



**analytica**

---

# **Analytica Optimizer 3.1**

**Analytica 3.1 for Windows**

# Copyright notice

Information in this document is subject to change without notice and does not represent a commitment on the part of Lumina Decision Systems, Inc. The software program described in this document is provided under a license agreement. The software may be used or copied, and license codes transferred, only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the licensee's personal use, without the express written consent of Lumina Decision Systems, Inc.

This document is © 1993-2005 Lumina Decision Systems, Inc. All rights reserved.

The software programs described in this document are copyrighted:

Analytica Optimizer: © 1992-2005 Lumina Decision Systems, Inc., all rights reserved.

Analytica: © 1982-1991 Carnegie Mellon University,

© 1992-2005 Lumina Decision Systems, Inc., all rights reserved.

Analytica Optimizer incorporates Solver.dll from Frontline Systems, Inc.: Software copyright © 1991-1999 by Frontline Systems, Inc.

Portions copyright © 1989 by Optimal Methods, Inc.

Portions copyright © 1994 by Software Engines.

Analytica was written using MacApp®: © 1985-1996 Apple Computer, Inc. Analytica incorporates Mac2Win technology, (c) 1997 Altura Software, Inc.

The Analytica® software contains software technology licensed from Carnegie Mellon University exclusively to Lumina Decision Systems, Inc., and includes software proprietary to Lumina Decision Systems, Inc. The MacApp software is proprietary to Apple Computer, Inc. The Mac2Win technology is technology to Altura, Inc. Both MacApp and Mac2Win are licensed to Lumina Decision Systems only for use in combination with the Analytica program. Neither Lumina nor its Licensors, Carnegie Mellon University, Apple Computer, Inc., and Altura Software, Inc., make any warranties whatsoever, either express or implied, regarding the Analytica product, including warranties with respect to its merchantability or its fitness for any particular purpose.

Analytica is a registered trademark of Lumina Decision Systems, Inc.

# Contents

## 1: Introducing the Analytica Optimizer

What do I need to know? .....	1
What is the Analytica Optimizer? .....	1
How do I obtain the Analytica Optimizer? .....	2
To Activate the Optimizer for Analytica .....	3
To activate Analytica Optimizer for ADE .....	4

## 2: Quick Start

Who this is for .....	5
Browsing Analytica Optimizer Functions .....	5
A Linear Program .....	6
A Non-linear Program .....	14

## 3: Formulating an Optimization Problem

<b>Continuous</b> , <b>integer</b> , and <b>mixed-integer</b> programs .....	20
Choosing the type of optimization .....	20
Solving <b>simultaneous</b> equations.....	21

## 4: Linear Optimization

Defining a Linear Optimization Problem.....	23
Optional parameters.....	24
Obtaining the Solution.....	26
Secondary Aspects to Solution .....	27
Examples .....	30
Integer & Binary Decision Variables.....	31
Controlling The Search .....	32
Array Abstraction.....	35

## 5: Quadratic Program Optimization

Defining a Quadratic Program.....	37
Solution Properties .....	38
Common Quadratic Situations .....	39
Obtaining the Solution.....	40
Examples .....	40

## 6: Non-linear Optimization

Problem Formulation.....	42
Obtaining the Solution.....	43
Optional Parameters for NLP .....	43
Integer, Binary and Mixed-Integer Programs .....	44
The Airline Example for NLP.....	45
Intelligent Arrays, array abstraction and NLP.....	49
Solving Systems of Equations.....	59

Other examples .....	60
Giving hints to help the Optimizer .....	60
Controlling the Search .....	63

## **7: Debugging a Problem Formulation**

Writing and Reading From a File .....	67
Diagnosing Conflicting Constraints .....	67
Debugging a Non-Linear Optimization .....	67

## **8: Optimization Function Reference**

Problem Definition Functions .....	71
Other Functions .....	72

# 1: 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 through 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. It 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 constraints. NLP handles general nonlinear objective and constraint functions. All three methods handle decision variables that are continuous, discrete (integer or Boolean), 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 2000 decision variables and 8000 constraints. 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.

The Analytica Optimizer is an add-on module for Analytica Enterprise 3.1. After developing optimizer-based models with Enterprise, you can deliver them to end users on the desktop using Analytica Power Player, 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 bundled with Analytica Enterprise, ADE, or Power Player. Or you can purchase a license for the Optimizer as an add-on module if you already have Enterprise or ADE.

If your copy of Analytica is for release 3.0 or earlier, you will need to upgrade it to release 3.1, because the Optimizer does not work with earlier releases. If you have a maintenance agreement for Analytica 3.0 (included free for 12 months from purchase), you can upgrade it free to Release 3.1.

If you have the Professional edition of Analytica, you will need to upgrade it to the Enterprise edition to work with the Optimizer.

For more information, visit the Lumina web site:

<http://www.lumina.com>

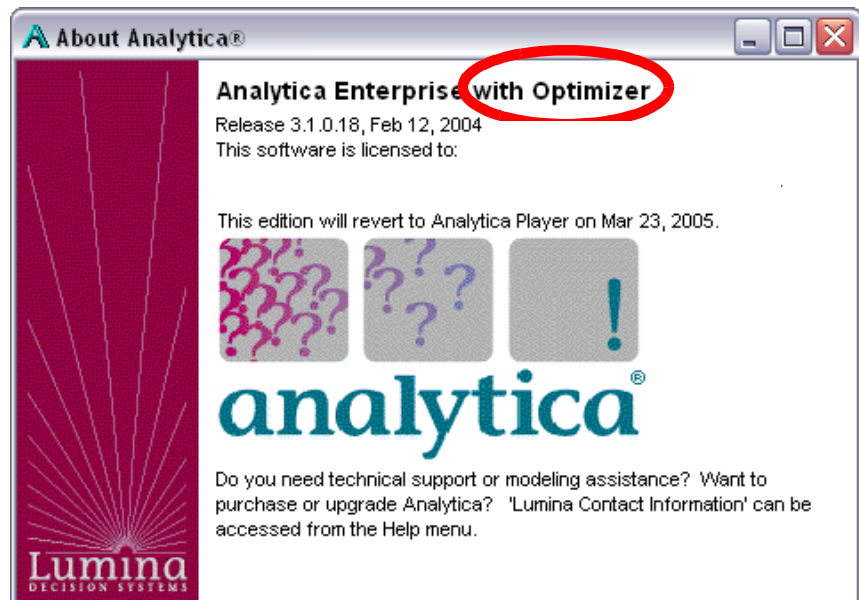
or call Lumina at 650-212-1212.

## To Activate the Optimizer for Analytica

If you have already installed any edition of Analytica 3.1, your installation already includes the Optimizer: 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:



## To activate Analytica Optimizer for ADE

Analytica Optimizer also works with the Analytica Decision Engine (ADE) release 3.1. 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 with the Optimizer includes a license code for Analytica Enterprise with the Optimizer as well as a code for ADE with Optimizer. See the preceding section on how to activate Analytica Enterprise with Optimizer.

If you currently use ADE release 3.0 or earlier, you will first need to upgrade it to release 3.1. The upgrade to ADE 3.1 with Optimizer will include license codes for Enterprise and ADE, each with the Optimizer.

To upgrade an existing installation of ADE 3.1 to activate the optimizer, follow these steps to enter a new license code with the Optimizer:

1. Open a command prompt. From the **Start** menu, select **Run**, type `cmd` (or `command` on Win98/ME) and press **OK**.
2. Type: `cd ADE_Dir`  
where `ADE_Dir` is the path to the directory for ADE 3.1. On many computers this will be:  
`cd c:\Program Files\Lumina\ADE3.1`
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**.



## 2: Quick Start

### Who this is for

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

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 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 [“8: Optimization Function Reference” on page 71](#).

#### 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 3.1 > Analytica 3.1**.
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](mailto: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?<sup>1</sup>*

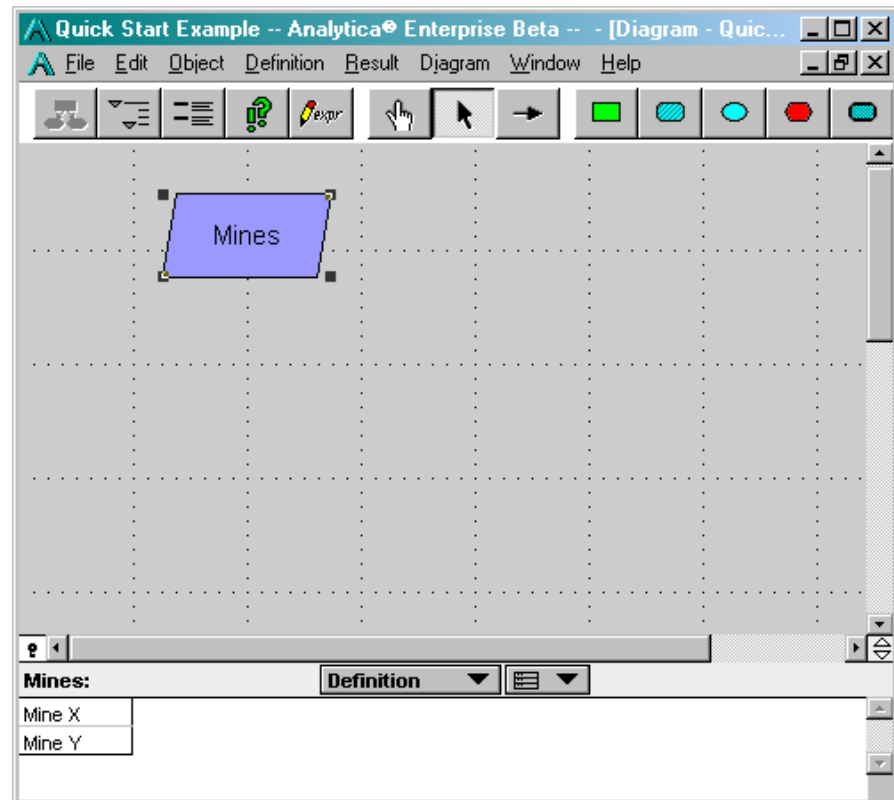
### LpStatusNum() Return Values

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

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

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:

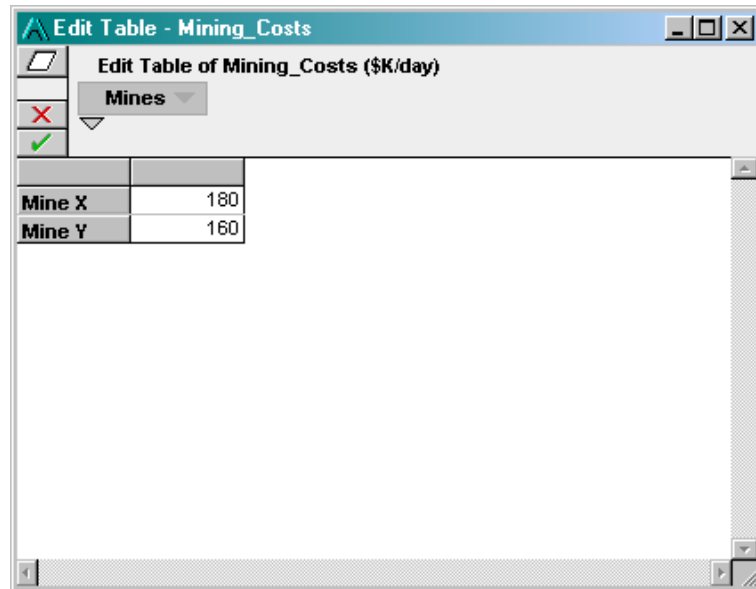
7. Create an index name and name it *Mines*.  
We will use this as the index for the objective variables, i.e., the number of days per week to operate each mine.:
8. Edit its definition attribute and set its definition pull-down to **List Of Labels**.
9. Enter the labels *Mine X* and *Mine Y*:



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

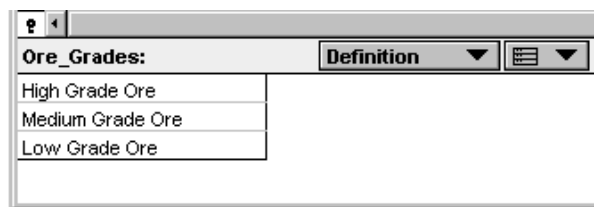
10. Create a variable and name it *Mining\_Costs*. Set its units attribute to  $\$K/day$ .

11. 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:



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:

12. Create an Index variable and name it *Ore\_Grades*. Set its definition to a list of labels, thus:



13. 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

14. Create a variable and title it "Ore Production Requirements". For convenience, set its identifier to *Ore\_Prod\_req*. Set its Units to **tons/week**.

15. 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:

High Grade Ore	12
Medium Grade Ore	8
Low Grade Ore	24

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

$$\text{Sum}(\text{Ore\_production} * \mathbf{x}, \text{Mines}) \geq \text{Ore\_prod\_req}$$

where  $\mathbf{x}$  is the objective, *i.e.*, the number of days per week to operate each mine.

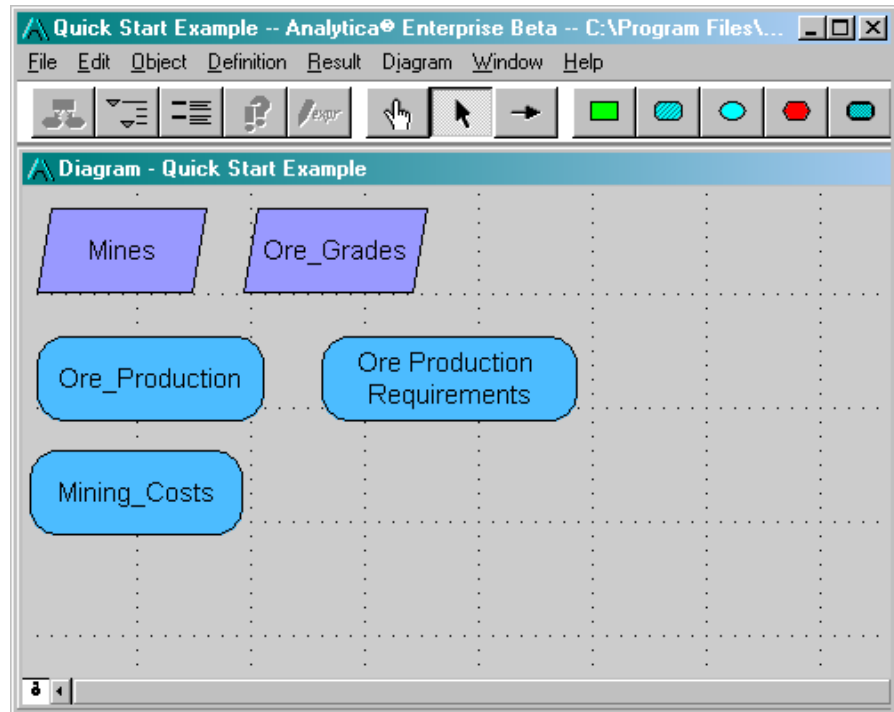
We now have all the inputs required to define the linear program.

To create the linear program to solve this problem:

16. 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)
```

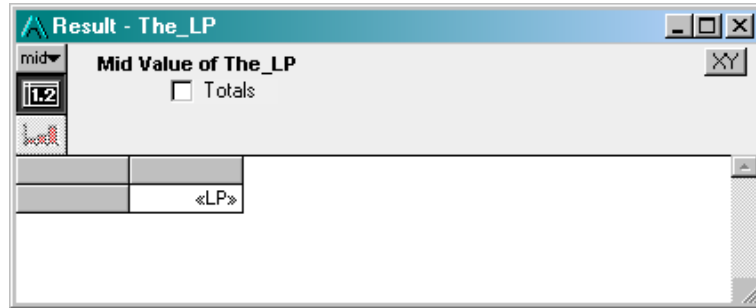
The parameters *lhs*, *sense*, and *rhs* refer to the left hand side of the constraint equations, the constraint equation comparator (greater than, equals, less than), and the right hand side of the



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.

**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.)

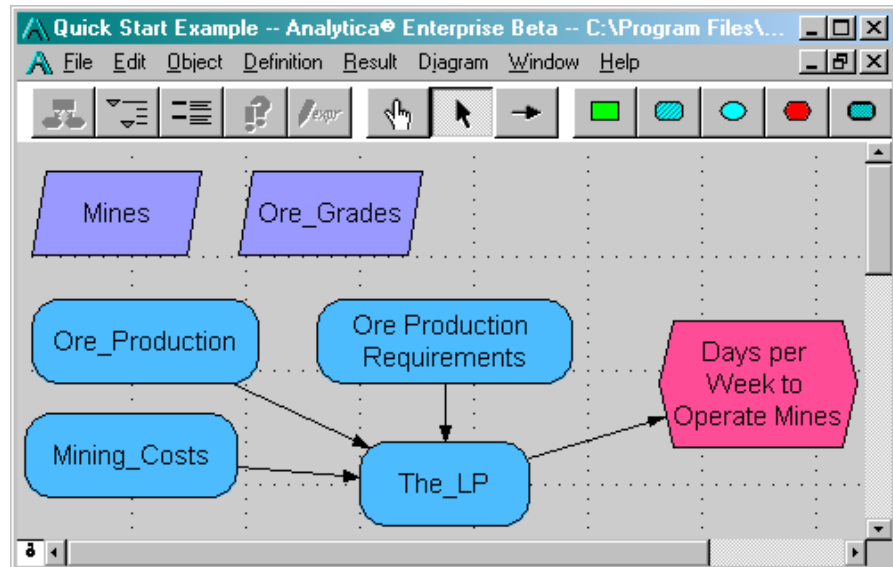
17. Select the *My\_LP* node and press *CTRL-R* to evaluate it.



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:

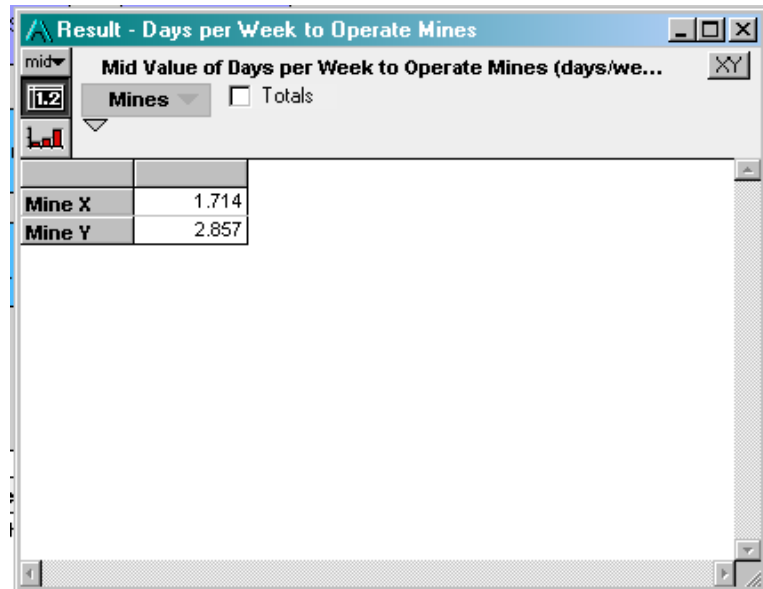
18. Create an objective node and title it "Days per Week to Operate Mines". Set its units attribute to `Days per week`, and set its definition to `LpSolution(My_LP)`

Your model should now look something like:





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



Mine	Days per Week
Mine X	1.714
Mine Y	2.857

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

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

In this case, *Status* should be “optimal solution 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:

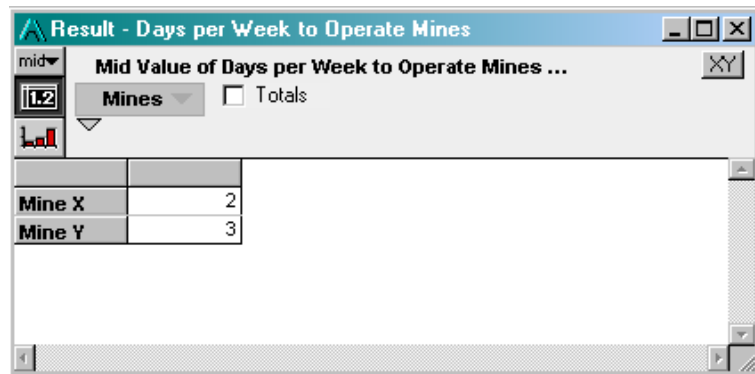
Click on *My\_LP* and change its definition by adding a ***ctype*** (constraint type) parameter to indicate that you want an integer solution:

```

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.



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.

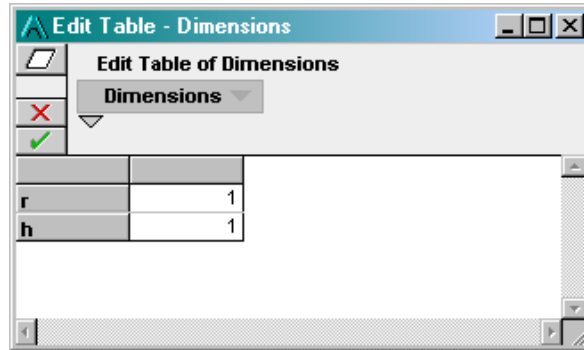
We will formulate the following optimization problem:

*Find the dimensions of a cylinder with minimum surface area with a volume of at least 500 cm<sup>3</sup>.*

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:



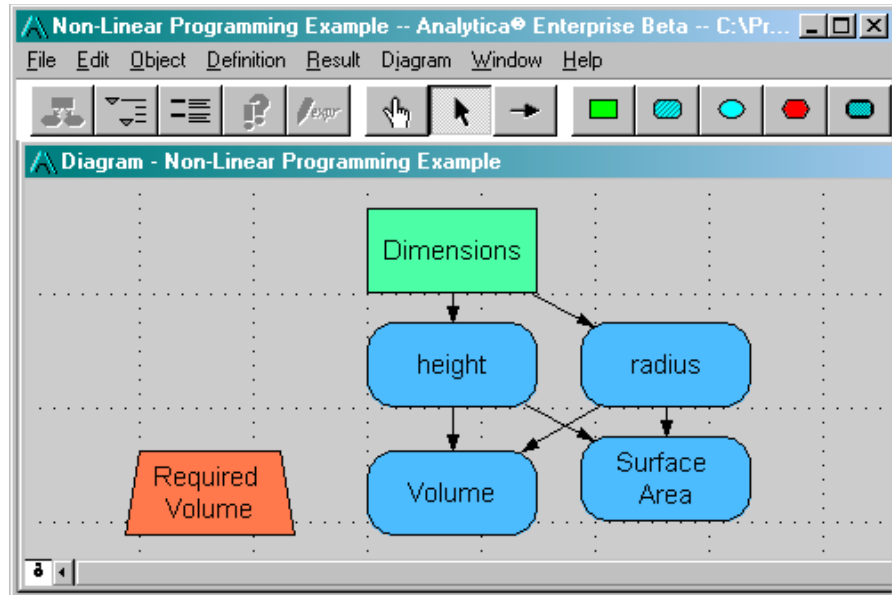
Since it is self-indexed, the *Dimensions* variable serves both as the optimization vector and as the **vars** index. During the optimization search, the cell values will be set to candidate solutions and other portions of the model evaluated.

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	'>='	Required_volume

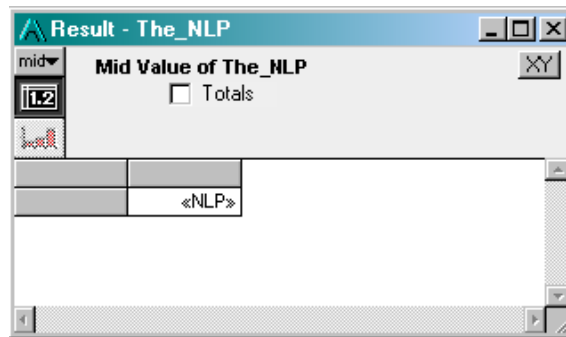
Now, define the non-linear optimization problem:

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"])
```

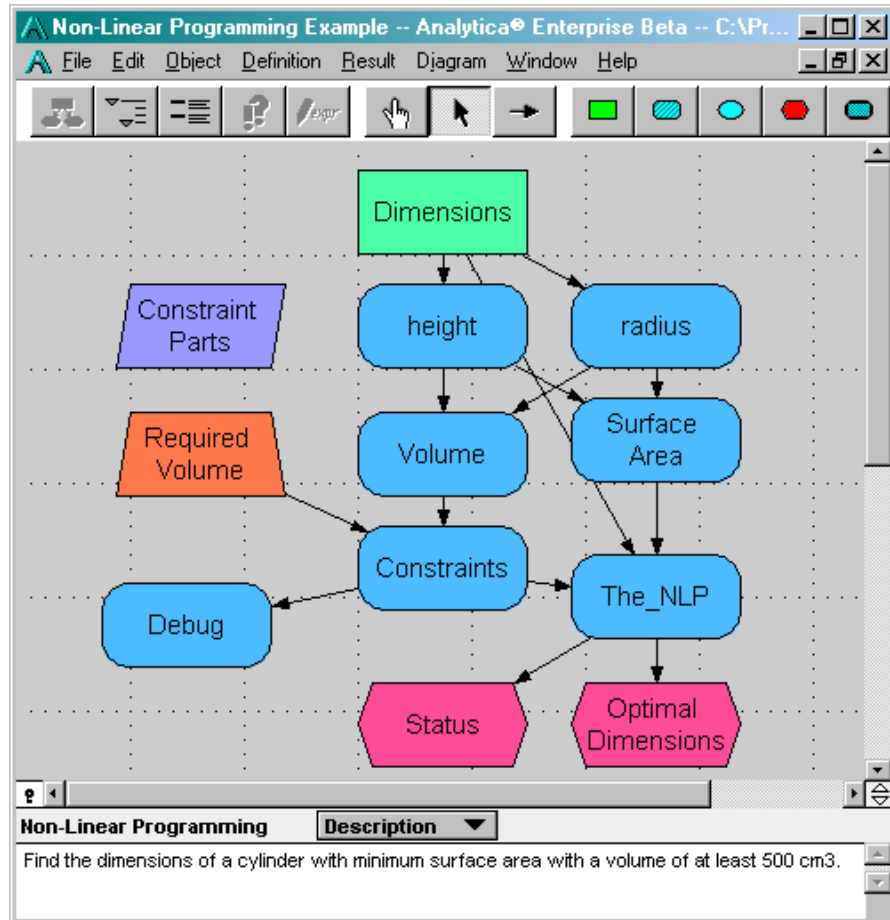
This defines the non-linear optimization problem. The objective function is *Surface\_area*, which is computed from the values in the *Dimensions* node. The left-hand side of the constraint is also computed from *Dimensions*.

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.

**Result - Status**

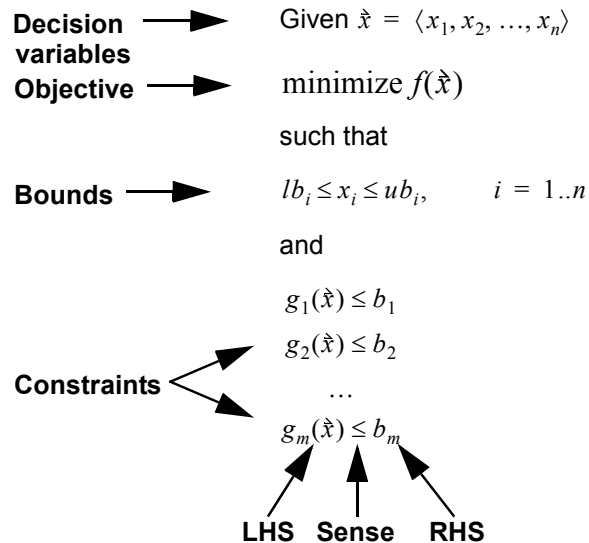
Mid Value of Status	
<input type="checkbox"/> Totals	
optimal solution found	

**Result - Optimal Dimensions**

Mid Value of Optimal Dim...	
<input type="checkbox"/> Totals	
Dimensions	
r	4.301
h	8.603

### 3: Formulating 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:



#### Decision variables

A vector (one dimensional array)  $\hat{x} = \langle x_1, x_2, \dots, x_n \rangle$  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  $f(\hat{x})$  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  $lb_i \leq x_i \leq ub_i, \quad i = 1..n$  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  $-\mathbf{INF}$  and  $+\mathbf{INF}$  — that is, there are no bounds.

### Constraints

The constraints, e.g.,  $g_1(\hat{x}) \leq b_1$ , are bounds on functions of the decision variables. They define which solutions are acceptable.

Each constraint consists of a **lefthand 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.  $b_1$ .

## 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, or as **binary** or **Boolean**, meaning its values may be True (1) or False (0). Optimization problems are classified as **continuous**, meaning the decision variables are all continuous, **integer**, meaning they are all integer or binary variables, or **mixed-integer** if they are a mixture of continuous and integer or binary 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 must be a linear or quadratic function 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 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 “6: Non-linear Optimization” on page 41. 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

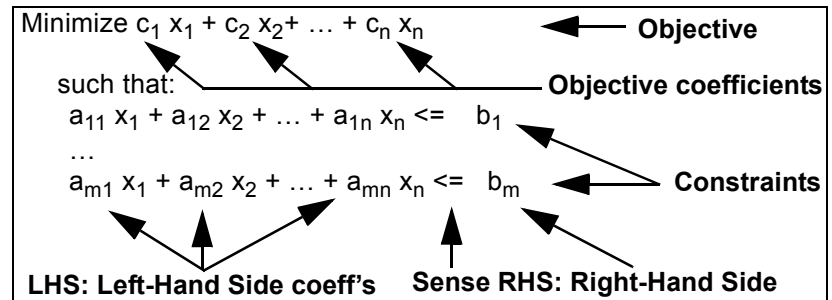
The optimizer first attempts to find a feasible solution. If found, it then attempts to optimize within the set of feasible solutions. Thus, the solving a set of simultaneous equations is a special case of the optimization problem, where each constraint has a **sense** of "=", the objective is irrelevant,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



## 4: Linear Optimization

### Defining a Linear Optimization Problem

A linear optimization problem has the following standard formulation:



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

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

```
LpDefine(Vars, Constraints: IndexType;
         ObjCoef: Numeric[Vars];
         LHS: Numeric[Vars, Constraints];
         RHS: Numeric[Constraints])
```

with these required parameters:

#### **Vars**

An index over the  $n$  **Decision Variables**,  $[x_1, x_2, \dots, x_n]$ , 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,  $[c_1, c_2, \dots, c_n]$ , 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,  $c_1 x_1 + c_2 x_2 + \dots + c_n x_n$ .

**LHS**

The Left-Hand Side of the constraints is an  $n$  by  $m$  array of coefficients, indexed by **Vars** and **Constraints**,  $a_{11}, a_{12}, \dots, a_{ij}, \dots, a_{mn}$ . A range 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.

**RHS**

The Right-Hand Side of the constraints, being an array of  $m$  constants,  $(b_1, b_2, \dots, b_m)$  indexed by **Constraints**. The

constraints, e.g.,  $g_1(\hat{x}) \leq b_1$ , are bounds on functions of the decision variables. They define which solutions are acceptable.

Each constraint consists of a **lefthand 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(VarIndex, ConIndex, ObjCoef, lhs,
rhs,
Maximize: True,
```

```
Lb: 0,  
Sense: ">=")
```

In this case, the first five parameters are the required indexes and coefficients as described in the previous section, and the last three parameters are optional parameters, specifying that we want to **Maximize** the objective, each decision variable ( $x_1, \dots, x_n$ ) 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 Numeric[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**.

## 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 TextType[Constraints]
```

is used to specify the sense for each constraint. When omitted, it assumes "<=" by default. The following 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 decision variables,  $x_1, \dots, x_n$ , 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:

$$\text{Sum}(\text{LpSolution}(\text{lp}) * \text{ObjCoef}, \text{Vars})$$

### **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, for the programming problem *lp*. These may be 1 and "Optimal solution found", or another number with text explaining why it has not found an optimal solution.

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	Description (LpStatusText)
1	Optimal solution found
2	No feasible solution
3	Objective unbounded
5	Iteration limit exceeded, feasible
6	Iteration limit exceeded, not yet feasible
7	Time limit exceeded, feasible
8	Time limit exceeded, not yet feasible
65	Objective function changing too slowly

---

**Analytica Note:** `LpSolution()` will generally 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 than just the optimal solution,  $(x_1, \dots, x_n)$ . 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

$$a_{i1} x_1 + a_{i2} x_2 + \dots + a_{in} x_n \leq b_i$$

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

$$a_{i1} x_1 + a_{i2} x_2 + \dots + a_{in} x_n + \text{slack}_i \leq b_i$$

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 ( $x_1, \dots, x_n$ ) 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,  $b_i$ , 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 23) there corresponds a dual linear program given by:

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

such that

$$a_{11} y_1 + a_{21} y_2 + \dots + a_{m1} y_m \geq c_1$$

...

$$a_{1n} y_1 + a_{2n} y_2 + \dots + a_{mn} y_m \geq c_n$$

The new variables in this program,  $y_1, y_2, \dots, y_m$ , 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, "2: Quick Start.")

## Integer & Binary 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 decision variables that are constrained to be integers, Boolean or binary, including a mixture of continuous and integer or binary variables (a *mixed integer program*).

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

***ctype***: *Optional TextType*[Vars]

which takes one of the following values:

- "C": Continuous
- "I": Integer
- "B": Binary or Boolean value, i.e. 0 or 1

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.

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. They return the following status numbers and text messages, respectively:

Status	Description
101	The MIP optimal solution found
102	MIP solution found within gap tolerance (see “Controlling The Search” on page 32)
103	No feasible integer solution
104	Integer solution limit exceeded
105	Node limit exceeded, feasible
106	Node limit exceeded, not feasible
107	Time limit exceeded, feasible
108	Time limit exceeded, not feasible

For a complete list of the possible values returned by `LpStatusNum()`, see “`LpStatusNum(lp: LpType)`” on page 73.

## Controlling The Search

Several optional parameters to `LpDefine()` can be used to influence how the search for the optimum proceeds and when it terminates. All the parameters described in this section may be optionally included with the `LpDefine()` function.

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. 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.

## Termination Control

### ***ItLimit***: Optional Positive Integer

Specifies the maximum number of iterations (pivots) by the Simplex Algorithm during the optimization. If this is exceeded, **LpStatusNum()** returns 5 (feasible solution found) or 6 (feasible solution not found).

**Default**: no limit.

### ***TimeLimit***: Optional Positive Integer

Maximum number of seconds the optimizer will spend on the problem. If exceeded, **LpStatusNum()** will be 7 (feasible found) or 8 (no feasible found) for a continuous problem, and 107 or 108 for a MIP problem.

**Default**: 65535 seconds (the maximum allowed).

### ***NdLimit***: Optional Positive Integer

Limits the number of nodes (or LP sub-problems) considered by the branch-and-bound algorithm when solving an integer, binary or mixed-integer problem. If exceeded, **LpStatusNum()** will return 105.

**Default**: no limit.

**MipLimit: Optional Positive Integer**

The maximum number of feasible solutions that the branch-and-bound algorithm will visit before terminating.

**Default:** no limit.

**GapTolerance: Optional Positive Percentage**

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%

**Tolerance and Precision Control****OptTolerance: Optional Positive**

The Optimal or Reduced Cost Tolerance. 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  $10^{-4}$

**PivotTolerance: Optional Positive**

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

**Default:**  $10^{-6}$

**Allowed range:**  $10^{-6}$  to  $10^{-4}$

**FeasTolerance: Optional Positive**

The Feasibility Tolerance for MIP problems. The tolerance is used to determine which constraints are considered satisfied and which decision variables are treated as integers.

**Default:**  $10^{-8}$

**Allowed range:**  $10^{-8}$  to  $10^{-4}$

**Algorithm Control****OptLb, OptUb: Optional Numeric**

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, only the lower bound is relevant, and for a minimization problem, only the upper bound is relevant.

**Default:** no bounding

**Scaling: Optional Boolean**

Setting this to **False** turns off internal scaling during the solution process. The optimizer will, by default, rescale decision variables and constraints internally for the Simplex algorithm, which usually leads to be reliable results and fewer iterations.

**Default:** True

**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-side of every constraint as 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.

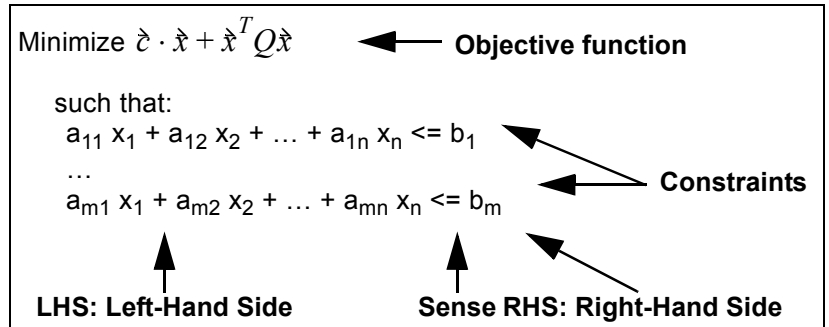


## 5: Quadratic Program Optimization

### Defining a Quadratic Program

A quadratic program has the basic form:

#### Quadratic Optimization Problem: Standard Form



The only difference between a quadratic and linear program formulation is the objective function. The objective function for a quadratic program is written here in matrix notation:  $\hat{c} \cdot \hat{x}$  is the linear part of the objective, where  $\hat{c}$  is a vector of coefficients. A quadratic program adds a second term,  $\hat{x}^T Q \hat{x}$ : to the objective.  $Q$  is a square matrix with the number of rows and number of columns equal to the number of decision variables, and  $\hat{x}^T$  is the transpose of the decision variables.

**Note:** While the objective function is quadratic, all constraints must be linear. If you have quadratic or non-linear constraints, you will need to formulate the problem as a non-linear program (NLP).

To ensure that the  $Q$  matrix is 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**, is required. (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:
  IndexType:
  c: Optional Numeric[Vars]:
  q: Numeric[Vars, Vars2]:
  LHS: Numeric[Vars, constraints]:
  RHS: Numeric[constraints];

  {Optional parameters;}
  sense: Optional TextType[constraints];
  maximize: Optional Boolean;
  lb,ub: Optional Numeric[Vars];
  ctype: Optional TextType[Vars];
  guess: Optional Numeric[Vars];
  warnIndefinite: Optional Boolean;
  ItLimit, NdLimit: Optional Positive;
  MipLimit, TimeLimit: Optional Positive;
  OptTolerance, PivotTolerance, FeasTolerance,
  GapTolerance: Optional Positive;
  OptLb, OptUb: Optional Scalar;
  Scaling: Optional Scalar)

```

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

All of the optional parameters of `LpDefine()` are also accepted by `QpDefine()`, see Chapter 4, “4: Linear Optimization,” on page 31. The chapter on linear programming should be consulted for further information on these. Analytica Optimizer supports integer, binary, and mixed-integer quadratic programs. Two additional optional parameters, ***guess*** and ***warnIndefinite***, which don’t exist for linear programs, may also be supplied, and are discussed below.

## Solution Properties

The **Q** matrix is the Hessian, consisting of the second partial derivatives of the objective function. Depending on the values in this 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  $x: x^T Q x > 0$ : 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 cup-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 Numeric[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.

You can also have `QpDefine()` issue a warning message if the  $Q$  matrix is indefinite by setting the optional `warnIndefinite` parameter to `True`.

## 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,  $\mathbf{Q}$ , can be used capture the pair-wise covariances between investments, as well as the variances of the individual investments (the diagonal elements of  $\mathbf{Q}$ ). Letting each element of the vector  $\hat{x}$  be the fraction of the total portfolio allocated to investment, the variance of the complete portfolio is  $\hat{x}^T \mathbf{Q} \hat{x}$ . 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 “Quadratic Optimization Problem: Standard Form,” on page 37. 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  $\mathbf{Q}$  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, “4: Linear Optimization”)

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

**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.

## 6: Non-linear Optimization

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:

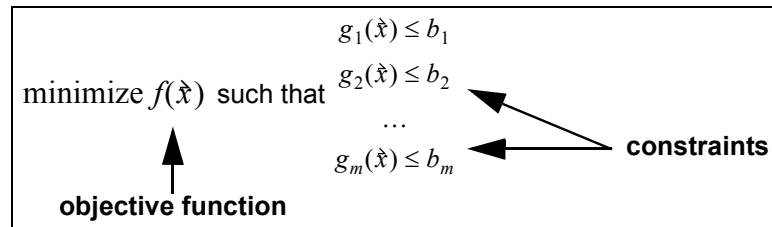
- The type of dependence — i.e., whether the objective or constraint functions vary linearly, smoothly, or discontinuously with each decision variable.

- 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:

### Non-Linear Optimization: Basic Form



where  $\hat{x}$  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: IndexType ;
          X: VariableType :
          Obj, LHS: Expression :
          RHS: Numeric[Constraints])
```

#### **Vars**

An index for the decision variables, **X** below.

#### **Constraints**

An index for the constraints, **LHS** and **RHS** below.

#### **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.

<b>LHS</b>	The Left-Hand Side of the constraints, indexed by <b>Constraints</b> . Each element of <b>LHS</b> may be an expression or a variable. They must depend on the decision variables <b>X</b> , directly or indirectly.
<b>RHS</b>	The Right-Hand Side of the constraints, indexed by <b>Constraints</b> . Each element of the array passed to <b>RHS</b> must evaluate to a single number. It must not depend on the decision variables. By default, feasible solutions are those in which the <b>LHS</b> is less than or equal to the corresponding value of <b>RHS</b> . You can change this with the optional <b>Sense</b> parameter, described below.

## 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 8, “8: Optimization Function Reference,” on page 71.

## Optional Parameters for NLP

The following are optional parameters to **NlpDefine()**:

### 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 TextType[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 Numeric[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**.

## 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 TextType[Vars]**

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

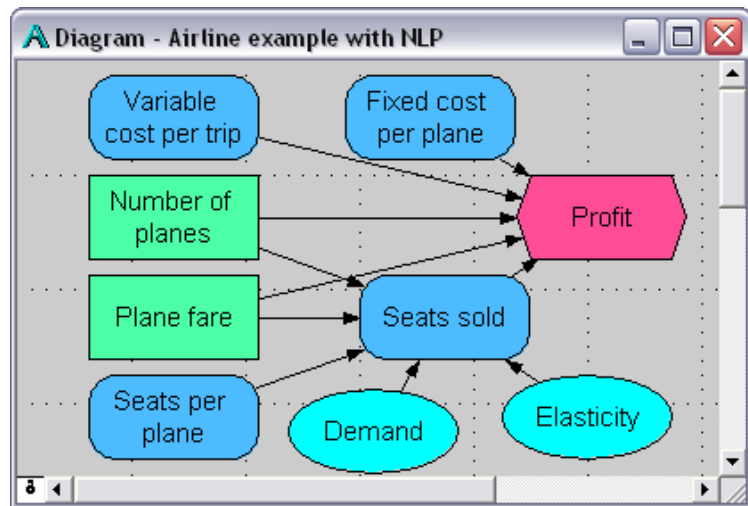
- "C": Continuous
- "I": Integer
- "B": Binary or Boolean

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



## 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 **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)
  
```

We assume that the demand is elastic with respect to changes in price, using a demand function that raises the ratio of the fare to the base fare of \$200 to the negative power of the elasticity. We

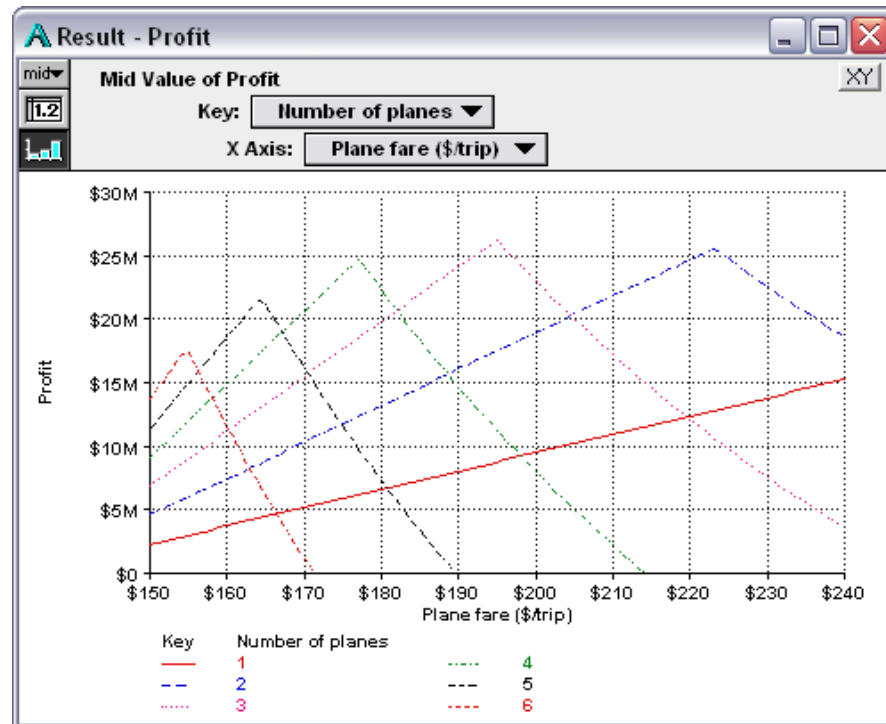
compute the actual `Seats_sold` as the lesser of the demand modified for price elasticity and the actual seats available, the product of the number of planes and `Trips_per_plane`:

```
Variable Trips_per_plane := 200 * 360 * 2
Variable Seats_sold := Min([Base_demand *
    (Fare/200)^-Elasticity1,
    Num_planes * Trips_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)(1, 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`:

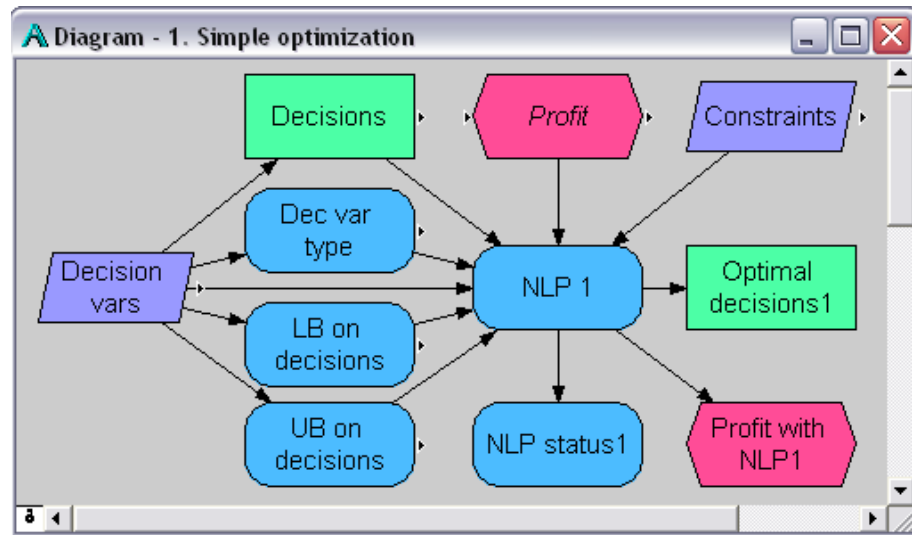
```
Num_planes:= Decisions[Dvars = 'Number of planes']
Fare := Decisions[Dvars = 'Plane fare']
```

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. Optimize decisions 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 NLP 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)
Index Constraints := [0]
```

In this example, we use no constraints, so we set the index **Constraints** to a single arbitrary single element. We can now define the NLP using these parameters:

```
Variable NLP1 := NLPDefine(Vars: Dvars,
  Constraints: Constraints,
  X: Decisions, Obj: Profit,
  LHS: 0, RHS: 1, Maximize: True,
  Ctype: Dec_type,
  LB: Lb_decisions, UB: Ub_decisions)
```

We set **LHS** to 0 and **RHS** to 1 to guarantee  $LHS \leq RHS$ , since this problem has no constraints (other than decision bounds). 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(Nlp1)
Objective Profit_with_nlp1 := LPOpt(Nlp1)
Variable Nlp_status1 := LPStatusText(Nlp1)
```

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):

Decision vars	Totals
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 61 for details on ***Gradient*** and ***Jacobian***.

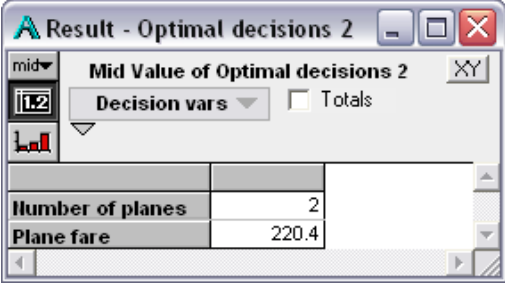
Note that `NLPDefine()` does generalize fully over extra dimensions for all parameters other than those five listed above. But, for those five parameters, it is up to you, the modeler, to make sure that they have only the required indexes. Otherwise it will flag an error. Read on to see how to get around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

## 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	
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.

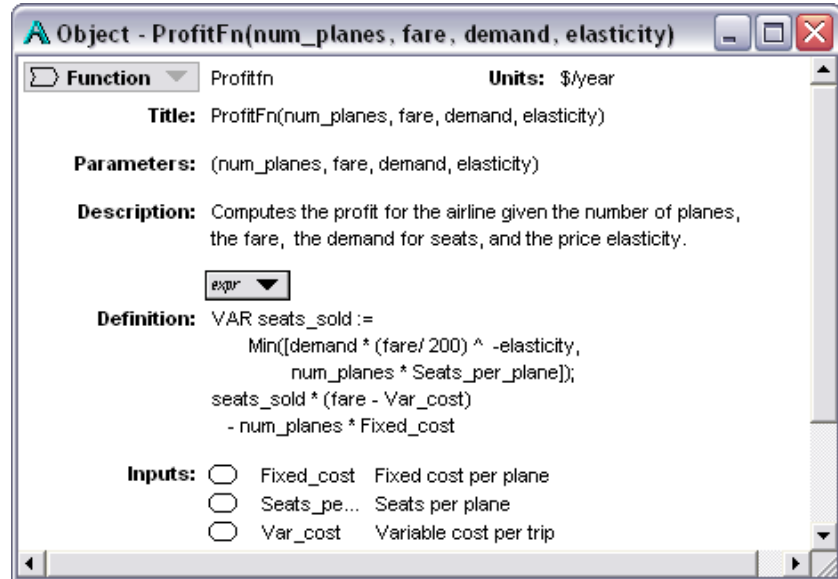
In these cases, you are trying to find the optimal decision now, *before* resolving the uncertainties that affect the objective or constraints. You can set the model to perform a single optimization and the result is a single optimal solution (set of decisions) and corresponding maximum expected value (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

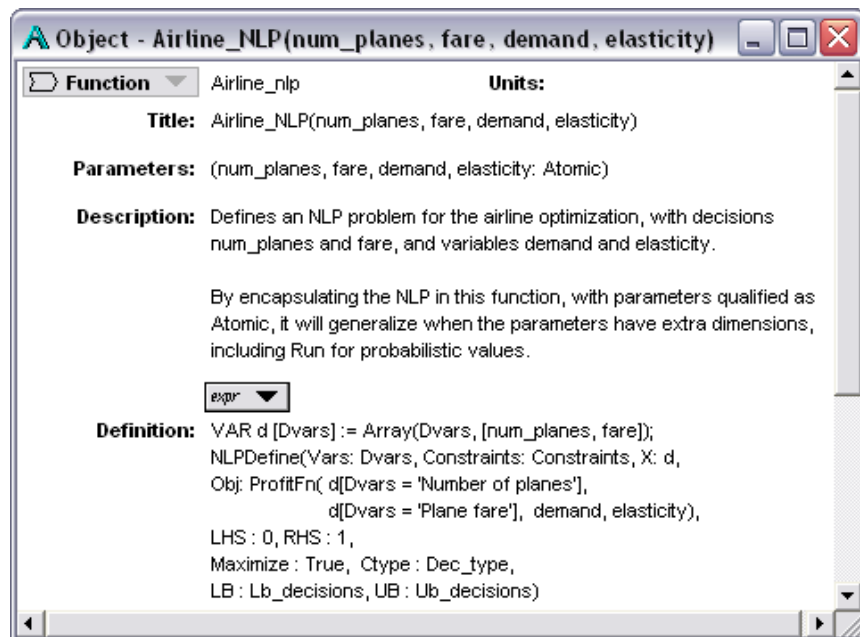
The second type of optimization under uncertainty is less common: The optimal decisions will be made *after* resolving the uncertainty, and you want to compute probability distributions over what those optimal decisions will be now while still uncertain. This is sometimes known as *preposterior* analysis 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 *preposterosus* 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 NLP1, by selecting a uncertain view, e.g. *Sample*, in the Result for `Optimal_decisions1`, shown previously: But, this would generate the warning "Value is not probabilistic. Mid value will be shown instead." This is because `NLPDefine()` always evaluates its parameters deterministically.

Instead, we need to create an NLP that abstracts over the Chance variables, so that the `Run` index does not cause problems for `NLPDefine`. 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.)



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





`Airline_nlp()` qualifies its parameters as **Atomic**. This 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:

Sample of HLP 3	
Iteration (Run)	
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:

Sample of Optimal decisions 3					
Decision vars	Iteration (Run)				
	1	2	3	4	5
Number of planes	3	4	3	3	3
Plane fare	188.4	189.1	197.7	194.9	195.8

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.

#### 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**:

The screenshot shows a window titled "Result - Optimal decisions 4". It contains a table with the following data:

	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.

#### 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 := Triangular(0%, 10%, 20%)
Time := 2005 .. 2010
Variable Demand_by_time := 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,
    Obj: Mean(NPV(Discount_rate,
        ProfitFn( Num_planes, Fare,
            Demand_by_year, Elasticity1), Time)),...)
```

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 deci...	
Decision vars	
Number of planes	3
Plane fare	203.5

## 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:

	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

## 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. Here are sample results for the optimal decisions:

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

The screenshot shows a window titled "Result - Optimal decisions 7". The window contains a table with the following data:

Decision var	Value
Num planes 2005	2
Num planes 2006	3
Num planes 2007	3
Num planes 2008	3
Num planes 2009	4
Num planes 2010	4
Fare 2005	220.5
Fare 2006	200
Fare 2007	205.4
Fare 2008	212.7
Fare 2009	199.6
Fare 2010	206.4

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.

The screenshot shows a window titled "Result - Optimal decisions 7". The window contains a table with the following data:

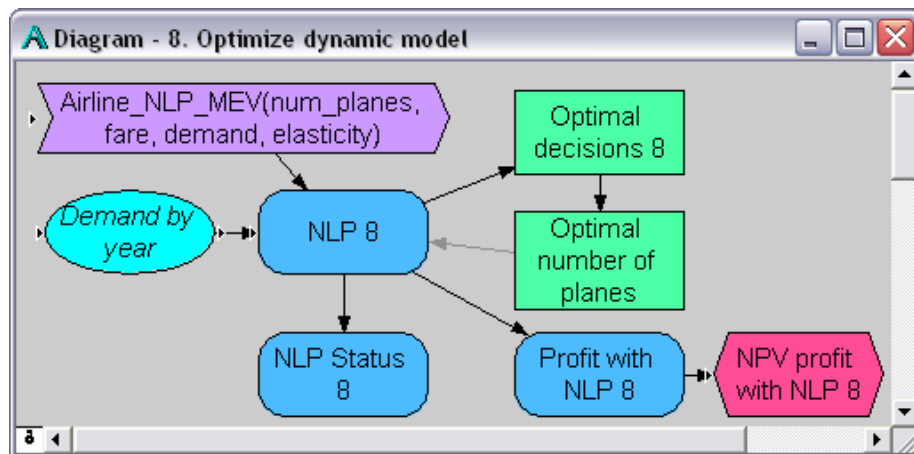
Decision var	Value
Num planes 2005	2
Num planes 2006	3
Num planes 2007	3
Num planes 2008	3
Num planes 2009	4
Num planes 2010	4
Fare 2005	220.5
Fare 2006	200
Fare 2007	205.4
Fare 2008	212.7
Fare 2009	199.6
Fare 2010	206.4

## 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_5 := 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:



## 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 maximize 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 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 set the objective function (the *Obj* parameter) to zero 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, "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.

## 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()`.



```
objN1: Optional TextType [Vars]  
lhsN1: Optional TextType [Vars,  
Constraints]
```

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

- "L": Linear or no 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 *lhsN1*, 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 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), 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 assume the worst case, *i.e.*, discontinuous. This limits its ability to take advantage of the simpler dependencies that might exist. On the other hand, if your search space is very rough, with many local optima, the genetic algorithm may actually perform better, so in some complex cases you may find better performance by using "D" (or omitting these parameters). You may simply need to try it both ways to find out.

## 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:

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

where  $f(\hat{x})$  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:

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

where  $g_j(\hat{x})$  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 49 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 **IpStatusNum()** = 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 Numeric[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.

## Controlling the Search

Several optional parameters to **NlpDefine()** can be used to control how the search is conducted, and when the search is terminated.

### Method Parameters

Several optional parameters influence how the optimizer makes decisions. The first group applies to gradient-based search, used with linear and smooth non-linear functions.

#### Gradient-search control

**LinVar: Optional Boolean**

When **LinVar** is specified and set to True, 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.

The **DerivMethod** parameter controls how derivatives are computed:

**DerivMethod: Optional TextType**

- **"forward"**: This is the default if Jacobian and gradient parameters are not supplied. The optimizer estimates derivatives using forward differencing, *i.e.*,

$$\frac{\partial}{\partial(x)} \approx \frac{f(x + \Delta) - f(x)}{\Delta}$$

- **"central"**: The optimizer estimates derivatives using central differencing, *i.e.*,

$$\frac{\partial}{\partial x} \approx \frac{f(x + \Delta) - f(x - \Delta)}{2\Delta}$$

- **"jacobian"**: The optimizer computes derivatives using the supplied Jacobian and gradient expressions. This is the default if these are supplied.
- **"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.

The  `DirecMethod`  parameter controls how the gradient-based search determines the next point to jump to during search:

**DirecMethod** : Optional TextType

- **"Newton"**: Uses a quasi-Newton method, maintaining an approximate Hessian matrix for the reduced gradient function.
- **"Conjugate-gradient"** or **"CG"**: Use a conjugate gradient method, which does not require the Hessian.

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

**EstimMethod** : Optional TextType

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

## Genetic Algorithm Control

When a problem has discontinuous dependencies, or when the optional  `objNI`  and  `lhsNI`  parameters are not specified, the non-linear optimizer uses a genetic algorithm-based method. You can use these parameters to modify how the Optimizer applies this algorithm:

**SampSz** : Optional Positive

**Mutate** : Optional Positive

**SampSz** specifies the population size of candidate solution to be maintained by the genetic algorithm. If not specified or zero, the algorithm will select a “reasonable” size, usually 10 times the number of decision variables, but no more than 200.

**Mutate** specifies 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.

## Termination Criteria

In general, the non-linear Optimizer has no way to know whether it has found the global optimum, since there may be many local optima and the search space may not be convex. In such cases, the termination criteria are particularly important. These optional parameters control when the non-linear Optimizer stops its search and returns a solution:

**itLimit: Optional Positive:**

**nolmpSeconds: Optional Positive:**

**timeLimit: Optional Positive:**

**convTolerance: Optional Positive**

**itLimit:** The maximum number of optimization steps during the search.

**Default:** no limit.

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

**Default:** 30 seconds.

**TimeLimit:** The maximum number of seconds that the Optimizer will spend on the entire optimization problem. If this limit is exceeded, it returns the best solution so far, if any; and **LpStatusNum()** returns 7 (“feasible solution found”) or 8 (“no feasible solution found yet”).

**Default:** no limit.

**ConvTolerance:** Convergence tolerance. Used to detect a slowly changing objective. When a smooth optimizer algorithm is employed, the optimization will terminate when the previous 5 iterations have not deviated by more than this amount. For the non-smooth, non-linear optimizer (which utilizes a genetic algorithm), the optimization will

terminate when 99% of the population have “fitness values” that differ by less than this value. In either case, the Optimizer returns status 65 (“objective changing too slowly”).  
**Default:**  $10^{-4}$

## 7: 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*: TextType)  
LpRead(*filename*: TextType)

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.

### 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*: TextType)

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, quadratic and non-linear optimizations.

LpWriteIIS() writes the IIS to an indicated file and returns the full file path. This function can be used with linear and quadratic optimizations, but not with non-linear optimization problems. The file format is the same as that used by LpWrite().

### 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

## Using MsgBox to Debug

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. Being familiar with a few Analytica tricks can be of great assistance here.

One of the first things to try is to simply peek at what values optimizer is plugging in for **X**. 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)
```

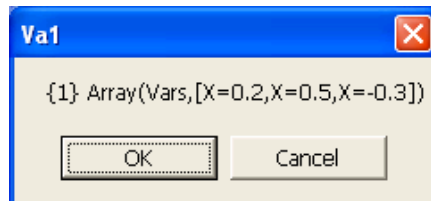
Then each time the optimizer evaluates the objective, a message box will appear on the screen, allowing you to view progress. Seeing the optimizer in action will often give you an understanding of what it would take to improve the search.

There are a few quirks to be aware of when using **MsgBox** in this fashion. First, the **noImpSeconds** 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:



rather than

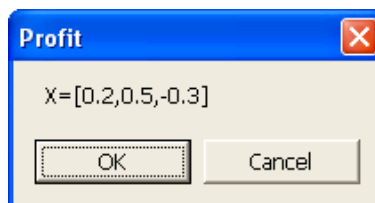
```
X=Array(Vars,[X=0.2,0.5,-0.3])
```



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

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

which outputs:



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

### Writing trace to file

One difficulty with using the `MsgBox()` trick is that you may need to view a lot of dialogs during the course of your optimization. Another option is to write trace information to a log file and view it separately to understand how the search progressed.

The Analytica function

```
WriteTextFile (filename, text, appendFlag)
```

can be used for this purpose. The filename is interpreted relative to the current Analytica data directory, and the function returns the full file path actually written. Passing `TRUE` for the third parameter appends each line to the log file, so that the full search is captured. You can then view the log file in a separate text editor to diagnose what went wrong. The information of interest is provided in the text parameter, which you should ensure is atomic and not an array. If you pass an array, each element of the array will be written separately.

It is often convenient to define a constant named `CRLF` having the definition `Chr(13) & Chr(10)`. With such a constant, your objective function expression might look like:

```
obj: LpWrite("log.out",
            "x=" & join(x, Vars) & crlf,
            True) :
      Sum(Exp(-a*x), Vars)
```



## 8: Optimization Function Reference

### Problem Definition Functions

**LpDefine**(*Vars*, *constraints: IndexType*;  
*objCoef: Numeric[Vars]*;  
*lhs: Numeric[Vars,Constraints]*;  
*rhs: Numeric[Constraints]*;  
*sense: Optional TextType[Constraints]*;  
*maximize: Optional Boolean*;  
*lb,ub: Optional Numeric[Vars]*;  
*ctype: Optional TextType[Vars]*;  
*ItLimit, NdLimit, MipLimit,*  
*TimeLimit: Optional Positive*;  
*OptTolerance, PivotTolerance, FeasTolerance,*  
*GapTolerance: Optional Positive*;  
*OptLb,OptUb: Optional Numeric*;  
*scaling: Optional Numeric*)

Defines a linear optimization program. See the “3: Formulating an Optimization Problem” on page 19 for a description of usage and parameters.

**QpDefine**(*Vars*, *Vars2*, *constraints: IndexType*;  
*c: Numeric[Vars]*;  
*Q: Numeric[Vars,Vars2]*;  
*lhs: Numeric[Vars,Constraints]*;  
*rhs: Numeric[Constraints]*;  
*sense: Optional TextType[Constraints]*;  
*maximize: Optional Boolean*;  
*lb,ub: Optional Numeric[Vars]*;  
*ctype: Optional TextType[Vars]*;  
*guess: Optional Numeric[Vars]*;  
*warnIndefinite: Optional Boolean*;  
*ItLimit, NdLimit, MipLimit,*  
*TimeLimit: Optional Positive*;  
*OptTolerance, PivotTolerance, FeasTolerance,*  
*GapTolerance: Optional Positive*;  
*OptLb,OptUb: v Numeric*;  
*scaling: Optional Numeric*)

Defines a quadratic optimization program. See the “3: Formulating an Optimization Problem” on page 19 for a description of usage and parameters.

**NlpDefine**(Vars, constraints: IndexType;  
 x: IVarType;  
 obj, lhs: Expression;  
 rhs: Numeric[Constraints];  
 sense: Optional TextType[Constraints];  
 maximize: Optional Boolean;  
 lb,ub: Optional Numeric[Vars];  
 ctype: Optional TextType[Vars];  
 guess: Optional Numeric[Vars];  
 gradient, jacobian: Optional Expression;  
 objnl: Optional TextType[Vars];  
 lhsnl: Optional TextType[Vars, Constraints];  
 itLimit, noImpSeconds, timeLimit,  
 convTolerance: Optional Positive;  
 mutate: Optional Positive;  
 linVar: Optional Scalar;  
 DerivMethod, EstimMethod, DirecMethod:  
 Optional TextType;  
 SampSz: Optional Positive)

Defines a non-linear optimization problem. See the “3: Formulating an Optimization Problem” on page 19 for a description of usage and parameters.

## Other Functions

### LpFindIIS(lp: LpType)

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.

### 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: TextType;  
Vars, constraints: Optional IndexType)**

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.

**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:

**LpStatusNum() Return Values**

Status	Description (LpStatusText)
1	Optimal solution found
2	No feasible solution
3	Objective unbounded
5	Iteration limit exceeded, feasible
6	Iteration limit exceeded, not yet feasible
7	Time limit exceeded, feasible
8	Time limit exceeded, not yet feasible
65	Objective function changing too slowly
66	All remedies failed to find a better point
67	Error in evaluating problem functions
68	Could not allocate enough memory
69	Attempt to re-enter Optimizer engine during solution
101	The MIP optimal solution found
102	MIP solution found within gap tolerance (see "Controlling The Search" on page 32)
103	No feasible integer solution
104	Integer solution limit exceeded
105	Node limit exceeded, feasible
106	Node limit exceeded, not feasible
107	Time limit exceeded, feasible
108	Time limit exceeded, not feasible

**LpWrite(lp: LpType; filename: TextType)**

Writes a TextType 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**.

**LpWritelis(lp: LpType; filename: TextType)**

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()**.

