any number smaller than 10, then the number of starting points used is 10 times the number of decision variables, but no more than 200.

Engine: "GRG Nonlinear", "Evolutionary"

Default: 0 (automatic)

Allowed range: 0, or integer >= 10

MutationRate

The probability that the evolutionary Optimizer engine, on one of its major iterations, will attempt to generate a new point by “mutating” or altering one or more decision variable values of a current point in the population of candidate solutions..

Engine: "Evolutionary"

Default: 0.075

Allowed range: 0 to 1

ExtinctionRate

This determines how often the Evolutionary engine throws out its entire population, except for the very best candidate solutions, and starts over from scratch.

Engine: "Evolutionary"

Default: 0.5

Allowed range: 0 to 1

***RandomSeed*:**

Both engines utilize a pseudo-random component in their search for an optima. As a result, the final result can differ each time an optimization of the exact same problem is performed. By setting the random seed, you can ensure that the same sequence of pseudo-random numbers is utilized, so that the same result obtains every time the same problem is re-evaluated. If you do not specify the random seed, Analytica uses its internal random seed, so that when you first load a model and evaluate results in a fixed order, you will get a predictable result. Setting RandomSeed to 0 causes the pseudo-random generated to be seeded using the system clock. Any positive value sets the initial seed to a fixed number.

Engine: "GRG Nonlinear", "Evolutionary"

Default: (use Analytica's random seed)

Allowed range: non-negative integer

Feasibility

When set to 1, the Evolutionary engine will throw out all infeasible points, and keep only feasible points in its population. When set to 0, it accepts feasible points in the population with a high penalty in the fitness score, which tends to be useful when it has a hard time finding feasible points.

Default: 0

Allowed range: 0 or 1

LocalSearch

Selects the local search strategy employed by the Evolutionary engine. In one step, or generation, of the algorithm, a possible mutation and a crossover occur, followed by a local search in some cases, followed by elimination of unfit members of the population. This parameter controls the method used for this local search. The decision for whether to apply a local search at a given generation is determined by two tests. First, the objective value for the starting point must exceed a certain threshold, and second, the point must be sufficiently far from any already identified local extrema. The threshold is based on the best objective found so far, but is adjusted dynamically as the search proceeds. The distance to local optima threshold is based on distance travelled previous times the local optima was reached.

There is a computational trade-off between the amount of time spent in local searches, versus the time spent in more global searches. The value of local searches depends on the nature of your problem. Roughly speaking, the Randomized method is the least expensive, the Gradient method tends to be the most expensive (i.e., with more time devoted to local searches rather than global search).

Engine: "Evolutionary"

Default: 0

Allowed Range: 0 to 3
1 = Randomized Local Search: Generates a small number of new trial points in the vicinity of the just-discovered "best" solution. Improved points are accepted into the population.
2 = Deterministic Pattern Search: Uses a deterministic "pattern search" method to seek improved points in the vicinity of the just-discovered "best" solution. Does not make use of the gradient, and so is effective for non-smooth functions.
3 = Gradient Local Search: Uses a quasi-Newton gradient descent search to locate an improved point to add to the population.

FixNonSmooth

Determines how non-smooth variables (see ***NlpDefine*** parameters ***objNl***=’D’ and ***lhsNl***=’D’) are handled during the local search step. If set, then only linear and nonlinear smooth variables are allowed to vary during the local search. Because gradients often exist at most points, even for discontinuous variables, leaving this off can still yield useful information in spite of the occasional invalid gradient.

Engine: "Evolutionary"

Default: 0

Allowed Range: 0 or 1

Mixed-Integer Controls

Integer Branch & Bound

IntCutoff

If you can correctly bound the objective function value for the optimal solution in advance, this can drastically reduce the computation time for MIP problems, since the branch-and-bound algorithm to prune entire branches from the search space without having to explore them at all. For a maximization problem, specify a lower bound, and for a minimization problem, specify an upper bound. If you specify this parameter, you need to be sure that there is an integer solution with an objective value at least this good, otherwise the optimizer might skip over, and thus never find, an optimal integer solution.

Default: no bounding

UseDual

When true, the LP/Quadratic engine uses the Dual Simplex method, starting from an advanced basis, to solve subproblems generated by the branch-and-bound method. When false, it uses the Primal Simplex method to solve subproblems. Use of Dual Simplex often speeds up the solution of mixed integer problems.

The subproblems of an integer programming problem are based on the relaxation of the problem, but have additional or tighter bounds on the variables. The solution of the relaxation (or of a more direct “parent” of the current sub problem) provides an “advanced basis” which can be used as a starting point for solving the current subproblem, potentially in fewer iterations. This basis may not be primal feasible due to the additional or tighter bounds on the variables, but it is always dual feasible. Because of this, the Dual Simplex method is usually faster than the Primal Simplex method when starting from an advanced basis.

Default: 2

Allowed range:
1 = Primal

2 = Dual

ProbingFeasibility

Probing is a pre-processing step during which the solver attempts to deduce the values for certain binary integer variables based on the settings of others, prior to actually solving a subproblem. While solving a mixed-integer problem, probing can be performed on each subproblem before running a constrained simplex. As branch-and-bound fixes one variable to a specific binary value, this may cause the values for other binary variables to become determined. In some cases, probing can identify infeasible subproblems even before solving them. In certain types of constraint satisfaction problems, probing can reduce the number of subproblems by orders of magnitude.

Default: 0

Allowed range: 0 or 1

BoundsImprovement

This strategy attempts to tighten bounds on variables that are not 0-1 or binary variables, based on values that have been derived for binary variables, before subproblems are solved.

Default: 0

Allowed range: 0 or 1

OptimalityFixing

This strategy attempts to fix the values of binary integer variables before each subproblem is solved, based on the signs of coefficients in the objective and constraints. As with ***BoundsImprovement*** and ***ProbingFeasibility***, this can result in faster pruning of branches by the branch & bound search; however, *in some cases Optimality Fixing can yield incorrect results*. Specifically, optimality fixing creates incorrect results when the set of inequalities imply an equality constraint. For example, if you have a situation where:

```
Lhs[ Constraints=1 ] = Lhs[ Constraints=2 ]
```

for all Vars, and your constraints are

```
Sum( Lhs[Constraint=1] * X, Vars ) <= 10
Sum( Lhs[Constraint=2] * X, Vars ) >= 10
```

this implies an =10 constraint. You must also watch out for more subtle implied equalities, such as where it is possible to deduce the value of a variable from the inequalities. Such equalities must be represented explicitly as equalities for OptimalityFixing to work correctly.

Default: 0

Allowed range: 0 or 1

PrimalHeuristic

This strategy attempts to discover a feasible integer solution early in the branch & bound process by using a heuristic method. The specific heuristic used by the LP Simplex solver is one that has been found to be quite effective in the "local search" literature, especially on 0-1 integer programming problems, but which not guaranteed to succeed in all cases in finding a feasible integer solution. If the heuristic method succeeds, branch & bound starts with a big advantage, allowing it to prune branches early. If the heuristic method fails, branch and bound begins as it normally would, but with no special advantage, and the time spent with the heuristic method is wasted.

Default: 0

Allowed range: 0 or 1

LocalHeur, RoundingHeur, LocalTree

These strategies look for possible integer solutions in the vicinity of known integer solution using a local heuristic ("local search heuristic" or "rounding heuristic"), adjusting the values of individual integer variables. As with the ***PrimalHeuristic***, finding an integer solution can help improve bounds used by the search, and thus prune off portions of the search tree.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

FeasibilityPump

An incumbant finding heuristic used by branch-and-bound to find good incumbants quickly.

Engine: "LP/Quadratic"

Default: 1

Allowed range: 0 or 1

GreedyCover

Another incumbant finding heuristic used by branch-and-bound to find good incumbants quickly.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

Cut Generation Control

Cut generation options are available for the LP Simplex method and is used when solving integer or mixed-integer LP problems.

A cut is an automatically generated constraint that "cuts off" some portion of the feasible region of an LP subproblem without eliminating any possible integer solutions. Many different cut methods are available each of which are capable of identifying different forms of constraints among integer variables that can be leveraged to quickly reduce the feasible set, and thus prune the branch-and-bound search tree. However, each of these methods require a certain amount of work in order to identify cut opportunities, so that when opportunities are not identified, that effort may be wasted. The defaults are set in ways that represent a reasonable trade-off for most problems, but for hard integer problems, you have the ability experiment with these to find the best settings for your own problem. You may find that some methods are more effective than others on your particular problem.

MaxRootCutPasses

Controls the maximum number of cut passes carried out immediately after the first LP relaxation is solved. This has an effect only if one of the cut method options is on. If this is set to a value of -1, the number of passes is determined automatically. The setting MaxTreeCutPasses is used for all iterations after the first.

Engine: "LP/Quadratic"

Default: -1 (automatically determined)

Allowed range: -1or more

MaxTreeCutPasses

Controls the maximum number of cut passes carried out at each step of the solution process with the exception of the first cycle. This setting is used only if at least one cut method is on. Each time a cut is added to a problem, this may produce further opportunities for additional cuts, hence cuts can continue to be added until no more cuts are possible, or until this maximum bound is reached.

Engine: "LP/Quadratic"

Default: 10

Allowed range: 0 or more

GomoryCuts

Gomory cuts are generated by examining the inverse basis of the optimum solution to a previous solved LP relaxation subproblem. The technique is sensitive to numeric rounding errors, so when used, it is important that your problem is well-scaled. It is recommended that you set the ***Scaling*** settings to 1 when using Gomory cuts.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

MaxGomoryCuts

This is the maximum gomory cuts that should be introduced into a given subproblem.

Default: 20

Allowed range: non-negative

GomoryPasses

The number of passes to make over a given subproblem looking for possible Gomory cuts. Each time you add a cut, this may present opportunities for new cuts. It is actually possible to solve an LP/MIP problem simply by making continual Gomory passes until the problem is solved, but typically this is less efficient than branch and bound. However, that may be different for different problems.

Default: 1

Allowed range: non-negative

KnapsackCuts

Knapsack cuts are only used with grouped-integer variables (whereas Gomory cuts can be used with any integer variable type). These are also called *lifted cover inequalities*. This setting controls whether Knapsack cuts are used.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

MaxKnapsackCuts

The maximum number of knapsack cuts to introduce into a given subproblem.

Default: 20

Allowed range: non-negative

KnapsackPasses

The number of passes the solver should make over a given subproblem, looking for knapsack cuts.

Default: 1

Allowed range: non-negative

ProbingCuts

Controls whether probing cuts are generated. 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.

Engine: "LP/Quadratic"

Default: 1

Allowed range: 0 or 1

OddHoleCuts

Controls whether Odd Hole Cuts (also called Odd Cycle cuts) are generated. This uses a method due to Grotschel, Lovasz and Schrijver that apply only to constraints that are sums of binary variables.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

MirCuts, TwoMirCuts

Mixed Integer Rounding Cuts and two-mixed integer Rounding Cuts.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

RedSplitCuts

Reduce and Split cuts are a variant of Gomory cuts.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

SOSCuts

Special Ordered Sets refer to constraints consisting of a sum of binary variables equal to 1. These arise common in certain types of problems. In these constraints, in any feasible solution exactly one of the variables in the constraint must be 1, and all the others zero, such that only n permutations need to be considered, rather than 2^n.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

FlowCoverCuts

Controls whether Flow Cover Cuts are used.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

CliqueCuts

Controls whether Clique cuts may be used, using a method due to Hoffman and Padberg. Both row clique cuts and start clique cuts are generated.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

RoundingCuts

A rounding cut is an inequality over all integer variables formed by removing any continuous variables, dividing through by the greatest common denominator of the coefficients, and rounding down the right hand side.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

LiftAndCoverCuts

Lift and Cover cuts are fairly expensive to compute, but when they can be generated, they are often very effective in cutting off portions of the LP feasible region, improving the speed of the solution process.

Engine: "LP/Quadratic"

Default: 0

Allowed range: 0 or 1

Coping with Local Optima

***MultiStart*:**

When turned on, the GRG engine restarts at multiple starting points, following the gradient from each to its corresponding local optima. Starting points are selected randomly between the specified lower and upper variable bounds, and clustered using a method called multi-level single linkage. The solver selects a representative point from each cluster, and then continues to successively smaller clusters based on the likelihood of capturing undiscovered local optima. Best results will be obtained from ***MultiStart*** when your variable upper and lower bounds are finite with as narrow range as possible. If finite bounds are not specified, you must set ***RequireBounds*** to 0. ***PopulationSize*** controls the number of starting points. ***TopoSearch*** can be set for a more sophisticated method of selecting starting points.

Engine: "GRG Nonlinear"

Default: 0 (off)

Allowed range: 0 or 1

***RequireBounds*:**

When ***MultiStart*** is used to select random starting positions, points between the bounds specified for each variable are sampled. If finite bounds on some variables are not specified, then ***MultiStart*** can still be used, but is likely to be less effective because starting value must be selected from an infinite range, which is unlikely to cover all possible starting points, and thus is unlikely to find all the local optima. When ***RequireBounds*** is on, as it is by default, an error results if you have not specified finite bounds on variables and have selected the ***MultiStart*** method, so as to remind you to specify bounds. If you really intend to use Multistart without finite bounds on the variables, you must explicitly set ***RequireBounds*** to 0.

When using the Evolutionary engine, finite bounds are also important in order to ensure a appropriate sampling for an initial population. Although it can still function without bounds, the infinite range that must be explored can dramatically slow down amount required to find a solution, and thus it is recommended that you always specify finite upper and lower bounds when using the Evolutionary engine. If RequireBounds is 1 (the default) when no bounds are specified, an error is reported in order to encourage the use of bounds.

Engine: "GRG Nonlinear", "Evolutionary"

Default: 1 (on)

Allowed range: 0 or 1

***TopoSearch*:**

TopoSearch is only used when ***MultiStart*** is 1. When set to 1, the multistart method uses a topographic search method that fits a topographic surface to all previously sampled starting points in order to estimate the location of hills and valleys in the search space. It then uses this information to find a better starting points. Estimating topography takes more computing time, but in some problems that can be more than offset from the improvements in each GRG search.

Engine: "GRG Nonlinear"

Default: 0 (off)

Allowed range: 0 or 1

Numeric Tolerance and Precision

ReducedTol

The Optimal or Reduced Cost Tolerance. The Simplex method looks for a variable to enter the basis that has a negative reduced cost. Decision variables whose reduced cost is less than the negative of this tolerance are candidates for entering the basis during the Simplex search.

Default: 10^{-5}

Allowed range: 10^{-9} to 10^{-4}

PivotTol

During the Simplex Algorithm, elements in the solution matrix must have an absolute value greater than this value to be candidates for pivoting.

Default: 10^{-5}

Allowed range: 10^{-9} to 10^{-4}

Precision

This value specifies how closely the calculated values on the left-hand side of constraints must match the right-hand sides in order for the constraint to be satisfied. Because of the finite precision arithmetic, a left-hand side that would ideally evaluate to 7.0 might compute as 6.9999999. With a Precision of 10^{-6}, the constraint A1 >= 7 would be considered satisfied in this case.

Default: 10^{-6}

Allowed range: 10^{-9} to 10^{-4}

PrimalTolerance

The maximum amount by which the constraints can be violated and still considered feasible.

Engine: "LP/Quadratic"

Default: 10^{-7}

Allowed range: 0 to 1

DualTolerance

The maximum amount by which the dual constraints and still considered feasible.

Engine: "LP/Quadratic"

Default: 10^{-7}

Allowed range: 0 to 1

Unused

There are a few optimizer settings that are not used by the standard engines in AnalyticaOptimizer, even though they do show up on the list of settings. Some of these are used by add-on engines (add-on engines have their own set of additional parameters in general).

Crashing

IntCutoffHigh, deprecated, used ***IntCutoff***

IntCutoffLow, deprecated, use ***IntCutoff***

PrecisionTol

SolutionAccuracy

SolutionResolution

SolutionTol

VariableReordering

Chapter 8 Debugging Optimization Problems

This chapter shows you how to:

- Write a linear or quadratic optimization formulation to a file for inspection
- Diagnose Conflicting Constraints
- Debug a Non-Linear Optimization

Debugging a Problem Formulation

Writing and Reading From a File

A linear or quadratic optimization formulation can be written to (and read from) a text file using the functions

LpWrite()
LpRead()

```
LpWrite(lp: LpType ; filename: Text )
LpRead(filename: Text)
```

`LpWrite()` returns the full filename path written to. **`LpRead()`** returns an ***<<LP>>*** or ***<<QP>>*** object. Viewing the resulting file can sometimes be useful for detecting problems with your call to **`LpDefine()`** or **`QpDefine()`**. These functions cannot be used on a non-linear optimization. The ***filename*** are interpreted relative to the current Analytica data directory.

Both functions accept an optional parameter:

```
format : optional text
```

which accepts the value of "LP", "MPS" or "LPFML". These are standard file formats used for exchanging problem specifications between other optimization software.

`LpRead()` also optionally accepts two indexes:

```
Vars,Constraints : optional Index
```

If specified, the length of each of these indexes must match the number of variables and number of constraints in the file being read exactly. When they are not specified, local indexes are created.

Diagnosing Conflicting Constraints

If you have conflicting constraints in your formulation, there will be no feasible solution. When you have many constraints, you can find the conflicting constraints by computing an ***Irreducibly Infeasible Subset*** (***IIS***) of constraints using one of the functions

LpFindIIS()
LpWriteIIS()

```
LpFindIIS(lp: LpType)
LpWriteIIS(lp: LpType; filename: Text)
```

An Irreducibly Infeasible Subset of constraints is a subset of your constraints which contains no feasible solution, but which has the property that if any single constraint is removed, there will be feasible solutions. Thus, it is a minimal set of conflicting constraints.

`LpFindIIS()` returns a subset of your ***Constraints*** index. This can be used on linear constraints defined from **`LpDefine()`** or **`QpDefine()`**.

`LpWriteIIS()` writes the IIS to an indicated file and returns the full file path. This function can be used with linear and quadratic optimizations with linear constraints, but not with non-linear optimization problems. The file format is the same as that used by **`LpWrite()`**. An optional ***format*** parameter of either "LP", "MPS" or "LPFML" can also be included.

When finding an IIS, there is an option of whether to only remove constraints, or whether variable bounds can also be removed in order to find an IIS. By removing variable bounds, it may be possible to find an IIS with a larger number of constraints. By default, **`LpFindIIS()`** removes only constraints, leaving variable bounds alone, which is necessary since a subset of the constraints index is returned. To allow variable bounds to be relaxed, you must include a second parameter to **`LpFindIIS()`**:

```
LpFindIIS( lp, newLp: true )
```

When this second parameter is included, **`LpFindIIS()`** returns a new <<LP>> object - the same type of object returned by **`LpDefine()`**. The new LP is still infeasible, but using it you can examine the reduced set of constraints and reduced set of variable lower and upper bounds using these expressions, where lp is the object returned by **`LpFindIIS()`**:

```
SolverInfo( "Constraints", lp )
SolverInfo( "lb", lp )
SolverInfo( "ub", lp )
```

By default, **`LpWriteIIS()`** relaxes both variable bounds and constraints. The setting "***IIS-Bounds***" can be used to override this behavior for both **`LpWriteIIS()`** and **`LpFindIIS(.., newLp:true)`**, see ***IISBounds*** in Chapter 7, “Control Settings”.

Debugging a Non-Linear Optimization

After formulating a non-linear problem, you may find that the optimization runs and returns something other than what you expect. After viewing the ***LpStatusText()***, it may not be clear why it terminated where it did, or why it didn’t succeed in solving your optimization as you desire. In these cases, you may need to monitor the optimization while it is searching in order to debug why it is doing what it is doing.

TraceFile

The optional parameter to **`NlpDefine()`**

```
TraceFile : optional Text
```

can be given a filename to write a log of all points visited during the optimization search. Written to the file are the values of the decision variables at each evaluation, the value computed for the objective, and the computed jacobian and gradient values if those expressions are also provided to **`NlpDefine()`**. The file can then be viewed in a text editor such as Text-Pad or NotePad. The values are tab-separated, so adjusting tab width in your text editor can help with readability. Often by studying how the search progressed, it will often become evident why the optimizer behaving as it is. Once this is understood, this may help to uncover errors in your problem formulation, or suggest approaches to improve the search.

Using MsgBox to Debug

Another "trick" that is often convenient is to simply peek at what values optimizer is plugging in for ***X*** in a more interactive fashion while the search is taking place. You can do this by inserting a **`MsgBox`** inside the expression that computes your objective (or in any node downstream of ***X*** and upstream of your objective expression). For example, if your objective expression is

```
obj: Sum(Exp(-a*x), Vars)
```

you might modify this to read:

```
obj: MsgBox(x,0,"X="): Sum(Exp(-a*x), Vars)
```

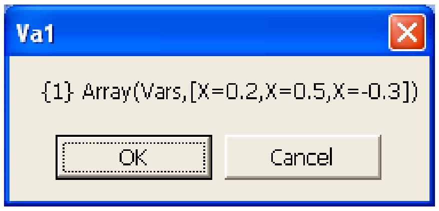
Then each time the optimizer evaluates the objective, a message box will appear on the screen, allowing you to view progress. Seeing the optimizer in action will often give you an understanding of what it would take to improve the search.

There are a few quirks to be aware of when using `MsgBox` in this fashion. First, the ***MaxTimeNoImp*** parameter specifies a maximum time the optimizer will work with no improvement in the best feasible solution found so far. Time spent staring at the message box will count towards time spent, and may result in an earlier termination. If this happens, you may want to explicitly set this parameter in your call to `NlpDefine()` to something large.

A second quirk is that if you decide to print out multiple pieces of information with a message box, you must consider how they will array abstract. `MsgBox()` prints out a description of your entire array result, but its parameter is evaluated before it even considers printing it. So, if you call `MsgBox()` using:

```
MsgBox("x=" & x)
```

when x is array-valued, you'll see something like:



{1} Array(Vars,[X=0.2,X=0.5,X=-0.3])

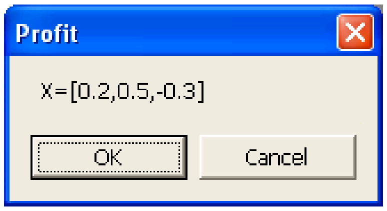
rather than

X=Array(Vars,[X=0.2,0.5,-0.3])

as you might have expected. If you plan on displaying multiple variables in the same message box, consider using expressions such as:

```
MsgBox("X=[" & JoinText(X, Vars,",") & "]")
```

which outputs:



X=[0.2,0.5,-0.3]

You can scatter `MsgBox()` calls throughout expressions to peek at the optimization at various points as it progresses.

Chapter 9 Optimizer Function Reference

This chapter lists and defines all the Analytica optimization functions.

Chapter 9

Optimization Function Reference

Problem Definition Functions

When defining an optimization problem, we highly recommend using a named-parameter syntax, rather than relying on the parameter being the first, second, or third parameter, etc. In a named-parameter syntax, you type the parameter name, followed by a colon (:), followed by the parameter value. For example:

```
NlpDefine( X:d, Obj: F(d) )
```

In the descriptions of **`LpDefine()`**, **`QpDefine()`** and **`NlpDefine()`** that follow, the parameters are shown in a logical grouping, but not in the actual order. We assume you will use named-parameter syntax. You can view the full parameter declarations from Analytica, in the actual parameter order, by selecting ***Definition --> Optimizer --> <function>*** from the Analytica menu which viewing an influence diagram with nothing selected.

LpDefine(*{ Decision variables }*

vars : Index,
lb, ub: optional Number[vars];
ctype: optional Text[vars] = ’C’;
group : optional Number[vars];
{ Objective function }
objCoef: Number[vars];
maximize: optional Boolean = false;
{ Constraint specification }
constraints: Index;
lhs: Number[vars,vonstraints];
sense: optional Text[constraints] = ’<=’ ;
rhs: Number[constraints] ;
{ Engine settings }
engine : optional Text ;
parameter : optional text ;
setting : optional number ;
{ deprecated }
ItLimit, NdLimit, MipLimit, TimeLimit: optional Positive;
optTolerance, pivotTolerance, feasTolerance, gapTolerance: optional Positive;
optLb,OptUb, scaling: Optional Numeric **)**

Defines a linear optimization program. See Chapter 4, “Linear Optimization”, for a description of usage and parameters, and Chapter 7, “Control Settings”, for possible engine settings.

QpDefine(*{ Decision Variables }*

vars, vars2 : Index ;
lb,ub: optional Number[vars];

ctype: optional Text[vars] = ’C’ ;
group : optional Number[vars]
guess: Optional Numeric[vars];
{ Objective Function }
c: Numeric[Vars]; Q: Numeric[vars,vars2];
maximize: Optional Boolean = false;
{ Constriant specification }
constraints: Index;
lhs: optional Number[vars,constraints];
lhsQ : optional Number[vars,vars2,constraints] ;
sense: Optional Text[constraints] = ’<=’ ;
rhs: Number[constraints];
{ Engine settings }
engine : optional Text ;
parameter : optional text ;
setting : optional number ;
{ Deprecated }
warnIndefinite: optional Boolean;
ItLimit, NdLimit, MipLimit, TimeLimit: optional Positive;
optTolerance, pivotTolerance, feasTolerance, gapTolerance: optional Positive;
***optLb,OptUb, scaling: optional Numeric*)**

Defines a quadratic optimization program. See Chapter 5, “Quadratic Optimization”, for a description of usage and parameters, and Chapter 7, “Control Settings”, for possible engine settings.

NlpDefine(*{ Decision Variables }*
vars: optional Index ;
x : LVarType ; *{ global or local variable }*
lb,ub: optional Number[vars] ;
ctype: optional Text[vars] = ’C’ ;
group : optional Number[vars] ;
guess: Optional Numeric[vars];
{ Objective Function }
obj : optional Expression { atomic };
maximize: optional Boolean = false ;
objNl : optional Text[vars] = ’N’ ;
gradient : optional Expression { vars };
{ Constraint specification }
constraints: optional Index ;
lhs : optional Expression { constraints };
sense : optional Text [constraints] = ’<=’ ;
rhs : optional Number[constraints] = 0 ;
lhsNl = optional Text [vars, constraints] = ’N’ ;
jacobian : optional Expression { constraints, vars }

{ Engine settings }
engine : optional Text ;
parameter : optional text ;
setting : optional number ;
***{** Deprecated }*
itLimit, noImpSeconds, timeLimit, convTolerance: optional Positive ;
mutate: optional Positive;
linVar: optional Scalar;
DerivMethod, EstimMethod, DirecMethod: optional Text;
***SampSz: optional Positive*)**

Defines a non-linear optimization problem. See the Chapter 6, "Non-Linear Optimization", for a description of usage and parameters, and Chapter 7, "Control Settings", for possible engine settings.

Other Functions

LpFindIIS(lp: LpType : newLp : optional boolean)

Computes and returns the *Irreducibly Infeasible Subset* (IIS) of the constraints. This is meaningful when `LpStatus(lp)=2` ("no feasible solution"), and is useful for identifying what portions of your constraint formulation make the problem infeasible.

When the optional parameter, ***newLp***, is specified, returns a new <<LP>> object having the subset of constraints (still infeasible). The components of this object can be accessed using `SolverInfo()`.

LpObjSa(lp: LpType; v: Optional)

Returns the sensitivity ranges for the objective function coefficients for a linear program ***lp*** for decision variable(s) ***v***, which should be one of or a subset of decision variables, ***Vars***. If ***v*** is omitted, it computes the sensitivity for all ***Vars***.

LpOpt(lp: LpType)

Returns the value of the objective function at the optimum.

LpRead(filename: Text; vars, constraints: optional Index ; format : optional Text)

Reads a linear or quadratic program definition from file ***filename***, previously written by `LpWrite()` and returns an opaque <<LP>> or <<QP>> object. The optional ***Vars*** and ***constraints*** are the corresponding indexes for the LP, and must be of the same size as the problem read in. The optional ***format*** parameter may be "LP" (default), "MPS", or "LPFML" to indicate the type of file being read.

LpReducedCost(lp: LpType)

Returns the reduced costs (dual values) of each variable as an array indexed by ***Vars***.

LpRhsSa(lp: LpType;constraint: Optional)

Returns the sensitivity ranges for the ***RHS*** values. The default is to compute sensitivities for all ***RHS*** values, with the result indexed by ***Constraints***. If you specify the optional second parameter, it returns the sensitivity for only that constraint or subset of constraints.

LpShadow(lp: LpType)

Returns the shadow prices (dual values of the constraints) as an array indexed by ***constraints***.

LpSlack(lpv)

Returns the slack or surplus values at the optimal solution as an array indexed by ***constraints***. If it cannot find an optimal solution, it generates an appropriate error.

LpSolution(lp: LpType)

Returns the optimal solution to the linear, quadratic, or non-linear programming problem ***lp*** defined by **`LpDefine()`**, **`QpDefine()`**, or **`NlpDefine()`**. The result is an array of decision variables indexed by ***Vars***. If it cannot find an optimal solution, **`LpSolution()`** returns the best values found during the search so far, and **`LpStatusNum()`** and **`LpStatusText()`** indicate why it has not found an optimal solution.

LpStatusNum(lp: LpType)

LpStatusText(lp: LpType)

Returns the status number as an integer and corresponding text message, respectively, of the optimization problem lp. It is wise to examine the status before evaluating **`LpSolution()`** to avoid an error message.Possible results include:

Status Number	Status Text
-3	Invalid status.
-2	Ignore status. Used when dummy result code needs to be overridden.
-1	Invalid license status. (License expired, missing, invalid, *etc.)*
0	Optimal solution has been found.
1	The Solver has converged to the current solution.
2	"No remedies" status. (All remedies failed to find better point.)
3	Iterates limit reached. Indicates an early exit of the algorithm.
4	Optimizing an unbounded objective function.
5	Feasible solution could not be found.
6	Optimization aborted by user. Indicates an early exit of the algorithm.

Status Number	Status Text
7	Invalid linear model. Returned when a linearity assumption renders incorrect.
8	Bad data set status. Returned when a problem data set renders inconsistent.
9	Float error status. (Internal float error.)
10	Time out status. Returned when the maximum allowed time has been exceeded. Indicates an early exit of the algorithm.
11	Memory dearth status. Returned when the system cannot allocate enough memory to perform the optimization.
12	Interpretation error. (Parser, Diagnostics, or Executor error.)
13	Fatal API error. (API not responding.)
14	The Solver has found an integer solution within integer tolerance.
15	Branching and bounding node limit reached. Indicates an early exit of the algorithm.
16	Branching and bounding maximum number of incumbent points reached. Indicates an early exit of the algorithm.
17	Probable global optimum reached. Returned when MSL (Bayesian) global optimality test has been satisfied.
18	Missing bounds status. Returned for EV/MSL Require Bounds when bounds are missing.
19	Bounds conflict status. Indicates <=, =>, = bounds conflict with existing binary or all different constraints.
20	Bounds inconsistency status. Returned when the lower bound value of a variable is grater than the upper bound value, i.e. lb[i] > ub[i] for some variable bound i.
21	Derivative error. Returned when API_Jacobian has not been able to compute gradients.
22	Cone overlap status. Returned when a variable appears in more than one cone.
999	Exception occurred status. Returned when an exception has been caught by try/catch top-level.
1000	Custom base status. (Base for Solver engine custom results.)
1102	The quadratic constraints are non-convex, the SOCP engine cannot solve this problem.

Note: *The possible status numbers and messages returned in Analytica Optimizer 4.0 have changed since Analytica 3.1, due to changes in the underlying Frontline solver. If you have legacy models that test against specific status numbers, you will need to adjust these accordingly.*

LpWrite(lp: LpType; filename: Text ; format : optional Text)

Writes a Text description of a linear or quadratic program, ***lp***, defined using **`LpDefine()`** or **`QpDefine()`**, to a file with the specified filename. Note that if ***lp*** is an array of LP problems, and the filename does not share the same dimension, the file written by **`LpWrite()`** will contain the result of only the last ***lp***.

LpWriteIIS(lp: LpType; filename: Text ; format : optional Text)

Writes an Irreducibly Infeasible Subset (IIS) of a linear or quadratic program to a file, including only a subset of constraints that is infeasible, but with the property that if any single constraint is removed, the resulting problem will be feasible. The format is the same as that used by **`LpWrite()`**.

SolverInfo(item : Text ; lp : optional LpType ; engine : optional Text)

Returns general Optimizer information, internals of a specific optimization problem, or information about an engine. There are three styles of use.

SolverInfo(item)

Information about the Optimizer. Possible values for item include:

- "AvailEngines" : Returns a list of installed optimizer engines.

SolverInfo(item, lp)

Returns information about an optimization problem definition. Possible values for item include the following, where dimensionality of the result is shown in brackets.:

- "type" : Problem type, one of "LP", "QP", "QCP", or "NLP".
- "vars" [vars] : Elements of the vars index, i.e., variable names.
- "lb" [vars]: lower bound for each variable. Indexed by vars.
- "ub" [vars]: upper bound for each variable. Indexed by vars.
- "ctype" [vars]: Integer type for each variable. One of ’C’, ’I’, ’B’, or ’G’.
- "group" [vars] : Group number for grouped-integer variables.
- "maximize" : 0 for minimization, 1 for maximization problem.
- "objCoef" [vars]: (LP,QP) - objective coefficients.
- "Q" [vars,vars2]: (QP) - objective function quadratic matrix.
- "lhs" [constraints,vars]: (LP,QP) The linear constraint coefficients.
- "lhsQ" [constraints,vars,vars2]: (QP) The quadratic constraint matrices.
- "sense" [constraints]: One of ’>=’, ’<=’, or ’=’ for each constraint.
- "rhs" [constraints]: Right hand side coefficient for each constriant.
- "constraintUb" [constraints]: Upper bound for each constraint.
- "constraintLb" [constraints]: Lower bound for each constraint.
- "engine" : The engine name used to solve the problem.
- "setting" [.Parameter] : The engine settings for this problem. The index is a local index, whose elements depend on the engine used for the problem.

Solver(item, engine:"*engineName*")

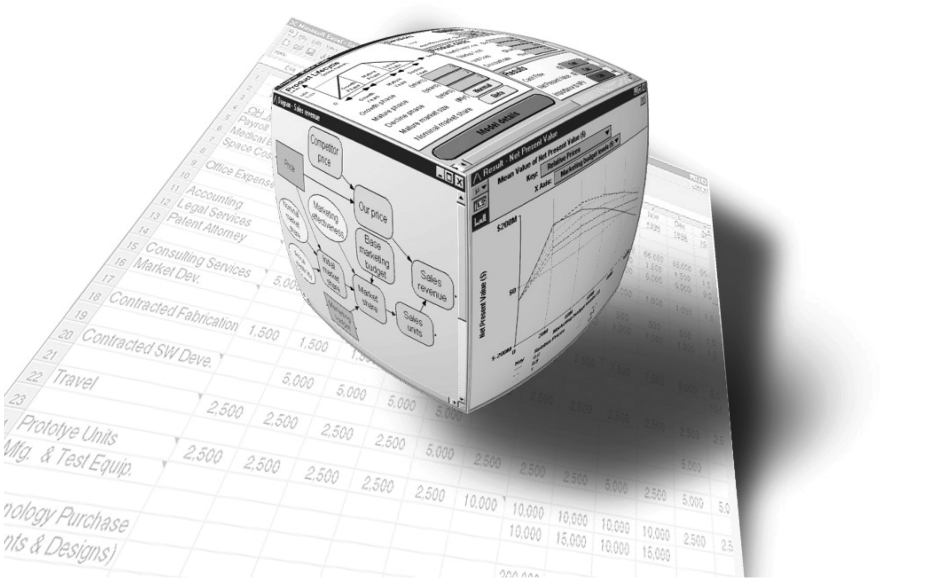
Returns information about an installed optimizer engine. The items providing engine capabilities are indexed by a local index ***.ProblemType***, which includes [’LP’,’QP’,’QCP’, ’CVX’, ’NLP’, and ’NSP’].

Chapter 9

Chapter 10 *Alphabetical Index*

The index provides page numbers in which mention of the following occur:

- Topics & terminology
- Optimizer Functions
- Control settings
- Algorithms
- Identifiers appearing in examples
- Example models



Index